Coordination in Intelligent Grid Environments

XIN BAI, HAN YU, GUOQIANG WANG, YONGCHANG JI, GABRIELA M. MARINESCU, DAN C. MARINESCU, SENIOR MEMBER, IEEE, AND LADISLAU BÖLÖNI, MEMBER, IEEE

Invited Paper

A computational grid is a complex system. The state space of a complex system is very large and it is infeasible to create a rigid infrastructure implementing optimal policies and strategies which take into account the current state of the system. An alternative to a rigid infrastructure is to base the system's reactions on logical inference, planning, and learning, the quintessential elements of an intelligent system. An intelligent grid is one where societal services exhibit intelligent behavior. A coordination service acting as a proxy on behalf of end users reacts to unforeseen events, plans how to carry out complex tasks, and learns from the history of the system. Various policies implemented by the societal services of an intelligent grid, such as brokerage and matchmaking, are based upon rules and facts gathered with the aid of a monitoring service. The question we address is how to construct intelligent computational grids which are truly scalable and could respond to the needs of a diverse user community. We present a prototype of a system used for a virtual laboratory in computational biology.

Keywords*—Agent, coordination, grid, knowledge, intelligent, ontology.*

I. Introduction and Motivation

Data, service, and computational grids, collectively known as information grids, are collections of autonomous computers connected to the Internet and giving to individual users the appearance of a single virtual machine [6], [8].

A *data grid* allows a community of users to share content. An example of a specialized data grid supporting a relatively small user community is the one used to share data from high-energy physics experiments. The World Wide Web can be viewed as a data grid populated with HTTP servers providing the content, data, audio, and video.

Manuscript received March 1, 2004; revised June 1, 2004. This work was supported in part by the National Science Foundation under Grants MCB9527131, DBI0296107, ACI0296035, and EIA0296179.

X. Bai, H. Yu, G. Wang, Y. Ji, G. M. Marinescu, and D. C. Marinescu are with the School of Computer Science, University of Central Florida, Orlando, FL 32816-2362 USA (e-mail: xbai@cs.ucf.edu; hyu@cs.ucf.edu; gwang@cs.ucf.edu; yji@cs.ucf.edu; magda@cs.ucf.edu; dcm@cs.ucf.edu).

L. Bölöni is with the Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816-2450 USA (e-mail: lboloni@cpe.ucf.edu).

Digital Object Identifier 10.1109/JPROC.2004.842770

A *service grid* will support applications such as electronic commerce, telemedicine, distance learning, and business-to-business. Such applications require a wide spectrum of end services such as monitoring and tracking, remote control, maintenance and repair, online data analysis, and business support, as well as services involving some form of human intervention such as legal, accounting, and financial services. An application of a monitoring service in health care could be monitoring outpatients to ensure that they take the prescribed medication. Controlling the heating and cooling system in a home to minimize energy costs, periodically checking the critical parameters of the system, ordering parts such as air filters, and scheduling repairs is an example of control, maintenance, and repair services, respectively. Data analysis services could be used in conjunction with arrays of sensors to monitor traffic patterns or to document a visitor's interest at an exhibition. There are qualitative differences between service and data grids. The content in a service grid is more dynamic; it is often the result of a cooperative effort of a number of service providers, it involves a large number of sensors, and it is tailored to specific user needs. To create the dynamic content, we need some form of dynamic service composition. Dynamic service composition has no counterpart in the current Web, where portals support static service coordination.

A *computational grid* is expected to provide transparent access to computing resources for applications requiring a substantial CPU power, very large memories, and secondary storage that cannot be provided by a single system. The requirements placed on the user access layer and on societal (core) services are even more stringent for a computational grid than for a service grid. The user access layer must support various programming models and the societal services of a computational grid must be able to handle low-level resource management.

There are many similarities between data, service, and computational grids and it is highly desirable for the three to share as many standards, architectural concepts, and even components, as practical. It seems very unfortunate that

for many years research in computational grids had a very loose connection with the mainstream efforts of the World Wide Web Consortium (W3C). Recently, a more rational approach is noticeable, e.g., the Globus project has embraced standards developed years ago, such as Web Services Definition Language (WSDL) and Simple Object Access Protocol (SOAP). WSDL is an XML format for describing network services as a set of endpoints operating on messages containing document- or procedure-oriented information. WSDL supports an abstract description of the operations and messages exchanged among endpoints. Both operations and messages are bound to a *concrete network protocol and message format* to define a *concrete endpoint*. One or more concrete endpoints are combined into *services* or abstract endpoints. WSDL description of the endpoints and their messages is independent of the message formats or network protocols used to communicate. SOAP is an XML-based application layer protocol developed as a standard by W3C. It is extensible, application-, and platform-independent. There are also important dissimilarities. For example, the service requests in a computational grid require a much finer granularity of resource allocation [9].

The focus of this paper is computational grids though many of the ideas discussed in the next sections are generic and could be applied to the other types of grids. For example, coordination is a problem likely to surface in the context of service grids as well.

A computational grid is a complex system. The state space of a complex system is very large and it is infeasible to create a rigid control infrastructure implementing optimal policies and strategies which take into account the current state of the system. An alternative to a rigid control infrastructure is to base the system's reactions on logical inference, planning, and learning, the quintessential elements of an intelligent system. An intelligent grid is one where societal services exhibit intelligent behavior. A coordination service acting as a proxy on behalf of end users reacts to unforeseen events, plans how to carry out complex tasks, and learns from the past history of the system. Various policies implemented by the societal services of an intelligent grid, such as brokerage and matchmaking, are based upon rules and facts gathered with the aid of a monitoring service. The question we address is how to construct intelligent computational grids which are truly scalable and could respond to the needs of a diverse user community.

The contribution of this paper is an in-depth discussion of intelligent computational grids, an analysis of some core services, the presentation of the basic architecture of the middleware we are currently constructing, and applications of the system to a complex computation. This paper is organized as follows: first, we discuss some of the most important requirements for the development of intelligent grids and present in some depth the problem of coordination and coordination services on a grid. Then we review the information, brokerage, matchmaking, planning, and the event services. Finally, we present the BondGrid and an application of it to computational structural biology.

A. Defining Characteristics of Computational Grids

The defining characteristics of computational grids are: 1) resource sharing among a large user population and 2) support for collaborative activities [19]. In the context of a computational grid, the term *resource* is used in a wide sense; it means hardware and software resources, services, and content. *Content* generally means some form of static or dynamic data or knowledge. *Autonomy* implies that the resources are in different domains and resource sharing requires cooperation between the administrative authorities in each domain.

Computational grids inherit many of the traditional attributes of the Internet. Among the characteristics of computational grids which distinguish them from the more traditional distributed systems of the past decades [19], we note the following.

1) Scale. A grid may consist of tens of thousands, or more, nodes.
2) Heterogeneity and diversity. Nodes with different processor and system architectures are expected to populate the grid. The communication channels linking these nodes differ in terms of latency and bandwidth. The operating systems (OS) of individual nodes may be different. The application software running on the nodes are very diverse; multiple versions of the same application software may be available.
3) Autonomy of individual nodes. The nodes are in different administrative domains possibly with different access, security, and resource management policies [9].
4) The dynamic and open-ended character. The grid evolves in time; new resources are constantly added to the grid, existing ones are modified, others are retired.
5) The dominant service policy in the grid is based upon a "best effort." Enforcing end-to-end quality of service constraints is rarely possible.
6) A large user population with individual and often conflicting objectives.
7) User's requirements may be dynamic, subject to change, or even cannot be known *a priori*.
8) Complex, resource-intensive tasks submitted by individual users [10]. The complexity of a task is rather difficult to quantify. It has multiple facets. It may refer to the number and relationship of component activities, the predictability of the amount of resources needed for the completion of individual activities, the security constraints, the presence or absence of soft deadlines, the duration of individual activities, the diversity of resources used, and so on [9].

B. Resource Management, Exception Handling, and Coordination

Whenever there is a contention for a limited set of resources among a group of entities or individuals, we need control mechanisms to mitigate access to system resources. These control mechanisms enable a number of desirable properties of the system (e.g., fairness) provide guarantees that tasks are eventually completed, and ensure timeliness

when timing constraints are involved. Security is a major concern in such an environment. We want to ensure confidentiality of information and prevent denial of service attacks, while allowing controlled information sharing for cooperative activities. Considerably simpler versions of some of the problems mentioned above are encountered at the level of a single system, or in the case of small-scale distributed systems (systems with a relatively small number of nodes in a single administrative domain). In the case of a single system, such questions are addressed by the operating system which transforms the "bare hardware" into a user machine and controls access to system resources. The question of how to address these problems in the context of a grid has been the main focus of research in grid environments, and, at the same time, the main stumbling block in the actual development of computational grids.

Some research in Grid computing proposes to transfer to Grid computing concepts, services, and mechanisms from traditional operating systems, or from parallel and distributed systems without taking into account the effect on system reliability and dependability of the specific attributes 1–7 discussed above, a clearly inadequate approach. For example, there is a proposal to extend the Message Passing Interface (MPI) to a Grid environment. In its current implementation, the MPI does not have any mechanism to deal with a node failure during a barrier synchronization operation. In such a case, all the nodes involved other than the defective one wait indefinitely, and it is the responsibility of the user to detect the failure and take corrective actions. It may be acceptable to expect the programmer to monitor a cluster with a few hundred nodes housed in the next room, but it is not reasonable to expect someone to monitor tens of thousands of nodes scattered over a large geographic area. Thus, we cannot allow MPI to work across system boundaries without any fault detection mechanism.

Coordination allows individual components of a system to work together and create an ensemble exhibiting a new behavior without introducing a new state at the level of individual components. Scripting languages provide a "glue" to support composition of existing applications. The problem of coordinating concurrent tasks was generally left to the developers of the parallel scientific and engineering applications. Coordination models such as the coordinator–worker or the widely used Same Program Multiple Data (SPMD) were developed in that context.

The problem of coordination of complex tasks has new twists in the context of Grid computing. First, it is more complex and it involves additional activities such as resource discovery and planning. Second, it has a much broader scope due to the scale of the system. Third, the complexity of the computational tasks and the fact that the end user may only be intermittently connected to the network force us to delegate this function to a proxy capable of creating the conditions for the completion of the task with or without user intervention. It is abundantly clear that such a proxy is faced with very delicate decisions regarding resource allocation or exception handling. For example, should we use a more expensive resource and pay more to have guarantees that a task completes in time, or should we take our chances with a less expensive resource? In the case of the MPI example, should we kill all the processes in all the nodes and restart the entire computation, should we roll back the computation to a previous checkpoint if one exists, or should we simply restart the process at the failing node on a different node?

There is little doubt that the development of computational grids poses formidable problems. In this paper, we concentrate on problems related to resource management, exception handling, and coordination of complex tasks. We argue that only an intelligent environment could reliably and seamlessly support such functions.

II. Intelligent Grid Environments

Most of the research in grid computing is focused on relatively small grids (hundreds of nodes) dedicated to a rather restricted community (e.g., high-energy physics), of well-trained users (e.g., individuals working in computational sciences and engineering), with a rather narrow range of problems (e.g., computer-aided design for the aerospace industry).

The question we address is whether a considerably larger grid could respond to the needs of a more diverse user community than in the case of existing grids without having some level of intelligence built into the core services. The reasons we consider such systems are precisely the reasons computational grids were introduced in the first place: economy of scale and the ability to share expensive resources among larger groups of users. It is not uncommon that several groups of users (e.g., researchers, product developers, individuals involved in marketing, educators, and students) need a seamless and controlled access to existing data or to the programs capable of producing data of interest. For example, the structural biology community working on the atomic structure determination of viruses, the pharmaceutic industry, and educational institutions ranging from high schools to universities, need to share information. One could easily imagine that a high school student would be more motivated to study biology if he or she were able to replay in the virtual space successful experiments done at the top research laboratories, leading to the discovery of the structure of a virus (e.g., the common cold virus) and understand how a vaccine to prevent the common cold is engineered.

An intelligent environment is in a better position than a traditional one to match the user profile (leader of a research group, member of a research group with a well-defined task, drug designer, individual involved in marketing, high school student, or doctoral student) with the actions the user is allowed to perform and with the level of resources he or she is allowed to consume. At the same time, an intelligent environment is in a better position to hide the complexity of the Grid infrastructure and allow unsophisticated users, such as a high school student without any training in computational science, to carry out a rather complex set of transformations of an input data set.

Even in the simple example discussed above, we see that the *coordination service* acting as a proxy on behalf of the

end user has to deal with unexpected circumstances, or with error conditions (e.g., the failure of a node). The response to such an abnormal condition can be very diverse, ranging from terminating the task to restarting the entire computation from the very beginning or from a checkpoint. Such decisions depend upon a fair number of parameters, e.g., the priority of the task, the cost of each option, the presence of a soft deadline, and so on. Even in this relatively simple case, it is nontrivial to hardcode the decision making process into a procedure written in a standard programming language. Moreover, we may have in place different policies to deal with rare events, policies which take into account factors such as legal considerations, the identity of the parties involved, the time of day, and so on. At the same time, hardcoding the decision making will strip us of the option to change our actions depending upon considerations we did not originally take into account, such as the availability of a new system just connected to the grid.

Very often the computations carried out on a grid involve multiple iterations, and in such a case the duration of an activity is data dependent and very difficult to predict. Scheduling a complex task whose activities have unpredictable execution times requires the ability to discover suitable resources available at the time when activities are ready to proceed. It also requires market-based scheduling algorithms, which in turn require metainformation about the computational tasks and the resources necessary to carry out such tasks.

The more complex the environment, the more elaborate the decision making process becomes, because we need to take into account more factors and circumstances. It seems obvious to us that under such circumstances a set of inference rules based upon facts reflecting the current status of various Grid components are preferable to hardcoding. Often, we also need to construct an elaborate plan to achieve our objective or to build learning algorithms into our systems.

Reluctant as we may be to introduce AI components into a complex system such as a grid, we simply cannot ignore the benefits the AI components could bring along. Inference, planning, and learning algorithms are notoriously slow and cannot be used when faced with fast approaching deadlines. We should approach their use with caution.

The two main ingredients of an intelligent grid are software agents and ontologies. A *software agent* is a special type of reactive program. Some of the actions taken by the agent are in response to external events; other actions may be taken at the initiative of the agent. The defining attributes of a software agent are autonomy, intelligence, and mobility. *Autonomy*, or agency, is determined by the nature of the interactions between the agent and the environment and by the interactions with other agents and/or the entities they represent. *Intelligence* is measured by the degree of reasoning, planning, and learning the agent is capable of. *Mobility* reflects the ability of an agent to migrate from one host to another in a network.

An agent may exhibit different degrees of autonomy, intelligence, and mobility. For example, an agent may have inferential abilities, but little or no learning and/or planning abilities. An agent may exhibit strong or weak mobility; in the first case, an agent may be able to migrate to any site at any time; in the second case, the migration time and sites are restricted.

Software agents have unique abilities to perform the following functions.

1) Support intelligent resource management. Peer agents can negotiate access to resources and request services based upon user intentions rather than specific implementations.
2) Support intelligent user interfaces. We expect agents to be capable of composing basic actions into higher level ones, to be able to handle large search spaces, to schedule actions for future points in time, and to support abstractions and delegations. Some of the limitations of direct manipulation interfaces, namely, difficulties in handling large search spaces, rigidity, and the lack of improvement of behavior, extend to most other facets of traditional approaches to interoperability.
3) Filter large amounts of information. Agents can be instructed at the level of goals and strategies to find solutions to unforeseen situations, and they can use learning algorithms to improve their behavior.
4) Adjust to the actual environment. Agents are network aware.
5) Move to the site when they are needed and, thus, reduce communication costs and improve performance.

The Grid community seems ready to accept the need of metainformation to facilitate the interpretability of various components, so we believe that sooner rather than later we shall witness an effort to build intelligent Grid environments.

The critical components of such an intelligent Grid environment are the *ontologies*, collections of structured data shared among different agents. Core services in such grids are provided by intelligent agents, programs with the ability to perform intelligent actions such as inference, planning, and possibly learning.

The need for an intelligent infrastructure is amply justified by the complexity of both the problems we wish to solve and the characteristics of the environment [18]. As we take a closer look at the architecture of an intelligent grid, we distinguish between several classes of services. Systemwide services supporting coordinated and transparent access to resources of an information grid are called *societal* or *core* services. Specialized services accessed directly by end users are called *end-user services*. The core services, provided by the computing infrastructure, are persistent and reliable, while end-user services could be transient in nature. The providers of end-user services may temporarily, or permanently, suspend their support. The reliability of end-user services cannot be guaranteed. The basic architecture of an intelligent grid is illustrated in Fig. 1.

A nonexhaustive list of core services includes authentication, brokerage, coordination, information, ontology, matchmaking, monitoring, planning, persistent storage, scheduling, event, and simulation. *Authentication* services contribute to the security of the environment. *Brokerage*

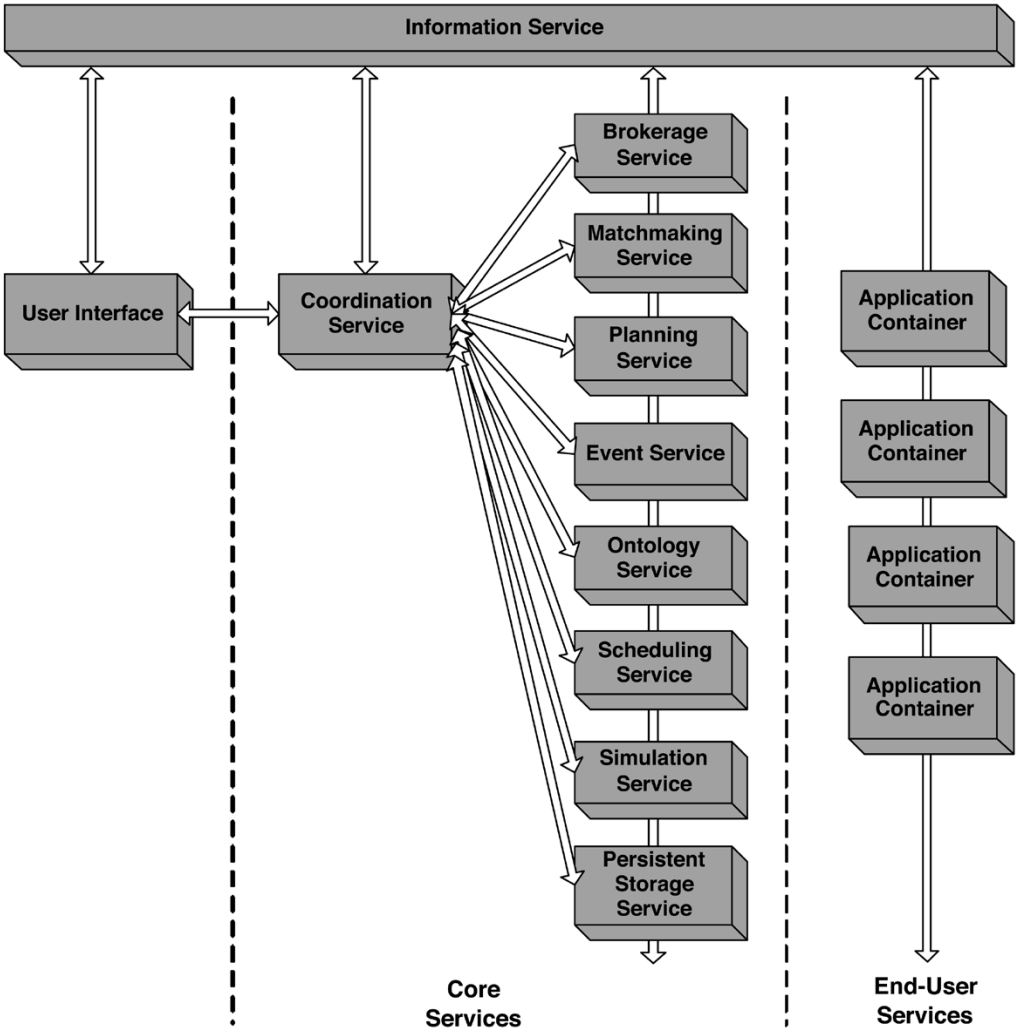


Fig. 1. Core and end-user services. The User Interface provides access to the environment. Applications Containers host end-user services. Shown are the following core services: Coordination Service, Information Service, Planning Service, Matchmaking Service, Brokerage Service, Event Service, Ontology Service, Simulation Service, Scheduling Service, and Persistent Storage Service.

services maintain information about classes of services offered by the environment, as well as past performance databases. Though the brokerage services make a best effort to maintain accurate information regarding the state of resources, such information may be obsolete. Up-to-date information about the status of any resource can be gathered using *monitoring* services. *Coordination* services act as proxies for the end user. A coordination service receives a case description and controls the enactment of the workflow. *Planning* services are responsible for creating the workflow. *Scheduling* services provide optimal schedules for sites offering to host application containers for different end-user services. *Information* services play an important role; all end-user services register their offerings with the information services. *Ontology* services maintain and distribute *ontology shells*, i.e., ontologies with classes and slots but without instances, as well as ontologies populated with instances, global ontologies, and user-specific ontologies. *Matchmaking* services allow individual users represented by their proxies (coordination services) to locate resources in a spot market, subject to a wide range of conditions. Individual users may only be intermittently connected to the network. *Persistent storage* services provide access to the data needed for the execution of user tasks. *Event* services provide a method for event handling and message passing. *Simulation* services are necessary to study the scalability of the system and are also useful for end users to simulate an experiment before actually conducting it.

Core services are replicated to ensure an adequate level of performance and reliability. Core services may be organized hierarchically in a manner similar to the Domain Name Services (DNSs) on the Internet. End-user services could be transient in nature. The providers of such services may temporarily or permanently suspend their support, while most core services are guaranteed to be available at all times. Content-provider services, legal, accounting, tracking, and various application software are examples of end-user services.

III. Coordination and Coordination Services

A cursory examination of recent research in Grid computing services reveals limited interest in the mechanisms used to coordinate the execution of complex computational tasks. This situation can be attributed to several reasons: 1) there are relatively few applications of Grid computing; 2) the users of Grid computing are highly computationally sophisticated individuals with a high threshold for pain in using computer systems; 3) powerful scripting languages such as Perl, Python, and tuple-spaces can be used for coordination [13]; and, last but not least, 4) the problems posed by a coordination service are nontrivial.

A. Process Coordination

Coordination is a very broad subject with applications to virtually all areas of science and engineering, management, social systems, defense systems, education, health care, and so on. Human life is an exercise in coordination: each individual has to coordinate his own activities with the activities of other individuals, and groups of individuals have to coordinate their efforts to achieve a meaningful result.

Coordination is critical for the design and engineering of new man-made systems and important for understanding the behavior of existing ones [11]. Coordination is an important dimension of computing. An algorithm describes the flow of control, the flow of data, or both; a program implementing the algorithm coordinates the software and the hardware components involved in a computation. The software components are library modules interspaced with user code executed by a single thread of control in case of sequential computations; in this case, the hardware is controlled through system calls supported by the operating system running on the target hardware platform.

Coordination of distributed and/or concurrent computations is more complex; it involves software components based on higher level abstractions such as objects, agents, and programs as well as multiple communication and computing systems. We can consider a three-dimensional (3-D) space for coordination models. The first dimension reflects the type of the network, i.e., an interconnection network of a parallel system, a local area network, or a wide area network; the second dimension describes the type of coordination, i.e., centralized, or distributed; the third dimension reflects the character of the system, i.e., closed or open.

A computer network provides the communication substrate. and its characteristics provide the first dimension of a coordination space. The individual entities can be colocated in space within a single system. They can be distributed over a LAN, or over a WAN. Coordination in a WAN is a more difficult problem than coordination confined to a LAN or to a single system; we have to deal with multiple administrative domains and in theory communication delays are unbounded. In a WAN, it is more difficult to address performance, security, or quality of service issues.

There are two approaches to coordination, a centralized and a distributed one. Centralized coordination is suitable in some instances such as *ad hoc service composition*. Suppose that one user needs a super service involving several services; in this case an agent acting on behalf of the user coordinates the composition. In other cases, a distributed coordination approach has distinct benefits. Consider, for example, a complex weather service with a very large number of sensors, of the order of millions, gathering weather-related data. The system uses information from many databases; some contain weather data collected over the years, others archive weather models. The system generates short-, medium-, and long-term forecasts. Different functions of this service such as data acquisition, data analysis, data management, and weather information dissemination will most likely be coordinated in a distributed fashion. A hierarchy of coordination centers will be responsible for data collected from satellites, another group will coordinate terrestrial weather stations, and yet another set of centers will manage data collected by vessels and sensors from the oceans.

The last dimension of interest of the coordination space reflects whether the system is: 1) closed and all entities involved are known at the time when the coordination activity is initiated or 2) open and allows new entities to join or leave at will. Coordination in an open system is more difficult than coordination in a closed system. Error recovery and fault tolerance become a major concern because a component may suddenly fail or leave the system without prior notice. The dynamics of coordination changes; we cannot stick to a precomputed coordination plan and we may have to revise it. The state of the system must be reevaluated frequently to decide if a better solution involving components that have recently joined the system exists.

B. Coordination Techniques

We distinguish between low-level and high-level coordination issues. Low-level coordination issues are centered on the delivery of coordination information to the entities involved; high-level coordination covers the mechanisms and techniques leading to coordination decisions.

The more traditional distributed systems are based on *direct communication models*, like the one supported by remote procedure call protocols, to implement the client-server paradigm. A client connected to multiple servers is an example of a simple coordination configuration; the client may access services successively, or a service may in turn invoke additional services.

In this model, there is a direct coupling between interacting entities in terms of name, place, and time. To access a service, a client needs to know the name of the service and the location of the service; the interaction spans a certain time interval.

Mediated coordination models ease some of the restrictions of the direct model by allowing an intermediary (e.g., a directory service) to locate a server, an event service to support asynchronous execution, an interface repository to discover the interface supported by the remote service, brokerage and matchmaking services to determine the best match between a client and a set of servers, and so on.

1) Coordination Based on Scripting Languages: Coordination is one application of a more general process

called software composition where individual components are made to work together and create an ensemble exhibiting a new behavior, without introducing a new state at the level of individual components. A component is a black box exposing a number of interfaces allowing other components to interact with it.

The components or entities can be "glued" together with scripts. Scripting languages provide "late gluing" of existing components. Several scripting languages are very popular: `Tcl`, `Perl`, `Python`, `JavaScript`, `AppleScript`, `Visual Basic`, languages supported by the `csh` or the `Bourne` Unix shells.

Scripting languages share several characteristics [14].

1) Support composition of existing applications; thus, the term "late gluing." For example, we may glue together a computer-aided design (CAD) system, with a database for material properties (MPDB). The first component may be used to design different mechanical parts; then the second may be invoked to select for each part the materials with desirable mechanical, thermal, and electrical properties.
2) Rely on a virtual machine to execute bytecode `Tcl`, or interpreted languages. `Perl`, `Python`, and `Visual Basic` are based on a bytecode implementation, whereas `JavaScript, AppleScript`, and the `Bourne Shell` need an interpreter.
3) Favor rapid prototyping over performance. In the previous example in 1), one is likely to get better performance in terms of response time by rewriting and integrating the two software systems, but this endeavor may require several man-years; writing a script to glue the two legacy applications together could be done in days.
4) Allow the extension of a model with new abstractions. For example, if one of the components is a CAD tool producing detailed drawings and specifications of the parts of an airplane engine, then the abstractions correspond to the airplane parts (e.g., wing, tail section, and landing gear). Such high-level, domain-specific abstractions can be easily understood and manipulated by aeronautic and mechanical engineers with little or no computer science background.
5) Generally, scripting languages are weakly typed and offer support for introspection and reflection and for automatic memory management.

`Perl, Python, JavaScript, AppleScript,` and `Visual Basic` are object-based scripting languages. All five of them are embeddable and can be included into existing applications. For example, code written in `Jython`, a Java version of `Python`, can be embedded into a data stream, sent over the network, and executed by an interpreter at the other site.

`Perl, Python,` and `JavaScript` support introspection and reflection. Introspection and reflection allow a user to determine and modify the properties of an object at runtime.

Scripting languages are very popular and widely available. `Tcl`, `Perl`, and `Python` are available on most platforms, `JavaScript` is supported by all popular Web browsers, the `Bourne Shell` is supported by Unix, and `Visual Basic` by Windows.

Script-based coordination has obvious limitations. It is most suitable for applications with one coordinator acting as an enactment engine, or in a hierarchical scheme when the legacy applications form the leaves of the tree and the intermediate nodes are scripts controlling the applications in a subtree. A script for a dynamic system, where the current state of the environment determines the course of action, becomes quickly very complex. Building some form of fault tolerance and handling exceptions could be very tedious.

In summary, script-based coordination is suitable for simple, static cases and has the advantage of rapid prototyping but could be very tedious and inefficient for more complex situations.

2) Coordination Based on Shared-Data Spaces: Tuple space coordination has also been intensely scrutinized [4], [13]. A shared-data space allows agents to coordinate their activities. We use the terminology *shared-data space* because of its widespread acceptance, though in practice the shared space may consist of data, knowledge, code, or a combination of them. The term *agent* means a party to a coordination effort.

In this coordination model, all agents know the location of a shared data space and have access to communication primitives to deposit and to retrieve information from it. As in virtually all other coordination models, a prior agreement regarding the syntax and the semantics of communication must be in place before meaningful exchanges of coordination information may take place.

The shared-data space coordination model allows *asynchronous communication* between mobile agents in an open system. The communicating components need not be coupled in time or space. The producer and the consumer of a coordination information item act according to their own timing; the producer agent may deposit a message at its own convenience and the consumer agent may attempt to retrieve it according to its own timing. The components need not be colocated; they may even be mobile. The only constraint is for each agent to be able to access the shared-data space from its current location. Agents may join and leave the system at will.

Another distinctive advantage of the shared-data space coordination model is its tolerance of heterogeneity. The implementation language of the communicating entities, the architecture, and the operating systems of the hosts where the agents are located play no role in this model. An agent implemented in Java, running in a Linux environment and on a SPARC-based platform, could interact with another one implemented in C++, running under Windows on a Pentium platform, without any special precautions.

Traditionally, a shared-data space is a *passive* entity, coordination information is *pushed* into it by a source agent and *pulled* from it by a destination agent. The amount of state information maintained by a shared-data space is minimal; it

does not need to know either the location or even the identity of the agents involved. Clearly, there are applications where security concerns require controlled access to the shared information; thus, some state information is necessary. These distinctive features make this model scalable and extremely easy to use.

An alternative model is based on *active* shared-data spaces; here the shared-data space plays an active role as it informs an intended destination agent when information is available. This approach is more restrictive and requires the shared-data space to maintain information about the agents involved in the coordination effort. In turn, this makes the system more cumbersome, less scalable, and less able to accommodate mobility.

Linda [4] was the first system supporting associative access to a shared-data space. *Associative access* raises the level of communication abstraction. Questions such as who produced the information, when was it produced, and who were the intended consumers are no longer critical and applications that do not require such knowledge benefit from the additional flexibility of associative access.

Tuples are ordered collections of elements. In a shared-tuple space agents use templates to retrieve tuples; this means that an agent specifies what type of tuple to retrieve, rather than a specific tuple.

Linda supports a set of primitives to manipulate the shared tuple space; `out` allows an agent to deposit or write a tuple with multiple fields in the tuple space; `in` and `rd` are used to read or retrieve a tuple when a matching has been found; `inp` and `rdp` are nonblocking versions of `in` and `rd`; `eval` is a primitive to create an *active* tuple, one with fields that do not have a definite value but are evaluated using function calls.

Several types of systems extend some of the capabilities of Linda. Some, including `T Spaces` from IBM [15] and `JavaSpaces` from Sun Microsystems, extend the set of coordination primitives, others affect the semantics of the language, yet another group modifies the model. For example, `T Spaces` allows database indexing and event notification, supports queries expressed in the structured query language (SQL), and allows direct thread access when the parties run on the same Java virtual machine.

A survey of the state-of-the-art tuple-based technologies for coordination and a discussion of a fair number of systems developed in the last few years are presented in [13]. Several papers referred in [11] provide an in-depth discussion of tuple space coordination.

3) Coordination Based on Middleware Agents: In our daily life, middlemen facilitate transactions between parties, help coordinate complex activities, or simply allow one party to locate other parties. For example, a title company facilitates real estate transactions; wedding consultants and planners help organize a wedding; an auction agency helps sellers locate buyers and help buyers find items they desire.

So it is not very surprising that a similar organization appears in complex software systems. The individual components of the system are called *entities* whenever we do not want to be specific about the function attributed to each component; they are called clients and servers when their function is well defined.

Coordination can be facilitated by agents that help locate the entities involved in coordination, and/or facilitate access to them. Brokers, matchmakers, and mediators are examples of middle agents used to support reliable mediation and to guarantee some form of end-to-end quality of service (QoS). In addition to coordination functions, such agents support interoperability and facilitate the management of knowledge in an open system.

A *broker* is a middle agent serving as an intermediary between two entities involved in coordination. All communications between the entities are channeled through the broker.

A broker does not actively collect information about the entities active in the environment. Each entity has to make itself known by registering itself with the broker before it can be involved in mediated interactions.

Entities may provide additional information such as a description of services or a description of the semantics of services. A broker may maintain a knowledge base with information about individual entities involved and may even translate the communication from one party into a format understood by the other parties involved.

A matchmaker is a middle agent whose only role is to pair together entities involved in coordination; once the pairing is done, the matchmaker is no longer involved in any transaction between the parties. For example, a matchmaker may help a client select a server. Once the server is selected, the client communicates directly with the server bypassing the matchmaker.

The matchmaker has a more limited role than a broker; while the actual selection may be based on a QoS criterion, once made, the matchmaker cannot provide additional reliability support. If one of the parties fails, the other party must detect the failure and again contact the matchmaker. A matchmaker, like a broker, does not actively collect information about the entities active in the environment; each entity has to make itself known by registering itself with the matchmaker.

A mediator can be used in conjunction with a broker or a matchmaker to act as a front end to an entity. In many instances, it is impractical to mix the coordination primitives with the logic of a legacy application, e.g., a database management system. It is easier for an agent to use a uniform interface for an entire set of systems designed independently than to learn the syntax and semantics of the interface exposed by each system. A solution is to create a wrapper for each system and translate an incoming request into a format understood by the specific system it is connected to; at the same time, responses from the system are translated into a format understood by the sender of the request.

Agent-based coordination and coordination of agent federations have been investigated by several groups [3], [12].

C. Process Coordination and Workflow Management

Grid users have complex tasks and want to take advantage of the resource-rich environment provided by the grid to solve their problems subject to a set of constrains such as

deadlines, cost, and quality of the solution. A complex task consists of multiple activities. *Activities* are units of work to be performed by the agents, humans, computers, sensors, and other man-made devices. A *process description* is a structure describing the activities to be executed and the order of their execution. A process description contains one start and one end symbol and includes various patterns [1].

The term *workflow* has been used for some time in the business community to describe a complex task. Originally, workflow management was considered a discipline confined to the automation of business processes. Today most business processes depend on the Internet and workflow management has evolved into a network-centric discipline. The scope of workflow management has broadened. The basic ideas and technologies for automation of business processes can be extended to virtually all areas of human endeavor from science and engineering to entertainment.

Production, administrative, collaborative, and *ad hoc* workflows require that documents, information, or tasks be passed from one participant to another for action, according to a set of procedural rules. Production workflows manage a large number of similar tasks with the explicit goal of optimizing productivity. Administrative workflows define processes, while collaborative workflows focus on teams working toward common goals. E-commerce and business-to-business are probably the most notable examples of Internet-centric applications requiring some form of workflow management. E-commerce has flourished in recent years; many businesses encourage their customers to order their products online and some, including PC makers, only build their products on demand. Various business-to-business models help companies reduce their inventories and outsource major components.

There are several distinctions between grid-based workflows and traditional workflows encountered in business management, office automation, or production management (see [8]).

1) The emphasis in a traditional workflow model is placed on the contractual aspect of a transaction. For a grid-based workflow, the enactment of a case is sometimes based on a "best-effort model" where the agents involved do their best to attain the goal state but there is no guarantee of success.
2) An important aspect of a transactional model is to maintain a consistent state of the system. A grid is an open system; thus, the state of a grid is considerably more difficult to define than the state of a traditional system.
3) A traditional workflow consists of a set of well-defined activities that are unlikely to be altered during the enactment of the workflow. However, the process description for a grid-based workflow may change during the lifetime of a case. After a change, the enactment of a case may continue based on the older process description, while under some circumstances it may be based on the newer process description. In addition to static workflows, we have to support dynamic ones.
4) The activities of a grid-based workflow could be long lasting. Some of the activities supported by the grid are collaborative in nature, and the workflow management should support some form of merging of partial process descriptions.
5) The individual activities of a grid workflow may not exhibit the traditional properties of transactions. Consider, for example, durability; at any instance of time before reaching the goal state a workflow may roll back to some previously encountered state and continue from there on an entirely different path. An activity of a grid workflow could be either *reversible* or *irreversible*. Sometimes, paying a penalty for reversing an activity is more profitable in the long run than continuing on a wrong path.
6) Resource allocation is a critical and very delicate aspect of the grid-based workflow enactment. The grid provides a resource-rich environment with multiple classes of resources and many administrative domains; there is a large variability of resources in each class; resource utilization is bursty in nature. Thus, we need resource discovery services, support for negotiations among multiple administrative domains, matchmaking and brokerage services, reservations mechanisms, support for dynamic resource allocation, and other sophisticated resource management mechanisms and services.
7) Mobility of various agents involved in the enactment of a case is important for grid-based workflows [8]. The agents may relocate to the proximity of the sites where activities are carried out to reduce communication costs and latency.

A workflow has three dimensions.

1) The process; the *process dimension* refers to the creation and the eventual modification of the process description.
2) The case; the *case dimension* refers to a particular instance of the workflow when the attributes required by the process enactment are bound to specific values.
3) The resources; the *resource dimension* refers to discovery and allocation of resources needed for the enactment of a case.

Workflow enactment is the process of carrying out the activities prescribed by the process description for a particular case.

The creation of a process description is similar to writing a program, the instantiation of a case is analogous to the execution of the program with a particular set of input data, while resource allocation has no direct correspondent in traditional computing where resources are under the control of the operating system. Static workflows correspond to traditional programs while dynamic workflows correspond to self-modifying ones.

The milestones in the life of a workflow are: 1) the creation of the *process description* (PD); 2) verification of the

PD; 3) the creation of a *case description* (CD); and 4) the enactment of a case.

Scripts, specialized workflow description languages, and formal methods can be used for process description [13], [14]. Several workflow description languages exist. Petri nets (PNs) and their restrictions have provided for many years the formal methods of choice for process description for business-oriented workflows; there is a vast literature on PNs and numerous algorithms and tools for PN analysis have been developed along the years.

To avoid enactment errors, we need to verify the process description and check for desirable properties such as *safety* and *liveness*. Some process description methods are more suitable for verification than others.

In an open system, it is desirable to support multiple process description methods but a single internal representation method.

We distinguish two types of workflows, *static* and *dynamic*. The process description of a static workflow is invariant in time. The process description of a dynamic workflow changes during the workflow enactment phase due to circumstances unforeseen at the process definition time. Exceptional conditions are handled by a static workflow using its exception handling mechanisms while unforeseen circumstances trigger planning and generation of a new process description. For example, an activity in a process description involves a service that has been discontinued, but there are several new services whose composition is equivalent to the missing service.

It is conceivable that multiple variations of a process description may coexist and it is useful to define the concept of *workflow inheritance* and exploit it in the implementation of a grid coordination architecture.

D. Process and Case Description

A *process description* is a formal description of the complex problem a user wishes to solve. For the process description, we use a formalism similar to the one provided by augmented transition networks (ATNs) [16]. The coordination service implements an abstract ATN machine. A *case description* provides additional information for a particular instance of the process the user wishes to perform, e.g., it provides the location of the actual data for the computation, additional constraints related to security, cost, or the quality of the solution, a soft deadline, and/or user preferences [8].

The process description used by our planning experiments is based upon the Backus Naur form (BNF) grammar presented below. The symbol S denotes the start symbol, while e stands for an empty string.

```
S::= ⟨ProcessDescription⟩
⟨ProcessDescription⟩ ::= BEGIN ⟨Activities⟩
                         END
⟨Activities⟩ ::= ⟨SequentialActivities⟩
            |⟨ConcurrentActivities⟩
            |⟨IterativeActivities⟩
            |⟨SelectiveActivities⟩
            |⟨Activity⟩
⟨SequentialActivities⟩ ::= ⟨Activities⟩
            ;⟨Activities⟩
⟨ConcurrentActivities⟩ ::= FORK ⟨Activities⟩
            ;⟨Activities⟩ JOIN
⟨IterativeActivities⟩ ::= ITERATIVE
            ⟨ConditionalActivity⟩
            ⟨SelectiveActivities⟩ ::= CHOICE
            ⟨ConditionalActivity⟩;
            ⟨ConditionalActivitySet⟩
            MERGE
⟨ConditionalActivitySet⟩ ::=
            ⟨ConditionalActivity⟩
            |⟨ConditionalActivity⟩
            ;⟨ConditionalActivitySet⟩
⟨ConditionalActivity⟩ ::=
            { COND ⟨Conditions⟩}
            { ⟨Activities⟩}
⟨Activity⟩ ::= ⟨String⟩
⟨Conditions⟩ ::=
            ( ⟨Conditions⟩ AND ⟨Conditions⟩ )
            | ( ⟨Conditions⟩ OR ⟨Conditions⟩ )
            | NOT ⟨Conditions⟩
            |⟨Condition⟩
⟨Condition⟩ ::= ⟨DataName⟩.⟨Attribute⟩
            ⟨Operator⟩⟨Value⟩
⟨DataName⟩ ::= ⟨String⟩
⟨Attribute⟩ ::= ⟨String⟩
⟨Operator⟩ ::= ⟨ | ⟩ | = | ⟨= | ⟩ =
⟨Value⟩ ::= ⟨String⟩
⟨String⟩ ::= ⟨Character⟩⟨String⟩
            |⟨Character⟩
⟨Character⟩ ::= ⟨Letter⟩|⟨Digit⟩
⟨Letter⟩ ::= a | b |...| z
            | A | B |...| Z
⟨Digit⟩ ::= 0 |1|...|9
```

E. Coordination Services

Let us now examine the question, why are coordination services needed in an intelligent grid environment and how can they fulfill their mission? First of all, some of the computational activities are long lasting and it is not uncommon to have a large simulation running for 24 h or more. An end user may be intermittently connected to the network, so there is a need for a proxy whose main function is to wait until one step of the complex computational procedure involving multiple programs is completed and launch the next step of the computation. Of course, a script will do, but during this relatively long period of time unexpected conditions may occur and the script would have to handle such conditions. On the other hand, porting a script designed for a cluster to a grid environment is a nontrivial task. The script would have to work with other grid services, e.g., with the *information* service, or directory services to locate other core services, with the *brokerage* service to select systems which are able to carry out different computational steps, with a *monitoring* service

to determine the current status of each resource, with a *persistent storage* service to store intermediary results, with an authentication service for security considerations, and so on.

While automation of the execution of a complex task in itself is feasible using a script, very often such computations require human intervention. Once a certain stage is reached, while some conditions are not met, we may have to backtrack and restart the process from a previous checkpoint using a different set of model parameters, or a different input data. For example, during the computation of the correlation coefficient indicating the resolution of the sample electron density map, we may decide to eliminate some of the original virus particle projections which introduce too much noise in the reconstruction process. It would be very difficult to automate such a decision, which requires the expertise of a highly trained individual. In such a case, the coordination service should checkpoint the entire computation, release most resources, and attempt to contact an individual capable of making a decision. If the domain expert is connected to the Internet with a palmtop computer with a small display and a wireless channel with low bandwidth, the coordination service should send low-resolution images and summary data enabling the expert to make a decision.

In summary, the coordination service acts as a proxy for the end user and interacts with core and other services on user's behalf. It hides the complexity of the grid from the end-user and allows user interfaces running on the network access devices to be very simple. The coordination service should be reliable and able to match user policies and constraints (e.g., cost, security, deadlines, quality of solution) with the corresponding grid policies and constraints.

A coordination service relies heavily on shared ontologies. It implements an abstract machine which understands a description of the complex task—we call it a *process description*—and a description of a particular instance of the task—we call it a *case description*.

IV. Other Core Services

An intelligent grid relies heavily on a set of core services to provide seamless access to grid resources. In a large-scale system most core services, including the information services, should be replicated to guarantee availability and a reasonable response time. We now discuss briefly some of these services.

The *information services* support a critical function: it allows individual agents to discover the resources available in the grid. Core and end-user services register with the information service and become known to the Grid community. The information provided by these services may be obsolete.

A *brokerage service* provides accurate information about the status of a few classes of resources to a narrowly targeted audience. The scope of the information services is broader than the scope of a brokerage service, they provide information about virtually all Grid resources, but the information may be obsolete. The scope of a brokerage service is limited to one or more classes of resources. Also, a brokerage service develops its own "clientele" of end-user services.

Individual request for resources can be crisp, or precise, and in such instances a broker is able to identify a set of resources matching the request without any difficulty. In other instances the resource requests are fuzzy and we need a *matchmaking service* to discover the best possible match between the consumer(s) and the producer(s) of resources. Typically, a matchmaking algorithm returns a set of targets that match the request with different degrees and then the ultimate decision is made either by the end-user, or by its proxy, the coordination service.

A *planning service* is responsible for creating original process descriptions or plans and, more often, for replanning. Replanning is necessary to adapt an existing process description to new conditions.

The goal of planning in a Grid environment is to generate a process description (also called a plan) so that the execution of the process description can produce the results that meet all goal specifications.

The planning service accepts planning requests from the coordination service. Such an assignment includes the set of the initial data available to the end user, the goal of planning, and possibly other useful information. The goal is often expressed in terms of results of the computations expected by the end user. Once the process description is created, the planning service sends it to the coordination service and, if so directed, to an archiving or persistent storage service. Traditional AI planning approaches can be applied to planning in a Grid system [5], [17].

In addition to *ab initio* generation of valid process descriptions, the planning service is involved in replanning. *Replanning* is triggered by the coordination service, whenever the state of the environment is such that the execution of the current case description based upon a valid process description cannot continue. When replanning is required, the coordination service sends to the planning service all available data, including the initial set of data and the data modified, or created during the execution of the case description.

Conceptually, replanning has the same attributes as planning. During replanning, the planning service avoids activities that prevented the successful execution of the original plan. To gather runtime information the planning uses a broker to locate a group of application containers able to host the activity that failed in the original plan. Replanning does not guarantee the successful execution of the task because the state of resources needed by various activities changes rapidly.

An *event* is caused by the change of the state of a system. The system where the change of state occurs is called the *producer* of the event and all systems which react to this event are *consumers* of the event. An *event service* connects a producer of events with the consumer(s) of the event. Most reactive systems are based upon the *event-action model* with an action associated with every type of event.

Most distributed systems such as `CORBA` or `JINI` support event services. The need for an event service is motivated by several considerations. First, the desire to support asynchronous communication between producers and the consumers of events intermittently connected to the network.

Second, in many instances there are multiple consumers of an event and it would be cumbersome for the producer to maintain state (a record of all subscribers to an event) and it would distract the producer from its own actions. Third, it is rather difficult to implement preemptive actions, yet multiple events of interest to a consumer may occur concurrently. An event service may serialize these events and allow the consumer to process them one after the other. Last, but not least, the event service may create composite events from atomic events generated by independent producers.

V. Case Study: The BondGrid

Fig. 1 summarizes the architecture of the system we are currently building. In the following sections we describe the BondGrid agents, the process description and the case description, the ontologies used in BondGrid, and the coordination service. The other core services are now being implemented.

A. Bondgrid Agents

Grid services are provided by `BondGrid` agents based on `JADE` [20] and `Protégé` [7], [21], two free software packages distributed by Telecom Italy and the Stanford Medical Institute, respectively.

`JADE`, which stands for "Java Agent DEvelopment Framework," is a Foundation for Intelligent Physical Agents (FIPA)-compliant agent system fully implemented in Java and using FIPA ACL as an agent communication language. The `JADE` agent platform can be distributed across machines which may not run under the same OS. Each agent has a unique identifier obtained by the concatenation $(+)$ of several strings:

$$\texttt{AID} \longleftarrow \texttt{agentname} + @ + \texttt{IPaddress/domainname} + \texttt{portnumber} + \texttt{/JADE}$$

`Protégé` is an open-source, Java-based tool that provides an extensible architecture for the creation of customized knowledge-based applications. `Protégé` uses *classes* to define the structure of entities. Each class consists of a number of *slots* that describe the attributes of an entity. A class may have one or multiple *instances*. `Protégé` can support a complex structure: a class may inherit from other classes; a slot may reference other instances. `BondGrid` uses a multiplane state machine agent model similar to the Bond agent system [2]. Each plane represents an individual running thread and consists of a finite-state machine. Each state of a finite-state machine is associated with a strategy that defines a behavior. The agent structure is described using a Python-based agent description language called *blueprint*. A `BondGrid` agent is able to recognize a blueprint, to create planes and finite-state machines accordingly, and to control the execution of different planes automatically. For example, the blueprint for a coordination service is

```
openKnowledgeBase("kb/BondGrid.pprj", "CS")
addPlane("ServiceManager")
s = bondgrid.cs.ServiceManagerStrategy(agent)
addState(s, "ServiceManager");
addPlane("Message Handler")
s = bondgrid.cs.MessageHandlerStrategy(agent)
addState(s, "MessageHandler");
addPlane("Coordination Engine")
s = bondgrid.cs.CoordinationEngineStrategy(agent)
addState(s, "Coordination Engine");
```

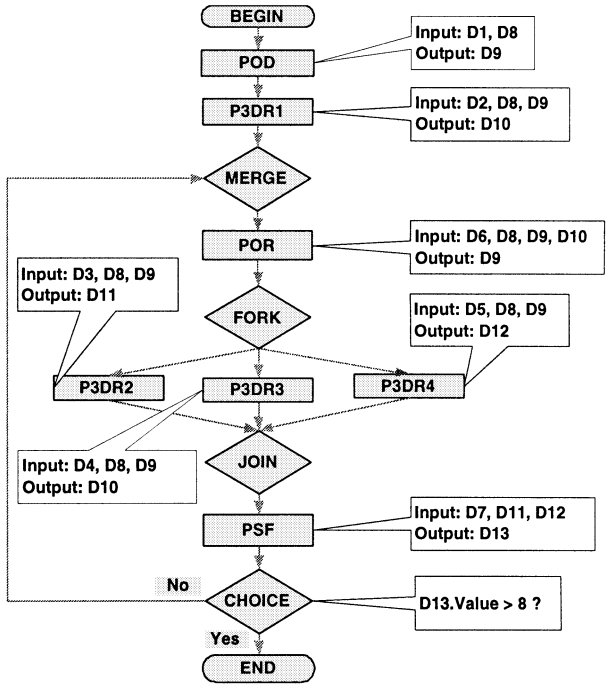


Fig. 2. A process description for the 3-D structure determination. $D1, D2, \ldots D13$ are the symbolic names of the input and output data files for the programs carrying out different end-user activities.

The knowledge bases are shared by multiple planes of an agent. The BondGrid agents provide a standard API to support concurrent access to the knowledge bases. Messages are constructed using the Agent Communication Language (ACL). A message has several fields: sender, receivers, keyword, and message content. The keyword enables the receiver of a message to understand the intention of the sender. A message may have one or more user-defined parameters. Two agents use XML formatted messages to exchange an instance of a class or the structure of a class.

B. Process Description and Case Description

A process description defines the data dependencies among the activities of a complex task and consists of *end-user activities* and *flow control activities*. The execution of an end-user activity corresponds to the execution of an end-user service thus of an application program. The specification of an end-user activity may include the symbolic names of the input and/or output data sets of the corresponding program. Fig. 2 shows a sample process description for the 3-D atomic structure determination of macromolecules based upon electron microscopy.

Flow control activities do not have associated end-user services. They are used to control the execution of end-user

activities. We define six flow control activities: `Begin`, `End`, `Choice`, `Fork`, `Join`, and `Merge`. Every process description should start with a `Begin` activity and conclude with an `End` activity. The `Begin` activity and the `End` activity should occur exactly once in a process description.

The *direct precedence* relation reflects the causality among activities. If activity $\mathcal{B}$ can only be executed directly after the completion of activity $\mathcal{A}$, we say that $\mathcal{A}$ is a *direct predecessor* activity of $\mathcal{B}$ and that $\mathcal{B}$ is a *direct successor* activity of $\mathcal{A}$. An activity may have a *direct predecessor set* of activities and a *direct successor set* of activities. We use the term "direct" rather than "immediate" to emphasize the fact that there may be a gap in time from the instance an activity terminates and the instance its direct successor activity is triggered. For the sake of brevity, we drop the word "direct" and refer to predecessor activity set, or predecessor activity and successor activity set, or successor activity.

A `Choice` flow control activity has one predecessor activity and multiple successor activities. It can be executed only after its predecessor activity has been executed. Following the execution of a `Choice` activity, only *one* of its successor activities may be executed.

A `Fork` flow control activity has one predecessor activity and multiple successor activities. The difference between `Fork` and `Choice` is that after the execution of a `Fork` activity, all the activities in its successor set are triggered.

A `Merge` flow control activity is paired with a `Choice` activity. It has a predecessor set consisting of two or more activities and only one successor activity. A `Merge` activity is triggered after the completion of *any* activity in its predecessor set.

A `Join` flow control activity is paired with a `Fork` activity. Like a `Merge` activity, a `Join` activity has multiple predecessor activities and only one successor activity. The difference is that a `Join` activity can be triggered only after all of its predecessor activities are completed.

A *case description* associates symbolic data names referred to by the corresponding process description with real data. If the data refers to a file, one or more URLs are specified. Multiple URLs refer to multiple copies of the same file. Various constraints (e.g., deadlines, cost, exclusion of some resources, special resource requirements) are often provided by case descriptions.

Process and case descriptions are part of the system-wide ontologies, as seen in Fig. 3. Their instances can be stored in knowledge bases and exchanged in XML format.

C. Ontologies

Ontologies are the cornerstone of interoperability. They represent the "glue" that allows different applications to use various grid resources. The term *ontology* means the study of what exists or what can be known. An ontology is a catalog of and reveals the relationships among a set of concepts assumed to exist in a well defined area. Creating ontologies in the context of Grid computing represents a monumental task. Fig. 3 shows the logic view of the main ontologies used in BondGrid and their relations. A nonexhaustive list of classes in this ontology includes: `task`, `process description`, `case description`, `activity`, `data`, `service`, `resource`, `hardware`, and `software`. A `task` class is related to the `process description` and the `case description` classes. A `process description` contains a set of `activities`. A `service` class is associated with a `resource` class. In turn a `resource` class is associated with `hardware` and `software` classes. The `data` class is connected to the in `activity`, `service`, and `case description` classes.

Ontologies are stored in a knowledge base which can be accessed by applications. Applications also need to exchange ontologies with each other.

BondGrid uses an XML format to describe instances of classes for exchange. Below is an informal description of the XML format. Each instance has a unique ID. The `slot-value` can be a value, or an instance.

```
⟨?xml version = "1.0" encoding = "UTF-8"?⟩
⟨project project-name = "projectname"⟩
 ⟨instances⟩
  ⟨instance class-name = "classname" ID = "id"⟩
   ⟨slot slot-name = "slotname"⟩
    ⟨value⟩
     slot-value
    ⟨/value⟩
    ⟨value⟩
     ......
    ⟨/value⟩
   ⟨/slot⟩
   ⟨slot slot-name = "slotname"⟩
    ......
   ⟨/slot⟩
  ⟨/instance⟩
  ⟨instance class-name = "classname" ID = "id"⟩
   ......
  ⟨/instance⟩
 ⟨/instances⟩
⟨/project⟩
```

BondGrid also uses an XML format to describe various classes. The `slot-type` can be string, boolean, float, integer, or an instance. The cardinality-value can be an nonnegative integer or a wild card *.

```
⟨?xml version = "1.0" encoding = "UTF-8"?⟩
⟨project project-name = "projectname"⟩
 ⟨classes⟩
  ⟨class class-name = "classname"⟩
   ⟨slot slot-name = "slotname"⟩
    ⟨type⟩
     slot-type*
    ⟨/type⟩
    ⟨cardinality⟩
     cardinality-value
    ⟨/cardinality⟩
```

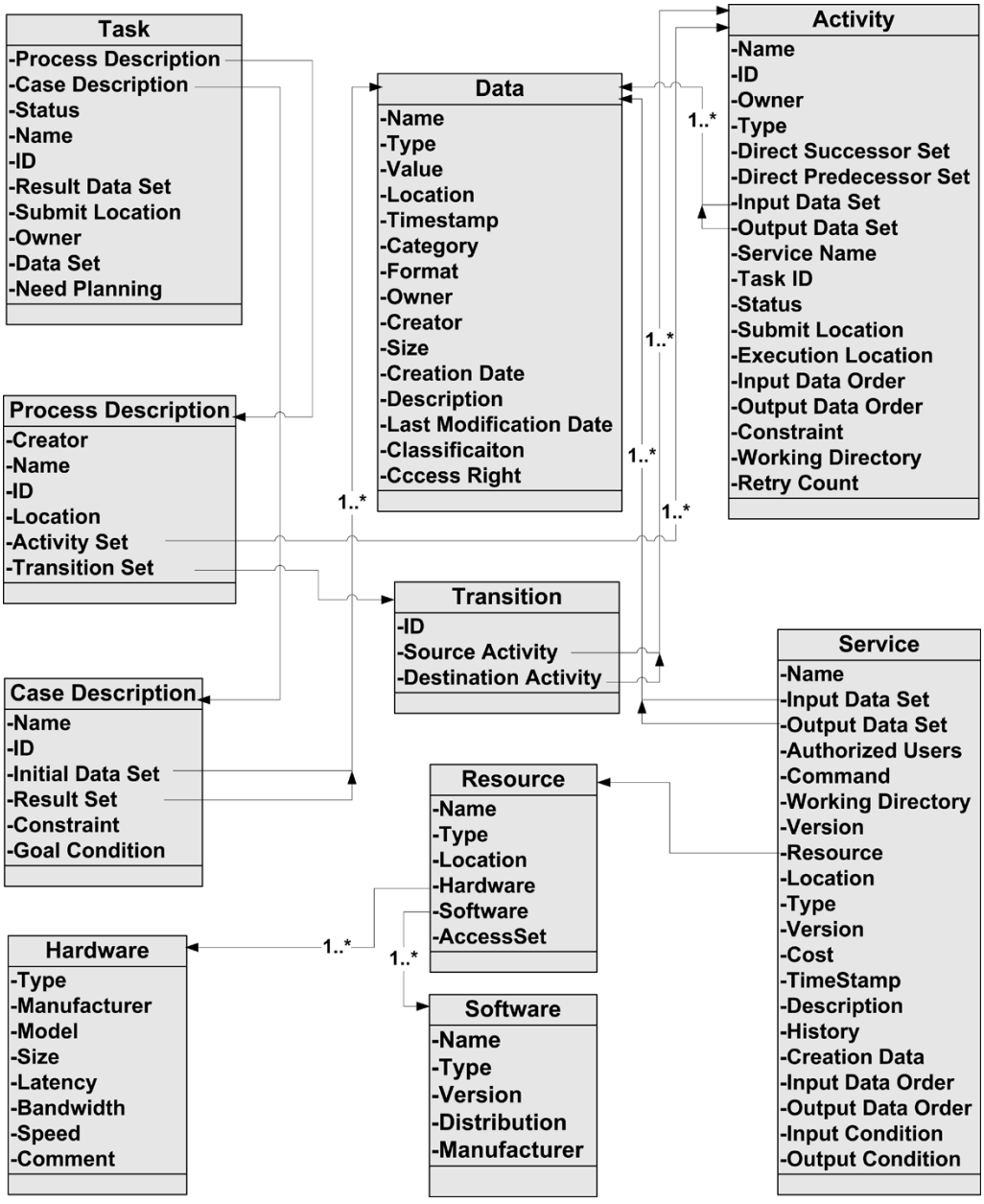


Fig. 3. Logic view of the main ontology in BondGrid.

```
    ⟨/slot⟩
    ⟨slot slot-name = "slotname"⟩
      ......
    ⟨/slot⟩
  ⟨/class⟩
  ⟨class class-name = "classname"⟩
    ......
  ⟨/class⟩
⟨/classes⟩
⟨/project⟩
```

D. The Coordination Service

The coordination service consists of a message handler, a coordination engine, and a service manager. The message handler is responsible for interagent communication. The coordination engine manages the execution of tasks submitted to the coordination service. The service manager provides a GUI for monitoring the execution of tasks and the interactions between coordination service and other services. These three components run concurrently on different planes of the agent and share the same knowledge base. Any modifications performed on the knowledge base by one component affects the other components.

As pointed out earlier, a task consists of a process description and a case description. The execution of a task is associated with a common data space shared by all activities. Each symbolic name in the process description corresponds to an entry in this data space. Initially, the case description may provide some binding of symbolic names to existing data files. As the execution of the task progresses, more symbolic names are bound to the data produced as the result of

end-user services. After the successful completion of a task, the data files containing the results are disposed of as specified by the case description (e.g., sent to a persistent storage service).

The states of a task are SUBMITTED, WAITING, RUNNING, PLANNING, REPLANNING, FINISHED, and ERROR. Once a *task submission* message is received it is queued by the message handler of the coordination service. Then the message handler creates a *task instance* in the knowledge base. The initial state of the newly created task is SUBMITTED.

The coordination engine keeps checking the state of all task instances in the knowledge base. When it finds a task instance in SUBMITTED state, it attempts to initiate its execution. One of the slots of the task class indicates if the task needs planning (the slot is set to PlanningNeeded). If the task has already been sent to the planning engine and awaits the creation of a process description the slot is set to Waiting. If the process description has been created the slot is set to PlanningComplete.

If the task needs planning, the coordination engine waits until the new process description is ready, then it updates the task instance accordingly and sets its state to RUNNING. When the execution of the task cannot continue (e.g., due to resource unavailability) the coordination engine may send the task to a planning service for replanning. In such a case, the state of the task is set to REPLANNING. After the successful completion of a task its state is FINISHED, while in case of an error it is set to ERROR.

The coordination engine takes different actions according to the type of each activity. The handling of flow control activities depends on their semantics. For an end-user activity, the coordination service collects the necessary input data and performs data staging of each data set, bringing it to the site of the corresponding end-user service. Upon completion of an activity, the coordination service triggers a data staging phase, collects partial results, and updates the data space.

The activity class has a slot describing the state of an activity, INACTIVE, ACTIVE, DISPATCHED, NOSERVICE, FINISHED, or ERROR. Initially, an activity is in the INACTIVE state. The coordination engine sets the state of its begin activity as ACTIVE when the state of a task transitions from WAITING to RUNNING. When the coordination engine finds an ACTIVE activity it checks the type slot of the activity class. In case of a flow control activity, the coordination engine sets: 1) the state of one or more successor activities to ACTIVE and 2) the state of the current activity to FINISHED. In case of an end-user activity, the coordination engine attempts to find an end-user service for this activity subject to a time and/or a retry count limit. If the coordination engine finds an end-user service, the state of this activity becomes DISPATCHED. Otherwise, the state becomes NOSERVICE. When the end-user service signals the successful completion of an activity the coordination engine sets: 1) the state of the corresponding activity to FINISHED and 2) the state of the successor activity to ACTIVE; otherwise, the state is set as ERROR.

The interactions between a user and the coordination service can be initiated by the user when submitting a task or requesting task status information, or by the coordination service when reporting an error condition or the successful completion of the task.

A request for coordination is triggered by the submission of a task initiated by a user. When receiving such a message the coordination engine first checks the correctness of the process and task description. Next, the task activation process presented earlier is triggered. The user interface then subscribes to the relevant events produced by the coordination service.

A user may send a query message to the coordination service requesting task state information. The message handler parses the request and fetches from its knowledge base the relevant slots of the task instance.

Upon completion of the task, or in case of an error condition, the coordination service posts the corresponding events for the user interface.

A coordination service acts as a proxy for one or more users and interacts on behalf of the user with other core services such as the brokerage service, the matchmaking service, the planning service, and the information service. If a task submitted by the user does not have a valid process description, the coordination service forwards this task to a planning service. During the execution of a task, when the coordination service needs to locate an end-user service for an activity, it interacts with the brokerage and matchmaking services. A brokerage service has up-to-date information regarding end-user services and their status. A matchmaking service is able to determine a set of optimal or suboptimal matchings between the characteristics of an activity and each service provider.

The event service supports asynchronous communication. For example, a user submits a task using a PDA connected via a wireless network to the Internet and subscribes for several events. Then the user is disconnected from the network. After the completion of the task the coordination service posts a termination event. When the end-user is reconnected to the network and inquires about the status of the task, the termination event is delivered to the user interface.

Besides core services, a coordination service interacts with application containers. When a coordination service attempts to locate the optimal end-user service for an activity, the status and the availability of data on the grid node providing the end-user service ought to be considered in order to minimize communication costs.

E. Performance Measurements

While we cannot attempt a realistic performance evaluation study of the coordination service before all the components of the BondGrid environment are fully implemented, we have conducted some performance studies on the communications between the coordination service and other societal (core) services.

The coordination service uses messages to exchange ontologies with other components of the environment.

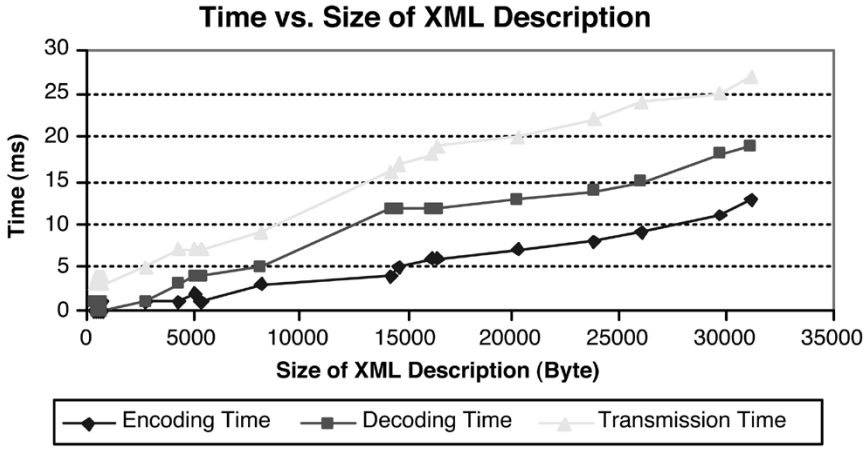


Fig. 4. The performance of encoding, transmitting, and decoding instances between the coordination service and other components of the environment.

An ontology is first converted to an XML format and then packed into a BondGrid message.

The footprint of instances varies function of the class: task instances and process description instances often consist of tens to hundreds of kilobytes; data instances are often less than one kilobyte. We experimented with ontologies of different sizes. Fig. 4 shows the relationship between the size of the ontologies and the encoding, transmission, and the decoding time. Our testing environment was provided by two systems with 1.8-GHz Pentium IV processors and 1 GB of main memory, under Linux. The two machines are connected to the same hub. The resolution of our clock is 1 ms. Fig. 4 indicates that as the size of the ontology (in XML format) increases, the time needed to encode, transmit, and decode increases as well. Encoding time, transmission time and decoding time are generally less than 30 ms. We observed considerably large transmission times (seconds), when the network load was high.

We also studied the message exchange rate between a coordination service and an agent. The coordination service is running on a system with a 1.8-GHz Pentium IV processor and 1 GB of main memory under Windows. The agent runs on a system with a 900-MHz Pentium III processor and 256 MB of memory under Windows. The agent keeps submitting tasks to the coordination service, which is able to process about 100 submissions/s while the incoming rate is about 6000 submissions/s.

VI. Applications to Computational Biology

The 3-D atomic structure determination of macromolecules based upon electron microscopy [10] is an important application of biology computation. The procedure for structure determination consists of the following steps:

1) Extract individual particle projections from micrographs and identify the center of each projection.
2) Determine the orientation of each projection.
3) Carry out the 3-D reconstruction of the electron density of the macromolecule.
4) Dock an atomic model into the 3-D density map.

Steps 2 and 3 are executed iteratively until the 3-D electron density map cannot be further improved at a given resolution. Then the resolution of the 3-D reconstruction is increased gradually from, say, 40 Å to the highest resolution supported by the quality of the data, say, 6–7 Å. The number of iterations could reach a few hundreds. One iteration for a medium size virus may take several days. Typically, it takes months to obtain a high resolution electron density map. Once we have a detailed electron density map of the virus structure, we can proceed to atomic level modeling, namely, placing groups of atoms, on secondary, tertiary, or quaternary structures.

The process description for the 3-D structure determination task is shown in Fig. 2. First, we determine the initial orientation of individual views using an *ab initio* orientation determination program called `POD`. Then, we construct an initial 3-D density model using our parallel 3-D reconstruction program called `P3DR`. Next, we execute an iterative computation consisting of multiresolution orientation refinement called `POR`.

In order to determine the resolution, we consider two streams of input data, e.g., by assigning the odd-numbered virus projections to one stream and the even-numbered virus projections to the second stream. Then, we construct two models of the 3-D electron density maps and determine the final resolution by correlating the two models using a program called `PSF`. The iterative process stops whenever no further improvement of the electron density map is noticeable, or when the target resolution has been reached.

The initial data files specified in the case description are: 1) a file containing the 2-D virus projections extracted from the micrographs; 2) a file containing the parameters of the model the expected goal resolution.

Then, using the user interface, we start the computation using the process description in Fig. 2. For every experiment we use different case descriptions with different input data sets and different target resolutions. The coordination service supervises the execution of the computation and provides the results upon completion.

VII. Conclusion

From our experience, it is abundantly clear that the development of complex and scalable systems requires some form of intelligence. We cannot design general policies and strategies which do not take into account the current state of a system. But the state space of a complex system is very large and it is infeasible to create a rigid control infrastructure. The only alternative left is to base our actions on logical inference. This process requires a set of policy rules and facts about the state of the system, gathered by a monitoring agent. Similar arguments show that we need to plan if we wish to optimally use the resource-rich environment of a computational grid, subject to QoS constraints. Further optimization is only possible if various entities making decisions have also the ability to learn.

Yet it is not so clear that the current AI technologies are at the point where their application to grid environments is unproblematic. Our limited experiments discussed in Section V-E point out that there are limitations of the current agent, knowledge base, inference, and planning technologies.

In the near future, the testbed system we are now developing will allow us to perform more comprehensive measurements. At the same time, data collected from these experiments will allow us to create realistic models of large-scale system and study their scalability.

References

[1] W. M. P. van der Aalst, A. H. ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns (technical report)," Eindhoven Univ. Technology, Eindhoven, The Netherlands, 2000.

[2] L. Bölöni, K. K. Jun, K. Palacz, R. Sion, and D. C. Marinescu, "The bond agent system and applications," in *Lecture Notes on Computer Science, Agent Systems, Mobile Agents, and Applications*, D. Kotz and F. Mattern, Eds. Heidelberg, Germany: Springer-Verlag, 2000, vol. 1882, pp. 99–112.

[3] G. Cabri, L. Leonardi, and F. Zambonelli, "Reactive tuple spaces for mobile agent coordination," in *Lecture Notes in Computer Science,Mobile Agents: Second International Workshop*, K. Rothermel and F. Hohl, Eds. Heidelberg, Germany: Springer-Verlag, 1998, vol. 1477, pp. 237–248.

[4] N. Carriero and D. Gelernter, "Linda in context," in *Commun. ACM*, vol. 32, 1989, pp. 444–458.

[5] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda, "Mapping abstract complex workflows onto grid environments," *J. Grid Comput.*, vol. 1, no. 1, pp. 9–23, 2003.

[6] I. Foster and C. Kesselman, Eds., *The Grid: Blueprint for a New Computer Infrastructure*, 1st ed. San Francisco, CA: Morgan Kaufmann, 1999.

[7] W. E. Grosso, H. Eriksson, R. W. Fergerson, J. H. Gennari, S. Tu, and M. A. Musen, "Knowledge modeling at the millennium (The design and evolution of Protégé—2000)," presented at the 12th Int. Workshop Knowledge Acquisition, Modeling and Mangement (KAW'99), Banff, AB, Canada, 1999.

[8] D. C. Marinescu, *Internet-Based Workflow Management: Toward a Semantic Web*. New York: Wiley, 2002.

[9] D. C. Marinescu, G. M. Marinescu, and Y. Ji, "The complexity of scheduling and coordination on computational grids," in *Process Coordination and Ubiquitous Computing*, D. C. Marinescu and C. Lee, Eds. Boca Raton, FL: CRC, 2002, pp. 119–132.

[10] D. C. Marinescu and Y. Ji, "A computational framework for the 3-D structure determination of viruses with unknown symmetry," *J. Parallel Distrib. Comput.*, vol. 63, pp. 738–758, 2003.

[11] A. Omicini, F. Zamborelli, M. Klush, and R. Tolksdorf, *Coordination of Internet Agents: Models, Technologies and Applications*. Heidelberg, Germany: Springer-Verlag, 2001.

[12] C. Petrie, S. Goldmann, and A. Raquet, "Agent-based project management," in *Lecture Notes in Artificial Intelligence, Artificial Intelligence Today*. Heidelberg, Germany: Springer-Verlag, 1999, vol. 1600, pp. 339–362.

[13] D. Rossi, G. Cabi, and E. Denti, "Tuple-based technologies for coordination," in *Coordination of Internet Agents: Models, Technologies and Applications*, A. Omicini, F. Zamborelli, M. Klush, and R. Tolksdorf, Eds. Heidelberg, Germany: Springer-Verlag, 2001, pp. 83–109.

[14] J. G. Schneider, M. Lumpe, and O. Nierstrasz, "Agent coordination via scripting languages," in *Coordination of Internet Agents: Models, Technologies and Applications*, A. Omicini, F. Zamborelli, M. Klush, and R. Tolksdorf, Eds. Heidelberg, Germany: Springer-Verlag, 2001, pp. 153–175.

[15] L. Tobin, M. Steve, and W. Peter, "T spaces: The next wave," *IBM Syst. J.*, vol. 37, no. 3, pp. 454–474, 1998.

[16] T. Winograd, *Language as a Cognitive Process*. Reading, MA: Addision-Wesley, 1983.

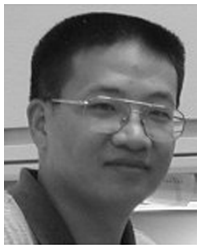
[17] H. Yu, D. C. Marinescu, A. S. Wu, and H. J. Siegel, "A genetic approach to planning in heterogeneous computing environments," presented at the 17th Int. Parallel and Distributed Proc. Symp. (IPDPS 2003), Nice, France.

[18] H. Yu, X. Bai, G. Wang, Y. Ji, and D. C. Marinescu, "Metainformation and workflow management for solving complex problems in grid environments," presented at the 18th Int. Parallel and Distributed Proc. Symp. (IPDPS 2004), Santa Fe, NM, 2004.

[19] Global Grid Forum [Online]. Available: http://www.gridforum.org/

[20] JADE Website [Online]. Available: http://sharon.cselt.it/projects/jade/

[21] Protégé Website [Online]. Available: http://protege.stanford.edu/

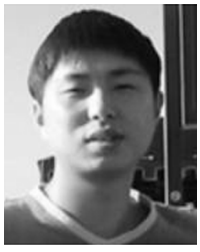


Xin Bai received the B.S. degree from Northern Jiaotong University, Beijing, China, in 1993 and the M.S. degree from the University of Central Florida, Orlando, in 2003, respectively. He is currently working toward the Ph.D. degree in the School of Computer Science at the University of Central Florida.

His research areas include Grid computing and multiagent systems.

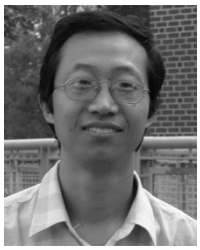
Han Yu received the B.S. degree from Shanghai Jiao Tong University, Shanghai, China, in 1996 and the M.S. degree from the University of Central Florida, Orlando, in 2002. He is currently working toward the Ph.D. degree in the School of Computer Science at the University of Central Florida (UCF).

His research areas include genetic algorithms and AI planning.



Guoqiang Wang received a B.S. degree from Southeast University, Nanjing, China, in 2001. He is currently working toward the Ph.D. degree in the School of Computer Science at the University of Central Florida, Orlando.

His research areas include *ad hoc* routing, Grid computing, and multiagent systems.



Yongchang Ji received the M.S. and Ph.D. degrees in computer science from University of Science and Technology of China, Heifei, in 1996 and 1998, respectively.

He was a Postdoctoral Researcher of computer sciences at Purdue University, West Lafayette, IN. He is currently a Postdoctoral Research Associate of computer science at University of Central Florida, Orlando. He has published more than 30 papers in professional journals and referred conference proceedings. His main research interests include high-performance computing, Grid computing, computational biology, parallel and distributed architecture, model, algorithm, and scalability.

Gabriela M. Marinescu received the B.S. degree in physics and the M.S. degree in nuclear physics from the University of Bucharest, Bucharest, Romania, and finished her D.Sc. degree studies in 1979.

She is a Senior Researcher with the School of Computer Science at University of Central Florida, Orlando. She has published more than 85 papers. She has conducted research in nuclear physics, material sciences, and computing.

Dan C. Marinescu (Senior Member, IEEE) received the M.S. degree from University of California, Berkeley, in 1969 and the Ph.D. degree from the Polytechnic Institute, Bucharest, Romania, in 1975.

He was Professor of Computer Science at Purdue University in West Lafayette, Indiana. He was a visiting faculty at IBM Research, Yorktown Heights, Intel, Institute for Information Sciences, Beijing, INRIA Paris, Deutsche Telecom. He is currently Professor of Computer Science at the University of Central Florida, Orlando, since 2001. He is the author of *Internet-Based Workflow Management* (New York: Wiley, 2002) and *Approaching Quantum Computing* (Upper Saddle River, NJ: Prentice-Hall, 2004). He coedited *Process Coordination and Ubiquitous Computing* (Boca Raton, FL: CRC, 2002). He has published more than 150 papers in professional journals and referred conference proceedings. His main research interests are scientific computing, performance evaluation, Petri nets, process coordination, and Grid computing.

Ladislau Bölöni (Member, IEEE) received the Diploma Engineer degree in computer engineering with honors from the Technical University of Cluj-Napoca, Cluj-Napoca, Romania, in 1993 and the M.S. and Ph.D. degrees from the Computer Sciences Department, Purdue University, West Lafayette, IN, in 1999 and 2000, respectively.

He is an Assistant Professor in the Computer Engineering department of University of Central Florida, Orlando. His research interests include autonomous agents, Grid computing, and knowledge representation.

Dr. Bölöni is a member of the Association for Computing Machinery and the Upsilon Pi Epsilon honorary society. He received a fellowship from the Hungarian Academy of Sciences for the 1994–1995 academic year.