1999

Biological Metaphors in he Design of Complex Software Systems

Dan C. Marinescu

Ladislau Bölöni

Report Number:
99-018

BIOLOGICAL METAPHORS IN THE DESIGN OF COMPLEX SOFTWARE SYSTEMS

Dan C. Marinescu
Ladislau Boloni

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD #99-018
May 1999

Biological Metaphors in the Design of Complex Software Systems

Dan C. Marinescu and Ladislau Bölöni

Computer Sciences Department
Purdue University
West Lafayette, IN 47907, USA
(dcm,boloni@cs.purdue.edu)

Abstract

In this paper we discuss metaphores inspired by structural biology, genetics, neurology and immunology for building complex software systems. Structural biology offers hints for software composition. Genetics provides ideas to construct software modules from descriptions. A network of software agents could emulate the nervous system, coordinate various activities and mediate amongst interacting entities. Immunology inspires the design of secure systems. As a case study we present Bond, a distributed-object system providing agent support for network centric computing.

Contents

Preprint submitted to Elsevier Preprint

1 Overview

A number of biological analogies have found their way into computer science. Neural networks provide an alternative to von Neumann architecture [1–3], genetic algorithms [4] are used to solve optimization problems, mutation analysis was proposed for software engineering. The question we are concerned with is if these analogies can be used for building complex systems out of components, [6], and emulate genetic mechanisms and the immune system.

In this paper we present a design philosophy for an infrastructure for network computing based upon software agents, within the larger context of building complex systems out of ready-made components and discuss biological metaphors that inspired the design of the system. Inherently, a complex computing system is heterogeneous and accommodating the heterogeneity of the hardware and the diversity of the software is a major concern of such a design. Though we accept the diversity of biological and social systems we have a low threshold for tolerating the impact the diversity of computer systems has upon us. Nature uses composition to build very complex forms of life and as we understand the principles and mechanisms that form the foundation of life we should try to emulate them to build more dependable and easy to use computing systems. The cornerstones of the architecture we propose are (metao)bjects, describing network objects, and software agents that perform operations on network objects. Examples of network objects are programs, data, hardware components including hosts and communication links, services, and so on.

The Bond project was triggered by a collaboration with structural biologists who provided the problems, the motivation to design a Virtual Structural Biology Laboratory, and simulated our desire to learn some basic facts about the structure of biological macromolecules. The individual algorithms and programs needed for data acquisition, data analysis and model building for x-ray crystallography and electron microscopy are discussed elsewhere [28–31]. Here we only note that processing of structural biology data involves large groups and facilities scattered around the world, and complex programs that are modified frequently. Most computations are data intensive, they require the use of parallel and distributed systems. Thus the need for an infrastructure for a Virtual Laboratory.

This paper is organized as follows. Section 2 addresses generic concerns related to the design and construction of complex systems and provides an overview of the biological metaphors. Section 3 introduces Bond. First we discuss structural biology metaphors applied to the design of a distributed-object system, then we discuss the security model. In Section 3.3 we discuss mobile agents and in Section 3.4 we introduce networks of cooperating agents.

A number of new initiatives and ideas for high performance distributed computing have emerged in the last few years. Object-oriented design and programming languages like Java open up intriguing new perspectives for the development of complex software systems. Java supports code mobility. This brings us to another significant development, computing grids [5]. Informally, a computing grid is a collection of autonomous computing platforms with different architectures, interconnected by a high-speed communication network. Computing grids are ideal for applications that have one or more of the following characteristics:

- are naturally distributed, data collection points and programs for processing the data are scattered over a wide area network,
- need a variety of services distributed over the network,
- have occasional or sustained needs for large amounts of computing resources e.g. CPU cycles, large memory, vast amounts of disk space,
- benefit from heterogeneous computing environments consisting of platforms with different architectures,
- require a collaborative effort from users scattered over a large geographic area, [11].

The software systems necessary to support computing on a grid are inherently more complex than existing ones. Yet, there is no rigorous and universally accepted definition of a complex system. Some elements that define the complexity of a software system are static others are dynamic. Static aspects, are the number of components of the system, the programming paradigms, the genealogy of the components. Dynamic aspects are concurrency, one versus multiple treads of control, locality, the components may be co-located or may be distributed and need to communicate via a non-reliable channel, and last but not least, the presence or absence of timing constrains.

Clearly, the size of the system, measured in terms of the number of components is an indication of its complexity. A system using many libraries and consisting of a very large number of modules is more complex than one with few components. Our intuition indicates that software systems written in an imperative programming language are more brittle than those written in declarative languages as the interactions among components are subject to strict conventions. For example the layout of an array of complex numbers in a module calling a library function to perform a 3D Fourier transformation must correspond exactly to the specification of the function. But virtually all existing scientific and engineering software is written in imperative languages and we have to find ways to accommodate its brittleness. Many complex software systems have a mixed genealogy. Some of the components may be *legacy programs*

using programming languages that do not enforce typing and do not check array boundaries. Other components may be written in strongly typed, modern languages supporting introspection and reflections.

Concurrency increases the complexity of a software system because multiple threads of control interact with one another often in a non-deterministic manner. But it is difficult to quantify this intuition based upon the interplay between the number of threads of control and the interaction patterns amongst them. For example the behavior of a system with only two threads of control interacting in some arbitrary fashion is more difficult to understand than that of a system with $n > 2$ threads of control interacting on a predefined pattern after a barrier synchronization. It is equally difficult to quantify the difference between *shared-memory* and *message passing* concurrent systems. Intuitively, we know that shared memory systems are more complex than message passing systems because it is more difficult to determine the history of a shared variable. It is impossible for the consumer of a shared variable to know if the variable has been updated or not by the producer, without maintaining the history of that variable. Distributed computing, spreading the threads of control on the nodes of a system adds a new dimension to system complexity. Communication is unreliable, data on communication channels can be affected by errors, duplicated, or lost. Distributed programs are more difficult to construct, debug and maintain. Finally, as sensor technology matures, the relative importance of systems with time-constrains increases. Embedded as well as mixed-mode systems including some components with real-time constrains will be pervasive. This adds an additional dimension to system complexity. In summary, concurrency, distributed computing, and timing constrains are dynamic aspects of system complexity that define new challenges for complex system design and can be mapped to biological metaphors.

The metaphors we propose for the design of complex system are inspired from structural biology, [22], genetics, neurology and immunology. Structural biology offers important hints for software composition, genetics provides useful information on how to actually construct software modules from descriptions, how to exploit the genetic economy principle, namely to build systems out of similar or identical components. The study of the neural systems may give us some hints on how to mix legacy components with more intelligent ones, capable to control various aspects of the system. Immunology can help the design of secure systems.

Nature uses composition to build extremely complex structures. There are 20 aminoacids, the basic building blocks of life. The aminoacids sequence of a protein's peptide chain is called a *primary structure.* Different regions of the structure form local regular *secondary structure* such as alpha helices and beta strands. The *tertiary structure* is formed by packing such structural elements onto globular units called *domains.* The final protein may contain

several polypeptide chains arranged in a *quaternary structure*. By formation of such tertiary and quaternary structures aminoacids far apart in the sequence are brought together in three dimensions to form a functional region, an *active site* [7]. The three dimensional structure of a protein determines its function, the disposition in space and the type of the atoms in a region of the protein provide a *lock* that can be recognized by other proteins that may bind to it, provided that they have the proper *key*. Living organisms *mutate*, the atomic structure of their cells changes and a *selection* mechanisms ensures the survival of those able to perform best their function.

Structurally, a complex system consists of components that must interact with one another. Our first observation is that an object-oriented system has distinct advantages compared to other approaches, it exposes to other components only the *active site*, the methods and the state variables the other components need to be aware of. In a complex system the more concise is the state of each component, and the better defined are the *channels* used by components to interact with one another the simpler is the description of the system, and the more likely it is that the system will have a predictable behavior.

Biological systems are composed of structures of increasing complexity, yet the 20 aminoacids are the building blocks of all biomolecules. A direct analogy of software and biological systems is difficult to justify by identifying a small number of primitive software components. But instead of looking for a small number of primitive components we should probably concentrate on design patterns. A number of design patterns for object-oriented systems have emerged, e.g. the proxy, the factory method, the decorator, the mediator, the strategy, and so on, [10].

Structurally, the list of the components of a system is equivalent to the primary structure of a protein, it gives some idea of the system complexity but cannot possibly describe the function of the system. Multithreading provides more insight into the dynamics of the system and it is analogous to the secondary structure of a protein. Different communication patterns e.g. global synchronization, pair-wise communication, etc. correspond to types of secondary structures e.g. alpha helices, and beta strands. Object distribution corresponds to tertiary structure, and timing constrains to the quaternary structure. We may infer from these analogies that a good strategy is to treat each aspect of a complex system design, concurrency, distribution, timing constrains separately and then combine them together into a unitary system by superposition. Ideally, the dynamic aspects of complex system design should be hidden by the middleware layer of the system.

A second observation reflects our belief that software agents are a critical component of any complex system. The agents represent the analog of the nervous system, they perform various coordination and control functions. An

useful step would be to define attraction and repulsion forces among components and couple the components based upon laws similar to molecular dynamics laws. Some components of the system need to exhibit an *autonomous behavior* very much like the behavior of a complex organism. For example a complex application may include a *scheduler* responsible to coordinate individual activities carried out by the other components of the application, on a system consisting of platforms with different architectures interconnected by a wide-area network. The scheduler must be provided with a set of rules to decide how to map the components to the nodes of the system, based upon: (a) information regarding the current load of a node, (b) the ability to bind a component to a node, (c) some optimization criteria for mapping, e.g. the need to minimize the network load for data staging from a producer site to a consumer site.

The critical aspect of a complex system design is the interoperability of its components thus we need to turn our attention to software composition. The idea of building a program out of ready made components has been around since the dawn of the computing age, backworldsmen have practiced it very successfully. Most scientific programs we are familiar with, use mathematical libraries, parallel programs use communication libraries, graphics programs rely on graphics libraries, and so on.

Modern programming languages like Java take the composition process one step further. A software component, be it a package, or a function, carries with itself a number of properties that can be queried and/or set to specific values to customize the component according to the needs of an application which wishes to embed the component. The mechanism supporting these functions is called introspection. Properties can even be queried at execution time. *Reflection* mechanisms allow us to determine run time conditions, for example the source of an event generated during the computation, [8]. The reader may recognize the reference to the Java Beans but other *component architectures* exists, Active X based on Microsoft's COM and LiveConnect from Netscape to name a few.

Can these ideas be extended to other types of computational objects besides software components, for example, to data, services, and hardware components? What can be achieved by creating *(meta)objects* describing *network objects* like programs, data or hardware?

The set of properties embedded into a (meta)object provide a "lock and key" mechanism similar with the one used by proteins to recognize one another. In a *workspace* populated with objects, one can envision mechanisms to link network objects together to form a *metaprogram*, a new object, capable of carrying out a well defined computational task. Example: to link a data object to a software object we need to search the workspace for an object describing

a crystallographic FFT program able to compute the 3D Fourier transform of a symmetric object with a known symmetry, given a lattice of real numbers describing the "unit cell", the building block of the object. At each step, the selection process succeeds if the target object possesses a set of "keys", corresponding to the desired properties needed for composition. This is a deliberate example, anyone familiar with FFTs recognizes that creating the objects describing the elements discussed above is a non trivial task.

We expect an object to contain *genetic information*, i.e. a description of all relevant properties of a network object . This information must be in a form suitable for machine processing. For example an object associated with a software component should include a description of the functions and interfaces of that component using a descriptive language like IDL, Interface Description Language. IDL reveals only the interfaces of an object and does not specify how these properties are to be *folded* into an actual implementation. The genetic information associated with a data object should reveal its ancestry, the characteristics of the sensor that has generated the data, or provide links to the program that has produced the data and to its input data objects. The genetic information would allow an autonomous agent to generate an actual implementation of the code in case of a software component, or a human contemplating the results of a sequence of computations to trace back decisions made at some point in the past.

Does the effort to build (meta)objects seem daunting? We should expect it according to the biological analogy discussed above. The task of abstracting the properties of network objects is monumental, one can only succeed if the network resources, software, data, and hardware, are classified into *disjoint classes* each with well-defined properties and *inheritance* mechanisms. We also need to modify the set of properties of an object. For example when an agent discovers that a processor executes incorrectly a sequence of instructions this new property should be added to the object describing that particular class of processors. A fundamental principle is that acquired traits take precedence over inherited ones. Both hardware and software are created as a result of an *evolutionary* process. Occasionally, new programs, or microprocessors, are created from scratch, but often, new versions are upgrades of existing objects, that inherit many characteristics of the older versions. Inheritance has the potential to simplify the task of building (meta)objects and agents capable to manipulate them intelligently. Care must be taken to expose in the object only *stable properties* of the network resources. The amount of main memory installed on a system is a stable property and though it may change, such changes are likely to be infrequent, while the load placed upon the system varies rapidly, it is a *transient property* and should not be exposed.

Once we recognize that creation of (meta)objects is a difficult task we have to ask ourselves if objects and the network resources need to be tightly or loosely

coupled with each other. The tightly coupled approach would require that the network object and the (meta)object be kept together to ensure consistency. There are fundamental flaws with the tightly coupled argument. First, this "ab-initio" approach would require re-creation of legacy components, software, hardware and data, to fit our scheme. Second, information would be unnecessarily duplicated and confidential information compromised.

By virtue of the arguments discussed above a (meta)object should only have a *soft link* to the network object it describes. Different properties of the network object may be accessible on a need to know basis, e.g. the source code may be available only to those who have the need for it. The (meta)object associated with a program may contain attributes describing the function of the program, its input and output. It should also provide references or pointers to the: source code, the executables for different platforms, the human readable documentation of the program, the implementation notes, an error log, and so on. Following the principle of genetic economy, components of the object would be shared among all the users of the network object. The price to pay for the distributed-object approach is that inconsistency between the objects and the network objects are occasionally unavoidable. We argue that in a network rich environment catastrophic consequences of such inconsistencies are unlikely to occur. Moreover, whenever possible, discovery agents should update the objects.

In summary, we propose to create (meta)objects describing properties of network objects. These properties should reveal how objects can be composed together using a lock and key mechanism and support a selection mechanism to eliminate components that do not perform their functions well. Linking objects together can only be done based upon a universally accepted *taxonomy of objects* and properties, and a given context. We acknowledge the fact that a considerable amount of work in the area of knowledge sharing still remains to be done, but we believe that a system built along the principles discussed above will allow a larger segment of the population to use computers to solve complex tasks.

3 A Case Study: An Infrastructure for Network Centric Computing

Bond, [21], is a distributed-object, message-oriented system. Our original objective was to support various functions needed for a Virtual Laboratory, an environment supporting data annotation, user-level resource management, workflows and scheduling of remote activities. Thus it became clear early on that we needed to integrate software agents into our distributed-object system. Once this decision was made we concluded that our system will be a

pure message-passing system because remote method invocation is too restrictive for software agents. Rather than designing our own language we adopted KQML, [15], [16]. Important as it may be, the fact that every object "speaks" KQML, in other words understands the syntax of a KQML message and is able to parse it, does not guarantee that two objects can cooperate with each other. An object should be able to "understand" semantically a message and respond to it in a coherent manner. To solve the problem of semantic understanding of a message we introduced the concept of subprotocols. A subprotocol is a closed set of messages, an object understands semantically every single message in a subprotocol and it is able to respond with a message in the same subprotocol. This guarantees the ability of objects to cooperate. Another design objective is to support objects with a wide range of granularity from simple objects like an address, a date, or a message, to complex objects like a scheduling agent or an authentication server. A server is an active object capable to provide a well-defined set of services. Though it may be very complex, a server does not have the autonomy an agent must exhibit. The ability of an object to "understand" a message is also very different.

The brief description of communication in Bond leads us to another design principle, namely minimize the need for server access. Bond objects communicate directly with each other. Objects use an *interface discovery* subprotocol, to find out what subprotocols other objects implement. An alternative is to use an interface repository service to discover the methods supported by a remote object as CORBA does. To reduce access to a directory server, each resident supports an "awareness" mechanism, as explained in the next section.

3.1 Structural Biology Metaphors Applied to the Design of a distributed-object System

An infrastructure based upon a distributed-object system is necessary to hide the intricacies of concurrency as well as distribution of data and computations across the nodes of a wide-area network. The components of the system have to interact with one another and use a universally understood "signaling" mechanism very much like biological macromolecules do. Objects inherit their methods including the ability to understand messages from their ancestors. Design patterns resembling the protein structures, including the proxy, the decorator, and the factory are discussed. Object composition by means of probes resembles protein binding.

Bond uses KQML [16], as a meta-language for inter-object communication. KQML offers a variety of message types (performatives) that express an attitude regarding the content of the exchange. Performatives can also assist agents in finding other agents that can process their requests. A performative

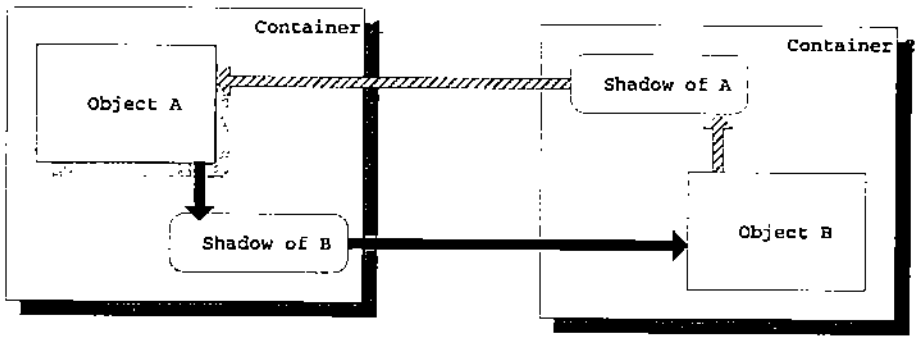

Fig. 1. Communication between remote objects using their shadows. A shadow is a proxy for a remote object. To communicate with object B in container 2, object A in container 1 creates a shadow of B using the `find` function. Once the shadow of B is created, A applies the local `say()` method to send messages to it; then the shadow delivers the message to B. To send a message to A, B follows the same same procedure, creates a shadow of A then sends messages for A to its shadow.

is expressed as an ASCII string, using a Common Lisp Polish-prefix notation. The first word in the string is the name of the performative, followed by parameters. Parameters in performatives are indexed by keywords and therefore order-independent.

The infrastructure provided by Bond supports basic object manipulation, inter-object communication, local directory and local configuration services, a distributed awareness mechanisms, probes for security and monitoring functions, and graphics user interfaces and utilities.

Shadows are proxies for remote objects, see Figure 1. Realization of a shadow provides for instantiation of remote objects. Collections of shadows form *virtual networks* of objects.

Residents are active Bond objects running at a *Bond address*. A resident is a container for a collection of objects including communicator, directory, configuration, and awareness objects.

Subprotocols are closed subsets of KQML messages. Objects inherit subprotocols. Every Bond object understands the property access subprotocols and every Bond agent understands the agent control subprotocol. The handling of the commands in these subprotocols is implemented by the methods of the corresponding objects. The messaging thread of a Bond executable delivers every incoming message to the `say()` function of the corresponding object. If the message is not understood, it is than passed to the `say()` function of the immediate ancestor in the object hierarchy. This way every Bond object inherits all the subprotocols implemented by the objects above it in the Bond object hierarchy. For example, the scheduler agent object implements the scheduling subprotocol and inherits the agent control subprotocol implemented by the `bondAgent`, and the property access subprotocol implemented

by the bondObject.

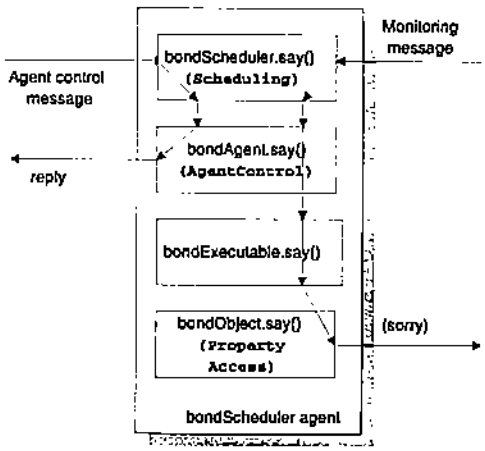

Fig. 2. Message processing by a Bond scheduler agent. The incoming messages are handled by the say() function of each object, and if not understood passed to the say() function of the parent. In parenthesis we have the subprotocol implemented by the corresponding say() function. The processing sequence is then presented for two messages: an agent control message, understood by every object which inherits from bondAgent and a monitoring message, which is not understood by this instance of the bondScheduler object.

Figure 2 shows two examples of messages delivered to a bondScheduler object, which extends a bondAgent, which in turn extends a bondExecutable, which in turn extends a bondObject. The subprotocols specified at each level are specified in paranthesis. The bondScheduler object does not understand a monitoring message (it does not inherit a monitoring subprotocol), so after being passed all the way in the hierarchy, the say() function of the bondObject class answers with sorry indicating that it does not understand it.

The transport mechanism between Bond residents is provided by a *communicator* object with four interchangeable communication engines based upon:

- UDP,
- TCP,
- Infospheres, (info.net) [9] , and
- IP Multicast protocols.

Probes are objects attached dynamically to Bond objects to augment their ability to understand new subprotocols and support new functionality.

Examples of probes are the bondMonitoringProbe which, when attached to an object, allows a remote monitor object to monitor the properties of the object, or the bondSecurityProbe which, when attached to an object, allows the object to understand encrypted messages. *Monitoring probes* implement a subscription-based monitoring model. *An autoprobe* allows loading of probes on demand.

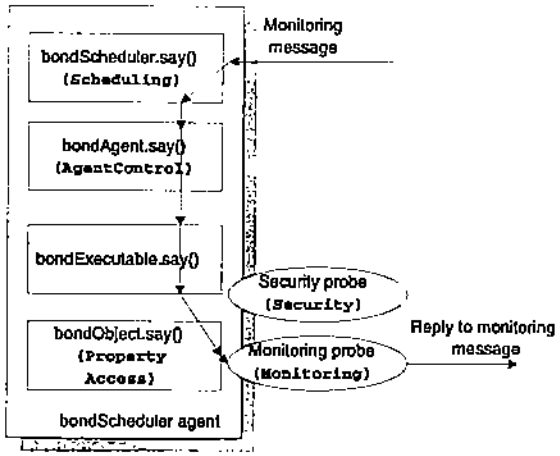

Fig. 3. The effect of extending an object with probes. In this case a `bondScheduler` is extended with a monitoring probe. The extended object understands the monitoring sub-protocol and is capable of providing a meaningful reply to a monitoring message.

In the Figure 3 we have the same scheduler agent, this time extended with two probes, a monitoring probe implementing the monitoring subprotocol, and a security probe implementing the security subprotocol. An incoming message which is part in the monitoring subprotocol is passed down in the inheritance hierarchy without being processed. At the bondObject level, after being checked that it is not part of the property access subprotocol, the object checks its dynamic properties for probes which implement the subprotocol of the message. In our case, the monitoring probe implements the required subprotocol, so the message is delivered to it, and from there the probe will take care of processing the message. If there is no probe implementing the subprotocol, the object replies `sorry`.

This construction is roughly similar in scope to the Decorator design pattern as presented in [10], allowing to dynamically extend the functionality of an object without subclassing. However the implementation is different - instead of a wrapper which captures the function call, we have a dynamically appended object which is consulted only in the case when the message does not make sense for the object itself. The difference in implementation is due to the message oriented nature of the objects: the higher flexibility and looser coupling between objects communicating by messages.

Another object-oriented structure which allows objects to acquire new functionality after "programming time" is the notion of a *mixin* [10], [12], [13]. Mixins are generally implemented as abtract classes, with reserved functions for future functionality. As such, the programmer needs at least a rough idea about the nature of the functionality with which the object may be extended. In our special case, the probes offer a larger flexibility, of course at the cost of the additional processing time to syntactically and semantically interpret the messages.

The *distributed awareness* mechanism provides information about other residents and individual objects in the network. This information is piggy-backed on regular messages exchanged among objects to reduce the overhead of supporting this mechanism. An object may be aware of objects it has never communicated with. The distributed awareness mechanism and the discovery subprotocol reflect our design decision to reduce the need for global services like directory service and interface repositories.

3.2 The Security Model and the Immune System

Security is a critical aspect of the design of any system and in this section we discuss analogies between computer security and the immune system. The immune system creates antibodies that bind to the active site of a virus and prevents it from binding to cells and infecting them. We propose to surround objects with a defense perimeter using security probes. In a message passing system the only manner to interact with an object is by sending and receiving messages. A probe is an active object screening all incoming and outgoing messages to an object and enforcing the authentication and access control models the owner of the object desires. Probes, like antibodies, are generated independent of the entities they are expected to bind to, but the analogy breaks down when we consider the attachment site. Antibodies bind to the agent causing the infection, the virus, while we propose to bind the probes to the objects that are the potential targets of undesirable actions.

Applications of network centric computing have vastly different security requirements and the trade-off between security and performance are application specific. It is unrealistic to consider one security model suitable for all applications and all environments. Additional security challenges posed by network computing are discussed below. The user population and the resource pool are large and dynamic. A user may only be aware of a small fraction of the components involved in a computation. The relations among components may be rather complex, a component may act both as a server and a client at the same time. Traditional distributed systems use RPC or TCP/IP as their primary communication mechanism. In contrast, a distributed computing environment may use two-sided communication mechanism like message passing, streaming protocols, multicast, and/or single-sided get/put operations, as well as RPC. Components may communicate through a variety of mechanisms.

The boundaries of trust are more intricate because of dynamic characteristic of components. The trust users have in components is threatened when components can be mobile between hosts and new components can be created on the fly. Boundaries of trust are more complex because an activity typically involves multiple domains with different security policies and security models.

Computation may be distributed to many more machines than any given user has control over.

Security in a network environment includes authentication and access control [23], [24]. Authentication refers to the process of identifying an individual, usually based on a username and password. Access Control is the process of granting or denying access to a network, based upon a two-step process, authentication to ensures that a user is who he/she claims to be, and access control proper which allows the user access to various resources based on the user's identity.

Some of the authentication models presented in the literature are:

- *PAP - Password Authentication Protocol.* Provides the most basic form of authentication. User's name and password are transmitted over the network and compared to a table of name-password pairs. Typically, the stored passwords are encrypted.
- *CHAP - Challenge Handshake Authentication Protocol.* The authentication agent, typically a network server, sends the client a key to encrypt the username and the password.
- *Kerberos - ticket-based authentication.* The authentication server assigns a unique key, called a ticket, to each user that logs on to the network. The ticket is then embedded in every message to identify the sender of the message.
- *Certificate-based authentication.* This model is based on public key cryptography. Each user holds a public and a private key. The user can get a certificate that proves the binding between the user and its public key from a third party. The private key is used to generate evidence that can be sent with the certificate to server side. The server uses the certificate and evidence to verify the identity of the user.

A credential is a secret code that proves the identity of an individual. Authentication models use different credentials, e.g. username/password in PAP and CHAP, user identifier/ticket in ticket-based authentication, and user certificate/private key in the certificate-based authentication.

In Bond we opted for an extensible core object that can support multiple security models and can be added dynamically to existing object. This philosophy leads to several design principles. The first is to provide a framework for security, not force an implementation. Bond leaves the decision of choosing the format of credentials, the authentication policy, the access control policy, and so on, to the system developer or the system administrator. The Bond Security Framework, BSF, is implemented as an extensible core Bond object called `BondSecurityContext` and a set of well-defined security interfaces. A second design principle is that various aspects of a complex object design,

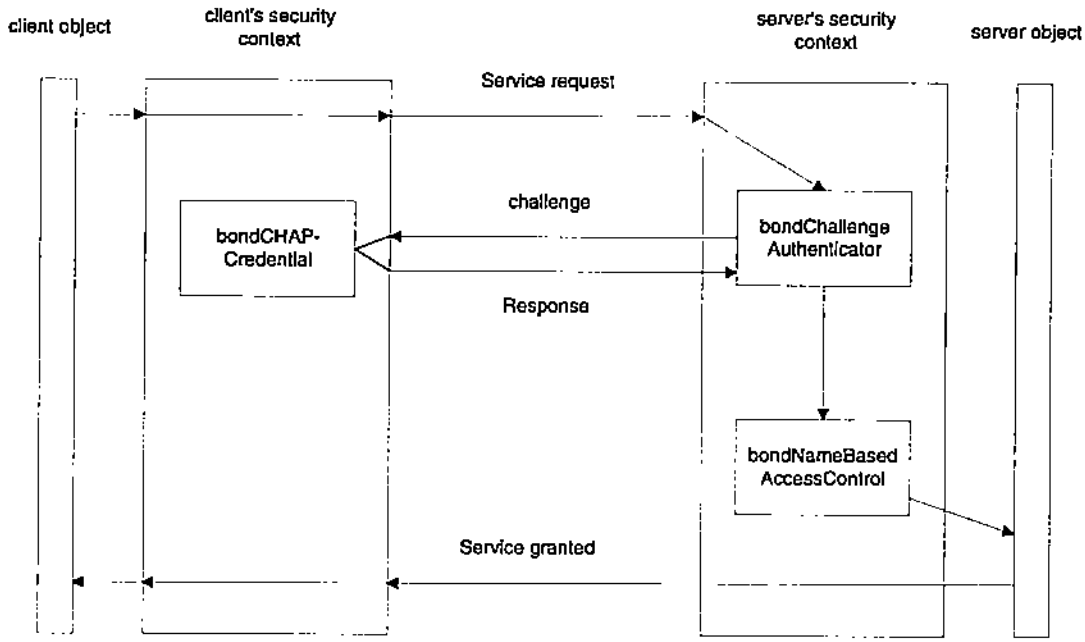

Fig. 4. The security probes attached to a client and a server provide a defense perimeter around each object. Service requests are authenticated and subject to access control tests before being forwarded to the server.

including security, should be separated from one another. In the initial design and implementation phase the creator of an object should only be concerned with functionality. Once the object is fully functional, the creator needs to investigate the security requirements and augment the object with the proper security context by including a probe called `BondSecurityContext`. Another design principle is to support multiple authentication and access control models. This goal is achieved by defining a common interface for different security functions, like credential, authentication and access control.

The security framework supports two authentication models, one based upon *username, plain password* and one based upon `CHAP`. Two access control models are supported, one based upon the *IP address (firewall)* and one based upon an *access control list*. The following example illustrates how to construct secure objects using BSF. Assume that we have one client, two servers, `serverA` and `serverB` and an authentication server that provides account management and authentication services.

Figure 4 illustrates the message exchange between a client and a server. The client sends a service request, its security context does not modify the message because it detects that a `CHAP` security model is used. On the other side, the message is intercepted by the security context of the server. The authenticator of the security context of the server sends a challenge back to the credential component of the security context of the client and expects a response derived from both the challenge and the information available in the client's credential.

Then the authenticator of the security context of the server uses the challenge and the response from the client to authenticate the client. If the service request is validated then the server object grants the service.

3.3 Mobile Agents and Genetic Information

Software agents seem to be at the center of attention in the computer science community. Yet different groups have radically different views of what software agents are, [18], [17] what applications could benefit from the agent technology. The concept of an agent was introduced by the AI community a decade ago, [14]. An AI agent exhibits an autonomous behavior and has inferential abilities and can also be used to ensure interoperability, [19]. A considerable body of work is devoted to agents able to meet the Turing test by emulating human behavior [20]. Such agents are useful for a variety of applications in science and engineering e.g. deep space explorations, robots, and so on. The object-oriented community views an agent as a mobile active object. Code mobility confers to an agent its autonomy.

In our view [21], a software agent is an abstraction for building complex systems. An agent is an *active mobile object that may or may not have inferential abilities*. Our approach to agent mobility is different too. We assemble agents dynamically from a description called a *blueprint* using components called *strategies*. Our agents consists of finite-state machines embedded into planes, that share a model, the memory of the agent. We can freeze an agent just after the completion of the strategy in one state and the transition to another occurs. To migrate an agent from one Bond resident to another we simply send the blueprint and the model to the new site. The model contains the current state of the agent. Thus the blueprint and the model provide the genetic information necessary to recreate the agent at a different location.

Several distributed-object systems provide support for agents. Infospheres (`//www.infospheres.caltech.edu/`), and Bond are academic research projects, while IBM Aglets (`www.trl.ibm.co.jp/aglets/index.html`) and Objectspace Voyager (`//www.objectspace.com`) are commercial systems.

A first distinctive feature of the Bond architecture, described in more detail in [21] is that agents are native components of the system. This guarantees that agents and objects can communicate with one another and the same communication fabric is used by the entire population of objects. Another distinctive trait of our approach is that we provide middleware, a software layer to facilitate the development of a hopefully wide range of applications of network computing. We are thus forced to pay close attentions to the software engineering aspects of agent development, in particular to software reuse. We

decided to provide a framework for assembly of agents out of components, some of them reusable. This is possible due to the agent model we overview now.

We view an agent as a finite-state machine, with a strategy associated with every state, a model of the world, and an agenda as shown in Figure 5. Upon entering a state the strategy or strategies associated with that state are activated and various actions are triggered. The model is the "memory" of the agent, it reflects the knowledge the agent has access to, as well as the state of the agent. Transitions from one state to another are triggered by internal conditions determined by the completion code of the strategy, e.g. success or failure, or by messages from other agents or objects.

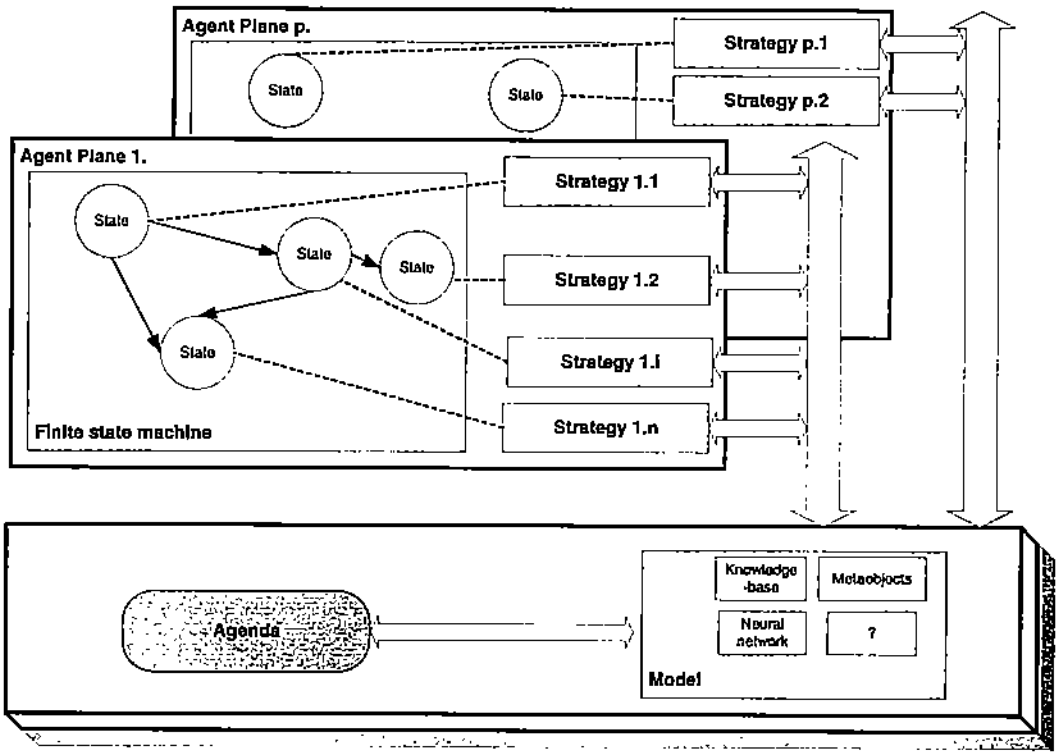

Fig. 5. The abstract model of a Bond Agent

The finite-state machine description of an agent can be provided at multiple granularity levels, a course-grain description contains a few states with complex strategies, a fine-grain description consists of a large number of states with simple strategies. The strategies are the reusable elements in our software architecture and granularity of the finite state machine of an agent should be determined to maximize the number of ready made strategies used for the agent. We have identified a number of common actions and we started building a strategy repository. Examples of actions packed into strategies are: starting up one or more agents, writing into the model of another agent, starting up a legacy application, data staging and so on. Ideally, we would like to assemble an agent without the need to program, using ready-made strategies from

repositories.

Another feature of our software agent model is the ability to assemble an agent dynamically from a "blueprint", a text file describing the states, the transitions, and the model of the agent. Every Bond-enabled site has an "agent factory" capable to create an agent from its blueprint. The blueprint can be embedded into a message, or the URL of the blueprint can be provided to the agent factory. Once an agent was created, the agent control subprotocol can be used to control it from a remote site.

In addition to favoring reusability, the software agent model we propose has other useful features. First, it allows a smooth integration of increasingly complex behavior into agents. For example, consider a scheduling agent with a mapping state and a mapping strategy. Given a task and a set of target hosts capable to execute the task, the agent will map the task to one of the hosts subject to some optimization criteria. We may start with a simple strategy, select randomly one of the target hosts. Once we are convinced that the scheduling agent works well, we may replace the mapping strategy with one based upon an inference engine with access to a database of past performance. The scheduling agent will perform a more intelligent mapping with the new strategy. Second, the model supports agent mobility. A blueprint can be modified dynamically and an additional state can be inserted before a transition takes place. For example a "suspend" new state can be added and the "suspend" strategy be concatenated with the strategy associated with any state. Upon entering the "suspend" state the agent can be migrated elsewhere. All we need to do is send the blueprint and the model to the new site and make sure that the new site has access to the strategies associated with the states the agent may traverse in the future. The dynamic alteration of the finite state machine of an agent can be used to create a "snapshot" of a group of collaborating agents and help debug a complex system.

We have integrated into Bond the JESS expert shell developed at Sandia National Laboratory as a distinct strategy able to support reasoning. Bond messages allow for embedded programs written in JPython and KIF.

Agent security is a critical issue for the system because the ability to assemble and control agents remotely as well as agent mobility, provide unlimited opportunities for system penetration. Once again the fact that agents are native Bond objects leads to an elegant solution to the security aspect of agent design. Any Bond object, agents included, can be augmented dynamically with a security probe providing a defense perimeter and screening all incoming and outgoing messages.

The components of a Bond agent shown in Figure 5 are:

- The **model of the world** is a container object which contains the infor-

mation the agent has about its environment. This information is stored in the form of dynamic properties of the model object. There is no restriction of the format of this information: it can be a knowledge base or an ontology composed of logical facts and predicates, a pre-trained neural network, a collection of meta-objects or different forms of handles of external objects (file handles, sockets, etc).

- The **agenda** of the agent, which defines the goal of the agent. The agenda is in itself an object, which implements a boolean and a distance function on the model. The boolean function shows if the agent accomplished its goal or not. The distance function may be used by the strategies to choose their actions.
- The **finite state machine** of the agent. Each state has an assigned strategy which defines the behavior of the agent in that state. An agent can change its state by performing *transitions*. Transitions are triggered by internal or external *events*. External events are messages sent by other agents or objects. The set of external messages which trigger transitions in the finite state machine of the agent defines the *control subprotocol* of the agent.
- Each state on an agent has a **strategy** defining the behavior of the agent in that state. Each strategy performs actions in an infinite cycle until the agenda is accomplished or the state is changed. Actions are considered atomic from the agent's point of view, external or internal events interrupt the agent only between actions. Each action is defined exclusively by the agenda of the agent and the current model. A strategy can terminate by triggering a transition by generating an internal event. After the transition the agent moves in a new state where a different strategy defines the behavior.

All components of the Bond system are objects, thus Bond agents can be assembled dynamically and even modified at runtime. The behavior of an agent is uniquely determined by its model (the model also contains the state which defines the current strategy). The model can be saved, transferred over the network.

A `bondAgent` can be created statically, or dynamically by a factory object `bondAgentFactory` using a *blueprint*. The factory object generates the components of the agent either by creating them, either by loading them from persistent storage. The agent creation process is summarized in Figure 6

3.4 Networks of Cooperating Agents and the Nervous System

A complex system consists of a variety of software components created independently, a heterogeneous computing environment, and vast amounts of data. Software agents can "glue" together these elements and act as a nervous

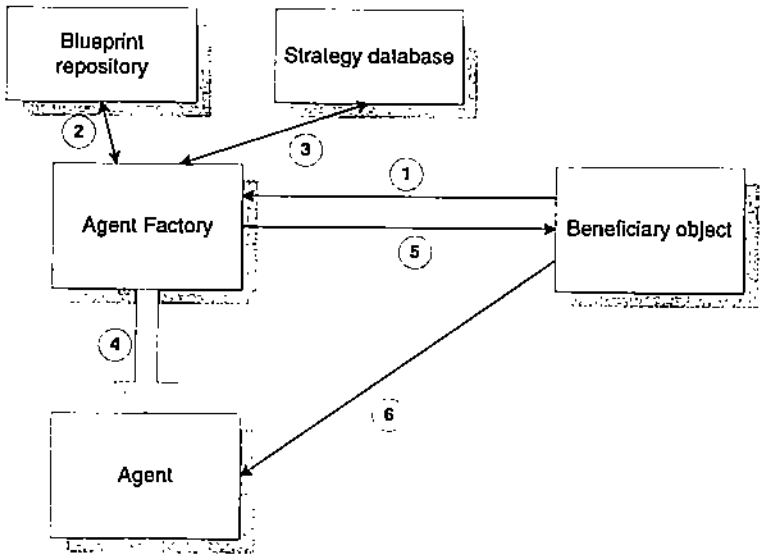

Fig. 6. Creating an agent remotely using an agent factory. (1) The beneficiary object sends a create-agent message to the agent factory (2) The blueprint is fetched by the agent factory from a repository or extracted from the message (3) The strategies are loaded from the strategy database (4) The agent is created (5) The id of the agent is communicated back to the beneficiary, and (6) The beneficiary object controls the new agent

system performing command and control functions. Thus the ultimate goal of Bond is to support a network of cooperating agents distributed across many sites.

The group of agents at each site form an agent stack tailored to a specific function as shown in Figure 7. An *application* site is concerned with the execution of tasks posed by an end user who is the beneficiary of all the actions triggered by local agents. The agent stack at an application site consists of an assortment of application support agents including a personal assistant, a software maintenance agent, and a user resource manager. Other agents in the stack include coordinator agents, monitoring agents, scheduler agents and possibly other agents. These agents may be independent or they may be planes of a complex agent. The difference is that independent agents communicate only through message-passing while individual planes communicate through shared-memory.

The *personal assistant* agent maintains the knowledge pertinent to user's preferences, personal data, and acts as an intelligent assistant. When necessary it activates local and remote agents. It supports strategies to ensure data privacy, to migrate data, to start, control and stop other agents, to initiate remote execution of programs, to maintain a history of user's transactions.

A *maintenance* agent is another application support agent. It builds a knowledge base covering a set of programs, reports errors to a remote peer agent,

monitors a number of sites for new products, and responds to help requests.

A *user resource managing* agent maintains information about hardware and software resources distributed across the network.

A *coordinator* agent is responsible with the collaborative aspects of end user's activity. It maintains a set of projects and goals related to each project, deadlines, collaborators.

A *scheduling* agent is responsible to execute an activity flow graph whose individual nodes are legacy applications. It involves several planes, one responsible to relate activities with programs, a plane to map programs to individual sites, one to coordinate the execution at each site and one plane for each *wrapper*, an agent supervising the execution of a legacy program.

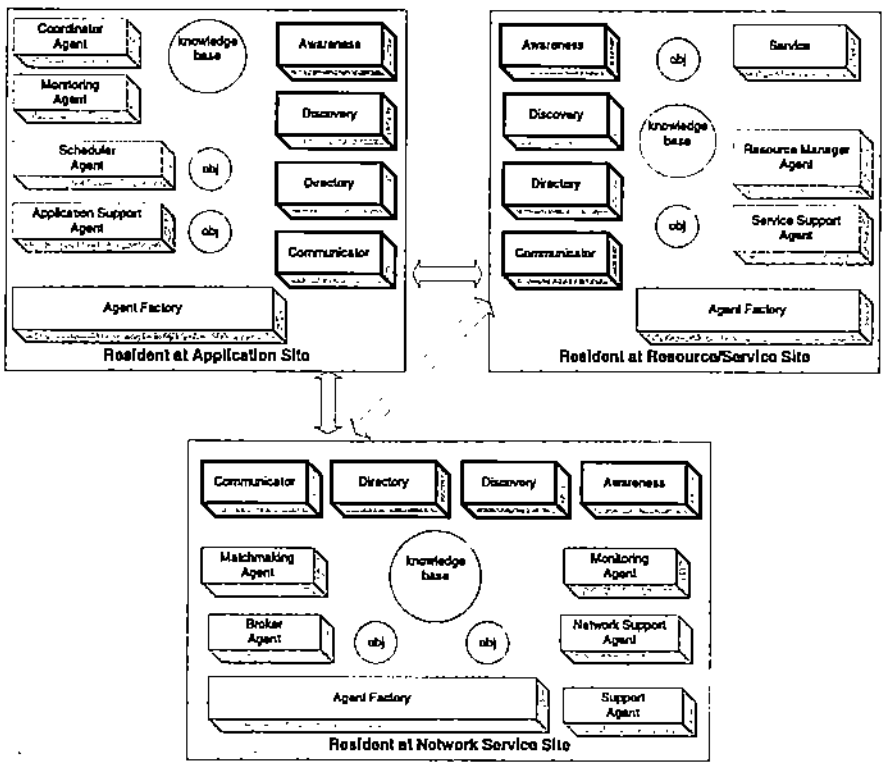

Fig. 7. Bond agent architecture. The agent stacks at a user, service, and network site.

The agent stack at a *service or resource* site is different than the one at the application site. The agents are started in behalf of the provider of service or the local resource manager. The *resource or service management* agent enforces policies imposed by the service provider and ensures the quality of service. It advertises the service to network brokers, replicates the service to other sites under its control to guarantee an agreed upon level of service, and monitors satellite services.

Typical *service support* agents include software maintenance agents, the ones responsible to gather error reports, answer questions about the service or resource, and inform the client side whenever a new software release takes place or a new service is provided. Other support agents provide accounting and access control. Last but least the agents stack at *network sites* provide brokerage and matchmaking services.

To illustrate the use of Bond, we describe in [25] a network of PDE solvers. A coordinator agent and a network of mediator and solver agents control a set of legacy PDE solvers running on a wide-area network. The coordinator agent controls the decomposition of a data domain into sub-domains and the activation of all other agents. The mediator agents implement interface relaxation policies and the solver agents act as wrappers around the legacy application. In [26] and [27] we describe the use of Bond as an infrastructure for Problem Solving Environments.

4 Conclusions

In this paper we argue that increasingly complex systems like computing grids require novel approaches to software composition and the development of an infrastructure to hide the more intricate aspects of concurrency, distribution of code and data, and timing constrains. We believe that analogies with biological phenomena are very useful in designing and implementing complex software system.

To illustrate these ideas we present a distributed-object, agent-based infrastructure and discuss various metaphors from structural biology, genetics, immunology, and neurology used in the design of Bond.

The Bond system consists of four packages, core, agents, services, and applications. A beta version of the Bond system was released in mid March 1999 under an open source license, LPGL, and can be downloaded from our web site, **http://bond.cs.purdue.edu**.

5 Acknowledgments

The work reported in this paper is partially supported by the National Science Foundation grants BIR-9301210 and MCB-9527131, by the California Institute of Technology, under the Scalable I/O Initiative, by a grant from Intel Corporation, and by the Computational Science Alliance and the NCSA at the University of Illinois.

References

[1] Rosenblatt, F. *"The perceptron: A perceiving and recognizing automaton"*, 85-460-1, Project PARA Cornell Aeronautical Laboratory Ithaca, NY, 1957.

[2] Hopfield, J. *Neural Networks and Physical Systems with Emergent "Collective Computational Abilities" Proceedings of the National Academy of Sciences*, 1982 vol.79, pp. 2554.

[3] McClelland, J.L. and D. E. Rumelhart, *"Parallel Distributed Processing: Explorations in the Microstructures of Cognition, Volume 2: Psychological and Biological Models"*, MIT Press Cambridge, Mass. 1986

[4] Koza, J.R. *"Genetic Programming: On the Programming of Computers by Means of Natural Selection"*, MIT Press 1992.

[5] *The Grid, Blueprint for a New Computing Infrastructure*, Foster, I. and C. Kesselman, Eds., Morgan Kaufmann, (1998).

[6] Fabre, J.C. and T. Perennou. *"A Metaobject Architecture for Fault Tolerant Distributed Systems: The FRIENDS Approach"*, *IEEE Trans. on Computers*, Vol 47, No 1, pp 78-95, 1998.

[7] Branden, C., and J. Toose. *"Introduction to Protein Structure"* Garland Publishing 1991.

[8] Stroud, R.J. *"Transparency and Reflection in Distributed Systems"*, *ACM Operating Systems* vol. 22, no. 2 pp. 99-103, 1993.

[9] Chandy, Mani, K. *"Caltech Infospheres Project Overview: Information Infrastructures for Task Forces."*

[10] Gamma, E., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison Wessley, (1994).

[11] Johnson W.E. *Real-time widely distributed instrumentation systems* . In *The Grid. Blueprint for a New Comput. Infrastructure*, (I. Foster and C. Kesselman, eds.), Morgan Kaufmann Publishers, 74-104, 1998.

[12] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, J. Irwing *Aspect oriented programming Proc. European Conference of Object-Oriented Programming (ECOOP)*, 1997.

[13] Kiczales, G., J. M. Ashley, L. Rodriguez, A. Vahdat and D. G. Bobrow *Metaobject Protocols - Why We Want Them and What Else They Can Do* in *Object-Oriented Programming - The CLOS Perspective*, MIT Press, 1993.

[14] Bradshaw, J. M., *An Introduction to Software Agents*, in *Software Agents*, J.M. Bradshaw, Ed., MIT Press, pp. 3-46, 1997.

[15] Finin, T., et al. *"Specification of the KQML Agent-Communication Language"* DARPA Knowledge Sharing Initiative draft, June 1993.

[16] Finn, T., Y. Labrou, and J. Mayfield *KQML as an Agent Communication Language*, in *Software Agents*, J.M. Bradshaw, Ed., MIT Press, pp. 291-316, 1997.

[17] Foner, L.N. *What's an Agent, Anyway? A sociological Case Study* Agents Memo 93-01, Agents Group, MIT Media Lab, 1993.

[18] Franklin, S. and A. Graesser, *Is it an Agent, or just a Program?, Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages*, Springer Verlag, 1996.

[19] Genesereth, M. R., *An Agent-Based Framework for Interoperability*, in *Software Agents*, J.M. Bradshaw, Ed.,MIT Press, pp. 317-345, 1997.

[20] Jennings, N. R., K. Sycara, M. Woolridge, *A Roadmap of Agent Research and Development*, in *Autonomous Agents and Multi-Agent Systems*, 1, pp. 275-306, 1998.

[21] Bölöni, L., and D.C. Marinescu, *An Object-Oriented Framework for Building Collaborative Network Agents. Intelligent Systems and Interfaces*, (A. Kandel, K. Hoffmann, D. Mlynek, and N.H. Teodorescu, eds). Kluewer Publising House, (1999), (in press).

[22] Bölöni, L., R. Hao, K.K. Jun, and D.C. Marinescu, *Structural Biolgy Metaphors Applied to the Design of a Distributed Object System, Proc. Second Workshop on Bio-Inspired Solutions to Parallel Processing Problems*, in LNCS, vol 1586, Springer Verlag, pp. 275-283, 1999.

[23] Hao, R., K.K. Jun, and D.C. Marinescu, *Bond System Security and Access Control Model, Proc. IASTED Conference on Parallel and Distributed Computing and Networks*, pp. 520-524, 1998.

[24] Hao, R., L. Bölöni, K.K. Jun, and D.C. Marinescu, *An Aspect-Oriented Approach to Distributed Object Security, Proc. 4-th IEEE Symp. on Computers and Communications*, IEEE Press, 1999, (in print).

[25] Tsompanopoulou, P., L. Bölöni, D.C. Marinescu, and J.R. Rice, *The Design of Software Agents for a Network of PDE Solvers, Proc. Workshop on Agent Technologies for High Performance Computing, at Agents 99*, pp. 57-68, 1999.

[26] Marinescu, D.C., *An Agent-Based Design for Problem Solving Environment, Proc. Workshop on Prallel/High Performance Scientific Computing, POOSC'99* 1999, (in press).

[27] Marinescu, D.C., and Bölöni L., *A Component-Based Architecture for Problem Solving Environments*, 1999, (in preparation).

[28] Cornea-Hasegan, M.C,, Z. Zhang, R.E. Lynch, D. C. Marinescu, A. Hadfield, J.K. Muckelbauer, S. Munshi, L. Tong and M.G. Rossmann. *"Phase Refinement and Extension by Means of Non-crystallographic Symmetry Averaging using Parallel Computers."* Acta Cryst D51, pp 749-759, 1995

[29] Martin, I., D.C. Marinescu, R. E. Lynch, and T. S. Baker. *"Identification of Spherical Virus Particles in Digitized Images of Entire Micrographs"* Journal of Structural Biology, 120, pp. 146-157, 1997.

[30] Martin, I. and D.C. Marinescu *"Concurrent Computations and Data Visualization for Structure Determination of Spherical Viruses" IEEE Computational Science and Engineering*, October-December, pp. 40-51, 1998.

[31] Lynch, R.E., D.C. Marinescu, H. Lin, and T. S. Baker. *"Parallel Algorithms for 3D Reconstruction of Asymmetric Objects from Electron Micrographs" Proc. 13th International Parallel Processing Symposium*, IEEE Press, pp. 632-637, 1999.