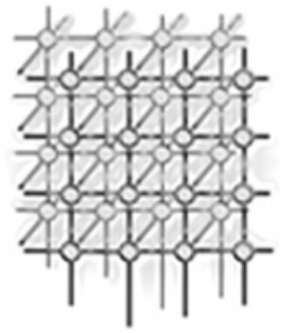
CONCURRENCY: PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper. 2000; **12**:845–861

Agent based scientific simulation and modeling



L. Bölöni, D. C. Marinescu*, J. R. Rice,
P. Tsompanopoulou and E. A. Vavalis

Computer Science Department, Purdue University

SUMMARY

The simulation and modeling of complex physical systems often involves many components because (i) the physical system itself has components of differing natures, (ii) parallel computing strategies require many (somewhat independent) components, and (iii) existing simulation software applies only to simpler geometrical shapes and physical situations. We discuss how agent based networks are applied to such multi-component applications. The network agents are used to (a) control the execution of existing solvers on sub-components, (b) mediate between sub-components, and (c) coordinate the execution of the ensemble. This paper focuses on partial differential equation (PDE) models as an instance of the approach and describes the implementation of networks using the PELLPACK problem solving environment for PDEs and the Bond system for agent based computing. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: software agent; inference; data-intensive applications; problem solving environments; simulation and modeling

1. INTRODUCTION

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 capable of simulating physical systems of interest to computational sciences and engineering.

Software agents may provide an answer to the increased complexity of the software systems expected to intelligently anticipate and adapt to the needs of dynamically distributed applications. Yet different groups have radically different views of what software agents are [1] and what applications could benefit from the agent technology, and many have a difficult time sorting out the reality from fiction in this rapidly moving field. A software agent is expected to exhibit to some degree attributes broadly classified into three groups [2].

*Correspondence to: D. C. Marinescu, Computer Science Department, Purdue University, West Lafayette, IN 47907, U.S.A.

Received 5 March 2000

- Agency: measures the degree of autonomy and authority of an agent. It reflects the nature of the interactions between an agent and the user, other agents, data, services. Asynchrony and reactivity are expressions of this facet of agent behavior.
- Intelligence: reflects the degree of preferences, reasoning, planning, and learning behavior.
- Mobility: the ability to travel through a network.

A considerable body of work has been devoted to creating agents able to meet the Turing test by emulating human behavior. Such agents are useful for a variety of applications in science and engineering, e.g. deep space explorations, robotics, and so on. Our view of an agent is slightly different [3]. For us a software agent is an abstraction for building complex systems. Our main concern is to develop a constructive framework for building collaborative agents out of ready-made components and to use this infrastructure for building complex systems including Problem Solving Environments, PSEs [4]. In this paper we examine alternative means to exploit the advantages of code mobility, object-oriented design, and agent technology for high performance distributed computing. To use a biological metaphor [5,6], software agents form a nervous system and perform command and control functions in a PSE. The agents themselves rely on a distributed object system to communicate with another. Though the agents are mobile, some of the components of the PSE are tightly bound to a particular hardware platform and software environment and cannot be moved with ease.

The primary function of a Problem Solving Environment is to assist computational scientists and engineers in carrying out multiple computations. We use the term dynamic workflow to denote both the static and the dynamic aspects of this set of computations. We argue that there are several classes of high performance computing applications that can greatly benefit from the use of agent-based PSEs:

- naturally distributed applications;
- data intensive applications;
- applications with data-dependent or non-deterministic workflows.

Many problems in computational science and engineering are naturally distributed, involve large groups of scientists and engineers, large collections of experimental data and theoretical models, as well as multiple programs developed independently and possibly running on systems with different architectures. Major tasks can and should be delegated to a PSE, including coordination of various activities, enforcing a discipline in the collaborative effort, discovering services provided by various members of the team, transporting data from the producer site to the consumer site, and others. The primary functions of agents in such an environment are: planning, scheduling and control, resource discovery, management of local resources, and use-level resource management.

Data-intensive applications are common to many experimental sciences and engineering design applications. As sensor-based applications become pervasive, new classes of data-intensive applications are likely to emerge. An important function of the PSE is to support data annotation. Once metadata describing the actual data are available, agents can automatically control the workflow, allow backtracking and restart computations with new parameters of the models.

Applications like data acquisition and analysis in physics, chemistry, biology, climate and oceanographic modeling often rely on many data collection points, and the actual workflow depends both upon the availability of the data and the confidence we have in the data. The main function of the agents in such cases is the dynamic generation of the workflows based upon available information.

A contribution of this paper is the idea of combining software agents with legacy applications to solve data intensive problems, the topic of Section 2 of the paper. In Section 3 we outline domain

decomposition, a common approach to simulating large scientific and engineering problems. The Bond system is introduced in Section 4, in Section 5 we discuss the design of a network of PDE solvers and in Section 6 we present an application. We conclude that the combination of Bond and PELLPACK allows one to quickly construct models and simulations for complex physical systems. This success should be repeatable in a variety of science and engineering areas.

2. COMBINING AGENTS AND LEGACY CODE FOR DATA INTENSIVE PROBLEMS

Data parallelism is a common approach to reducing the computing time and to improving the quality of the solution for data-intensive applications. In this case individual nodes execute the same computations but, on different segments of the data, domain-decomposition is probably the best known instance of this paradigm. Often the algorithm for processing each data segment is rather complex and the effort to partition the data, to determine the optimal number of data segments, to combine the partial results, and to adapt to a specific computing environment and to user requirements must be delegated to another program. Mixing control and management functions with the computational algorithm leads in such cases to brittle and complex software.

In this paper we advance the idea of mixins, combinations of legacy code and software agents [7] where agents perform control and management functions for data parallel applications. The principal advantages of this approach are as follows.

- Separation of concerns. The processing algorithms are created by a scientist or engineer knowledgeable in an an application area, e.g. chemistry, physics, biology, materials, and so on, the control functions and structures by a computer scientist.
- Legacy code requires minor or no adaptation at all. This leads to a substantial reduction of the development time and cost and to increased reliability.
- The resulting ensemble is more adaptive, more functional, and easier to use as shown in the example given below.

We also recognize potential problems with this approach. First, the overall performance may be affected, and there is an additional overhead for communication and control functions. Agents are rather slow, they are written in Java and some of their functions, e.g. the inference, require a fair amount of iteration. But the control functions are exercised seldom and in most cases the overall performance is not affected, the control functions add a few seconds to minutes or hours required by the computations. In our experience the total time required by agents is less than 1% of the total execution time. Second, though the agents are mobile, the legacy code is not. An agent may choose to run one instance of the legacy code but only on systems that support that. Porting a large legacy application to a new hardware architecture and operating system is a major endeavor.

Typically the agent has as input a set of rules and facts. Some of the facts are rather static, e.g those describing the configuration of the system or the characteristic of the problems, while others are more dynamic, e.g. partial results needed to make a decision for the next step, the system load needed to adapt to the environment. The agent and the legacy application interact through a *problem description file* produced by the agent. This file contains the location of the data, parameters of the model, parameters of the algorithms, and hints. The function of the agent is thus to generate the problem description file,

e.g. using inference, and then start up the legacy application and monitor it throughout its execution. The agent acts as a *wrapper* of the legacy application.

Let us give an example illustrating the interactions between an agent and a legacy application for a particular application that has already been parallelized but to be used effectively requires a substantial amount of knowledge. The application is the 3D reconstruction of a spherical virus from its 2D projections obtained experimentally with an electron microscope [8]. The agent controlling this application is under development as a component of a larger effort to build a Virtual Laboratory for Computational Biology.

The 3D reconstruction process is iterative—it starts with (i) a set of n, 2D projections each of size $p \times p$ obtained experimentally and (ii) an initial model of the virus given as a 3D lattice of electron density values, of size $m \times m \times m$. Here the number of experimental images may vary from few hundreds for an icosahedral virus to possibly 100 000 for an asymmetric one. The size of a projection may be as large as 500×500 pixels and the electron density map may consist of 500^3 real values.

The first step of the process is to determine the orientation of each projection and once the orientation is known we proceed to the 3D reconstruction and produce a new electron density map. The process is repeated until the convergence criteria are met.

We focus only on the orientation determination. The idea of the algorithm is to construct a database of calculated projections using as input the current value of the electron density map and then to compare each of the n images obtained experimentally against each calculated image in the database, whose orientation is known. We assign the orientation of the experimental projection based upon the correlation coefficient between the two. Thus the algorithm is straightforward and embarrassingly parallel. The only problems are the structural biology knowledge necessary to formulate the problem and the amount of data involved. An image may consist of 0.5 MB of data and the database of calculated images may consist of about 4000 images or about 2 GB for a virus with icosahedral symmetry. For an asymmetric virus the database may be two orders of magnitude larger.

Let us discuss briefly the problem of symmetry. Some viruses exhibit icosahedral symmetry—they resemble a soccer ball and are made out of 60 identical wedges—but others are not symmetric. When we say that the virus exhibits some symmetry we only refer to its protein shell, the virus core containing genetic material does not. If we know that a virus has icosahedral symmetry then we only need to know the electron density in a wedge. Moreover, for an icosahedral virus from every single 2D projection we can construct 60 others using the symmetry operators. Sometimes, the symmetry of the virus is not known and we are interested to determine the axes of symmetry. In this case we have to construct the full database of calculated images and for each experimental image determine its correlation coefficient with every single image in the upper hemisphere, then rank the correlation coefficients and deduce the symmetry.

We outline some of the options we have and the decisions to be made. Our first algorithm is based upon a multi-resolution approach. For an icosahedral virus we construct first a low-resolution database (at 3 deg resolution we have only 67 images), find out the approximate orientation of each image, then we construct the high resolution database at say 0.3 deg resolution (it has about 4000 images) and conduct the search of the optimal orientation on it. The high resolution database is distributed across nodes and experimental images are sorted based upon the approximate orientation determined in the first phase. Each node will process only a fraction of the total number of experimental images. The choices to be made are:

- How to get the data in each node. We can read the electron density file in parallel in each node or have one node read the electron density data and broadcasts it to the other nodes. This decision is based upon the actual machine configuration, the presence of a parallel file system, the speed of the interconnection network, the amount of main memory available in each node, the architecture of the system, distributed memory or a set of shared memory nodes interconnected together.
- How to construct the low resolution database. We can construct it in one node and broadcast it to all other nodes or have each node construct it. Though the first approach seems more effective we have to keep in mind that for the second step each node needs the electron density map to construct its own segment of the database.
- How to distribute the data and computations amongst nodes for the second phase. We have the choice of dividing the database evenly among nodes, but in that case the number of images to be processed by each node is likely to be different, and thus the load is imbalanced. The alternative is to distribute evenly amongst nodes the number of experimental images but risk that the segment of the database assigned to one node is much larger than the one for other nodes. Hints about the choice are available at the end of the first phase.
- How to deal with a shared system. If other processes are running on some of the nodes the optimal load distribution is affected by them.
- How to deal with a cluster of heterogeneous workstations. The load distribution is affected by the relative speed of individual nodes.

The problem is further complicated when we do not know the symmetry. The multi-resolution algorithm described above will not work at all for large virues because at low resolution their features are completely lost. Thus for a large particle whose symmetry is not known we need to construct from the beginning a very large database consisting of, say, 100 000 calculated images of 0.5 MB each. Because of the sheer amount of data we need a large system configuration; e.g. for $N = 100$ nodes, each one will be assigned 1000 calculated images or about 5 GB. Then we construct a circular list of experimental images and process groups of size N of them in a pipelined fashion. In step one we assign experimental image I_1 to processor P_1, I_2 to $P_2 \ldots I_N$ to P_N and compute the best correlation coefficients, $\sigma_1^1, \sigma_2^2, \ldots, \sigma_N^N$. Then in step two I_1 migrates to P_2, I_2 to $P_3, \ldots, I_N$ to P_1 and we compute $\sigma_1^2, \sigma_2^3, \ldots, \sigma_N^1$. After N steps we determine the orientation by selecting $\sigma_i = \min[\sigma_i^j]$ $\forall i, j \in [1, N]$. Then we process the next batch of N experimental images, I_{N+1} to I_{2N}, and continue until the entire pool of experimental images is exhausted.

We omitted many details of the algorithm; for example, the Polar Fourier Transforms of the calculated and experimental images are first computed and then compared to determine the orientation of the experimental one. Even from this relatively brief description it is clear that an agent provided with a set of rules is a better choice than to hardwire in a Fortran program the information necessary to make the decisions outlined above.

As pointed out earlier, in some cases one can achieve parallelism using sequential legacy codes without the need to modify them at all. Whenever we can apply a divide and conquer methodology based upon the partitions of the data into sub-domains, solve the sub-problems independently in each sub-domain, and then resolve with ease the eventual conflicts between the individual workers we have an appealing alternative to code parallelization. The agents should be capable of coordinating the execution and mediating conflicts. This is the case of the network of PDE solvers discussed in more depth in the next sections.

3. DOMAIN DECOMPOSITION AND COLLABORATING SIMULATORS

There are three computational approaches to simulating large scientific problems. The first and most common approach is to discretize the geometrical domain using grids or meshes to create a large discrete problem. These grids or meshes are then partitioned to create a set of inter-connected discrete problems. This is simple *domain decomposition* [9] and the coupling between components (discrete problems) is rather tight as the mathematical model along interface points or elements is discretized into equations that involve details from both neighboring components. The second and oldest approach is Schwarz splitting, which decomposes the geometrical domain into components with a small overlap. The mathematical models on each component can then be solved independently in some way and the *Schwarz alternating method* is applied iteratively to compute the global solution. Of course, some discretization method is applied to the solution process on each individual component. The overlapping creates a serious complication in the Schwarz method [10] even when the global problem has a simple geometry. The method has become more feasible with the discovery of non-overlapping domain versions. The third and newest approach is *interface relaxation* [11], where the geometrical domain is decomposed into sub-domains, each with its own mathematical model. Along the interfaces between sub-domains one must satisfy interface conditions derived from the physical phenomena (e.g. continuity of mass or temperature, conservation of momentum). The models on each sub-domain are solved in the inner loop of the interface relaxation iteration method to compute the global solution. These methods use one of a variety of 'smoothing' formulas to reduce the error in satisfying the interface conditions.

The goals of handling different physical models, using parallel computers and reusing existing software all lead to the need for high flexibility and loose coupling between components in the computation. These three approaches have similar goals but are quite different in their generality and flexibility. The tight coupling of domain decomposition requires that neighboring components have a lot of shared information about their discretizations. Further, this approach is quite awkward when the models are different on neighboring components. The *mortar method* creates specialized refinements of the models and meshes along the interfaces to accommodate changes in models across interfaces. Overlapping Schwarz methods are similarly constrained to a single physical model and also create a tight coupling between neighboring sub-domains. The non-overlapping Schwarz methods are restricted to a single mathematical model for neighboring sub-domains. The interface relaxation approach imposes no coupling conditions, except those inherent in the mathematical models, and it provides maximum generality and flexibility.

Since interface relaxation is relatively new and unfamiliar, we outline the method and then present one specific instance. The method assumes that one can solve exactly any single PDE on any simple domain or, more realistically, that, given such a PDE problem, we can select a highly accurate solver for it from a library. The interface relaxation method uses a library of 'single, simple-domain, exact' PDE solvers to solve composite PDE problems. It is an iterative method of the classical type, based on relaxation, as follows.

1. Guess solution values (and derivatives if needed) on all sub-domain interfaces.
2. Solve all single PDEs exactly and independently on all the sub-domains with these values as boundary conditions.

3. Compare and improve the values on all interfaces using a relaxer (discussed below).
4. Return to Step 2 until satisfactory accuracy is achieved.

The simplest relaxers do some sort of 'smoothing' of values on the interfaces and averaging is a good mental model for a relaxation formula.

The attraction of interface relaxation is threefold. First it allows the accurate coupling of independent models and the reuse of PDE software that handles single-phenomenon models. Second it uncouples the parallelism of the computation somewhat from that of the machines used. Finally, it is intuitively consistent with a person's view of the geometry and physical models of a composite PDE problem.

Interface relaxation is an iteration defined at the continuum (mathematical) level; its convergence properties are a question of mathematical analysis, not of numerical analysis. Let U_N and V_N be the PDE solution values at iteration N on opposite sides of an interface. Assume for simplicity that the interface conditions to be satisfied are a continuity of value ($U_N = V_N$) and normal derivative ($\partial U_N/\partial n = -\partial V_N/\partial n$). Then two simple relaxation formulas for U and V are:

$$V_{N+1} = U_{N+1} = (U_N + V_N)/2 - f(\partial U_N/\partial n + \partial V_N/\partial n)$$
$$V_{N+1} = U_{N+1} = \omega U_N + (1-\omega)[\alpha(U_N - V_N)^2 + \beta(\partial U_N/\partial n + \partial V_N/\partial n)^2]$$

where f, ω, α, and β are relaxation parameters. Approximately ten relaxation formulas appear in the literature, seven of which are cataloged in [11]. One of the important open questions concerns the comparative performance of these methods and procedures for computing good relaxation parameters.

The convergence analysis of interface relaxation presents formidable mathematical challenges; almost any question asked will be both hard and open. Even for the single-PDE case—one global PDE or domain decomposition—work on convergence analysis has appeared rarely, starting about 10 years ago [10] and then more recently in 1992 [9].

Given that theoretical analysis is intractable for the moment, we use experiments to provide guidance and insight for interface relaxation. Numerous experiments done in recent years indicate that interface relaxation converges for a wide variety of problems and relaxers. The convergence is sometimes very fast, other times not. The results in [12] show (for a single problem) that the rate of convergence is independent of the number of sub-domains, and this was verified by experiments using up to 500 sub-domains. There is reason to be hopeful that, as we better understand interface relaxation, it can become a very useful method for solving composite PDEs. (A crude form of interface relaxation already in fairly widespread use simply involves 'trading' current values across interfaces without any relaxation. This method makes the most sense in time-varying problems, but we are not aware of any attempts to analyze the effects of the errors involved.)

A 1997 result [13] illustrates the progress in the mathematical analysis of interface relaxation. We consider the Helmholtz equation $Lu = u_{xx} + u_{yy} - \gamma u = f(x, y)$ on the rectangular domain $\Omega = [-x_1, x_2] \times [-1, 1]$ with zero boundary conditions. The domain Ω is partitioned into two sub-domains Ω_1 and Ω_2 along the line $x = 0$ (called Γ), and the solutions on the left Ω_1 and right Ω_2 are denoted by $u_1(x, y)$ and $u_2(x, y)$. Given initial guesses $u_1^{(0)}$ and $u_2^{(0)}$ with zero values as the boundary of $-\Omega$, the interface relaxation iteration is defined by the following set of four PDE problems and boundary conditions. These boundary conditions comprise the relaxation formulas and combine values

on Γ of previous solutions to provide better conditions for the next step.

$$
\begin{aligned}
Lu_1^{(2k+1)} &= f \text{ in } \Omega_1, \quad u_1^{(2k+1)} = \alpha u_1^{(2k)} + (1-\alpha)u_2^{(2k)} \text{ on } \Gamma \\
Lu_2^{(2k+1)} &= f \text{ in } \Omega_2, \quad u_2^{(2k+1)} = \alpha u_1^{(2k)} + (1-\alpha)u_2^{(2k)} \text{ on } \Gamma \\
Lu_1^{(2k+2)} &= f \text{ in } \Omega_1, \quad \frac{\partial u_1^{(2k+2)}}{\partial \nu^1} = \beta \frac{\partial u_1^{(2k+1)}}{\partial \nu^1} + (1-\beta)\frac{\partial u_2^{(2k+1)}}{\partial \nu^1} \text{ on } \Gamma \\
Lu_2^{(2k+2)} &= f \text{ in } \Omega_2, \quad \frac{\partial u_2^{(2k+2)}}{\partial \nu^2} = \beta \frac{\partial u_2^{(2k+1)}}{\partial \nu^1} + (1-\beta)\frac{\partial u_1^{(2k+1)}}{\partial \nu^2} \text{ on } \Gamma
\end{aligned}
$$

The relaxation parameters α and β are determined to accelerate the convergence; experiments and theory both suggest that $\alpha = \beta = 1/2$ are good values. The following theorem is established in [13].

Theorem. *Set* $s = (x_1 + x_2)\min\{x_1, x_2\}$. *If* $\alpha = 1/2$ *the sequences* $u_1^{(j)}, u_2^{(j)}$ *converge provided* $0 < \beta \leq 2/3$ *and*

$$(1-\beta)^2 s^2 - 2\beta(1-\beta)s - 2(1-2\beta) < 2$$

If $\beta = \beta_{opt} = (s^2 + s - 2)/(s^2 + 2s)$ *then the error decays exponentially with ratio* $(s-2)/2s$.

This theorem provides a nice result for a very simple model problem, but extending such analysis further is quite difficult. These results are confirmed by experiments and the optimum β is seen to provide faster convergence than two other interface relaxation methods. Mo Mu [12] provides another set of mathematical theorems with a different flavor.

4. BOND AGENTS

Bond [3], is a distributed-object, message-oriented system, providing a constructive framework for building collaborative network agents. Several distributed object systems provide support for software agents: Infospheres (`//www.infospheres.caltech.edu/`) and Bond (`http://bond.cs.purdue.edu`) are academic research projects, while Objectspace Voyager (`//www.objectspace.com`) and IBM Aglets (`www.trl.ibm.co.jp/aglets/index.-html`) are commercial systems. Bond is released under an open source license, LPGL, and was used as a workflow enactment engine supporting dynamic workflows [14], for an adaptive video service [15], for resource discovery in a wide area distributed system [16], for the design of a network of PDE solvers [17], and for other applications.

Bond uses KQML [18] as a meta-language for inter-object communication. Support for XML based communication was added to the system recently. 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 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 are therefore order-independent.

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

Shadows are proxies for remote objects. 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. The discovery subprotocol allows an object to determine the set of subprotocols understood by another object. Examples of other subprotocols are monitoring, security, agent control, and the property access subprotocol understood by all objects.

The transport mechanism between Bond residents is provided by a *communicator* object with four interchangeable communication engines based upon: (a) UDP, (b) TCP, (c) Infospheres, (info.net), and (d) IP Multicast protocols.

Probes are objects attached dynamically to Bond objects to augment their ability to understand new subprotocols and support new functionality. A *security probe* screens incoming and outgoing messages to an object [19,20]. The security framework supports two authentication models, one based upon *username, plain password* and one based upon the *Challenge Handshake Authentication Protocol, CHAP* [21]. Two access control models are supported, one based upon the *IP address (firewall)* and one based upon an *access control list*. *Monitoring probes* implement a subscription-based monitoring model. An *autoprobe* allows loading of probes on demand.

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 (CORBA-like) interface repositories.

A first distinctive feature of the Bond architecture, described in more detail in [3], is that agents are integrated with the distributed object model. This guarantees that agents and objects can communicate with one another and that 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 attention 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 describe next.

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 1. 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.

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 the 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 have started building a strategy repository. Examples of actions packed into

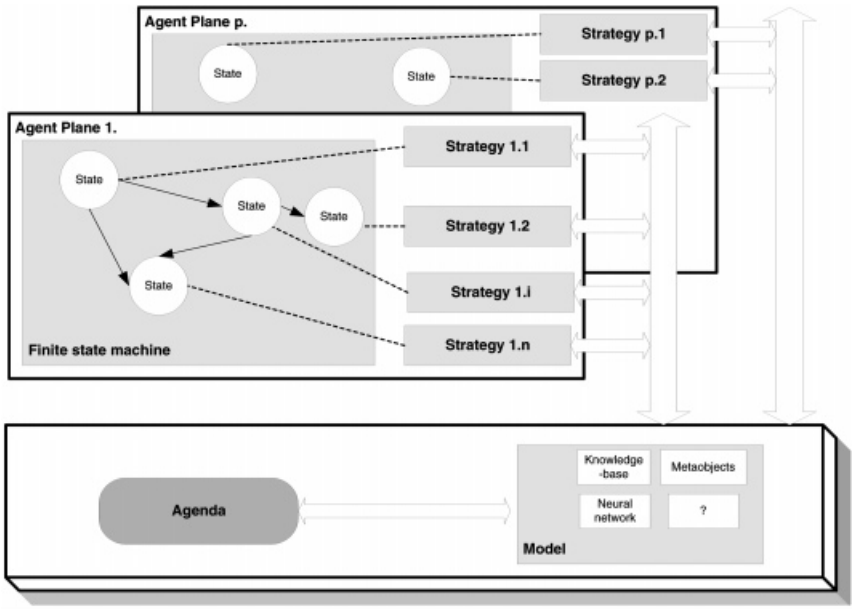


Figure 1. The model of a Bond agent.

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 instead ready-made strategies.

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. XML-based blueprints are now supported in Bond. Every Bond-enabled site has an 'agent factory' capable of creating 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 is created, the agent control sub-protocol 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 of executing 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, selecting 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 can

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.

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. Like any Bond object, Bond agents 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 1 are as follows.

- The **model of the world**—a container object which contains the information the agent has about its environment. This information is stored in the form of dynamic properties of the model object and thus it allows for various representation formats. 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 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, and 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 and 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 or by loading them from persistent storage. The agent creation process is summarized in Figure 2. The beneficiary can be the user or another agent capable of creating the blueprint [22,23].

The methodology to create an agent in Bond is: (a) write down a brief description of the actions in each state; (b) create a state transition diagram for each agent; (c) search Bond databases for strategies suitable for new agents; (d) write new strategies whenever necessary.

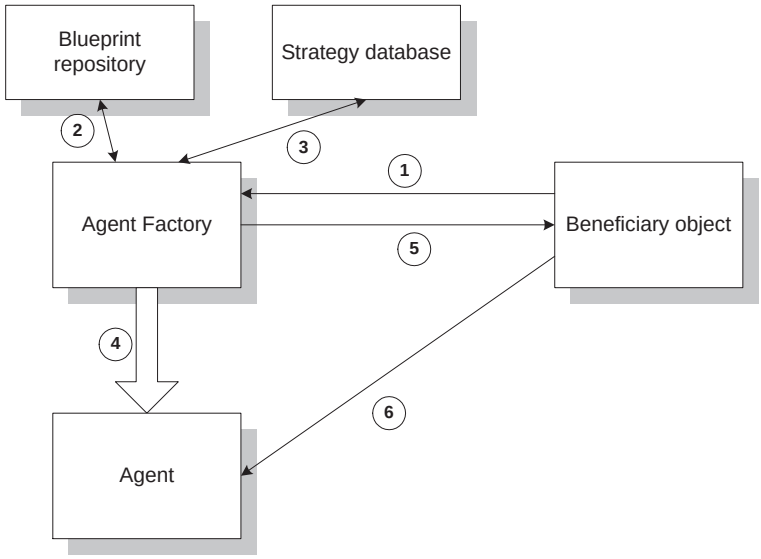


Figure 2. Creating an agent remotely using an agent factory. (1) The beneficiary object sends a create-agent message to the agent factory. (2) The blueprint is fetched by the agent factory from a repository or extracted from the message. (3) The strategies are loaded from the strategy database. (4) The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

5. A NETWORK OF PDE SOLVING AGENTS

Details of the actual implementation of the network of PDE solvers including the state machines of all agents involved are provided in [17]. Here we outline the component agents and their functionality. The basic functionality of individual agents involved in a network of PDE solvers is presented in the SciAgents system [24]. Thus we were able to identify with relative ease the functions expected from each agent and write new strategies in Java. The actual design and implementation of the network of PDE solving agents took less than one month.

Three types of agents are involved: one `PDECoordinator` agent, several `PDESolvers` and `PDEMediator` agents. The `PDECoordinator` is responsible with the control of the entire application, a `PDEMediator` arbitrates between the two solvers sharing a boundary between two domains, and a `PDESolver` is a wrapper for the legacy application.

As pointed out earlier, it is rather difficult to install legacy software on a new system, and in this paper we assume that the software is already installed and the paths to executables on all systems are known. An installer agent would need to discover the actual configuration of the system, locate libraries, compilers, ensure that enough resources are available locally and so on. The operation of the network of agents is presented next.

A `PDECoordinator` agent is started by means of a GUI. Once started the agent reads and parses a problem description file and writes the information into its model. This input file contains

information about the number of solvers and mediators, the characteristics of the interfaces, the initial guesses on the interfaces, the relaxation methods and the names of the machines that will be used to solve the global problem. The next step is the creation and the configuration of `PDESolver` and `PDEMediator` agents. Since the agents are alive, the `PDECoordinator` uses their addresses to set up the communication among them. Then the coordinator waits for messages from the mediators, regarding the status of the convergence to the solution of the problem, or from the user. The messages from the user are to change the values of specific variables of the input file, such as the convergence tolerance, or to force the execution to stop.

The `PDESolver` agent is created, configured and started by the `PDECoordinator`. Its model contains addresses of the legacy programs used to solve the problem locally, paths of the input/output files, addresses of the visualization programs, etc. In the first state, the `PDESolver` starts-up the `Pelltool` which compiles the `.e` file that describes the local PDE problem, and creates the executable that will be used later on by the `ExecuteTool`. These tools and file designations are all part of the **PELLPACK** system [25]. In the next state, the solver extracts the points on the interfaces from the file that contains the mesh/grid points, and writes them into a file. Then the solver notifies the mediators that the files are ready. The `PDESolver` agent remains idle until being notified by the mediators that the list with all the points and their initial guesses are stored in a file at a specific location. Then the solver uses these files of points and initial guesses to run the `ExecuteTool` to solve the problem. When the execution is finished, the solver sends a message to the mediators that new values are computed, and then waits for their response. Depending on the message from the mediators, the solver will solve the problem again, remain idle waiting for the other solver to reach convergence, or plot the local solution. The `PDECoordinator` is able to terminate the `PDESolver` by sending an appropriate message.

The mediator agent, `PDEMediator`, is created and configured by the coordinator agent. The mediator agent has a complete description of the interface, the relaxation method used, the solvers to collaborate with, the location of the input/output files, the location of the legacy programs, the tolerance used to decide convergence, and the initial guess function. This information is provided by the coordinator agent. After being started, the mediator waits for the boundary points from the two neighboring solvers. In the next state, the mediator combines the two point lists and then uses the initial guess to compute values at these points. Afterwards, the mediator sends a messages to the two solvers that the files with the points and their values are ready. The mediator remains idle, waiting for new values from the two solvers. When it receives new values it moves to the next state, reads the new data and compares them with current data. Then the mediator agent uses the relaxation method to calculate the new boundary conditions. If convergence is reached on this interface then the mediator sends messages to the solvers and informs the `PDECoordinator` about the local convergence so it will be able to decide on global convergence. A message from the coordinator will cause the mediator to (i) finish, in the case of global convergence, or (ii) wait for new data from the two solvers. In the latter case the procedure is repeated until convergence.

6. AN APPLICATION: THE HELMHOLTZ MODEL IN UNDERWATER ACOUSTICS

In this section we present one of the important problems solvable with a network of agent controlled PDE solvers. A standard approach in modeling propagation and scattering of acoustic waves in the ocean is based on the use of the Helmholtz equation, or its approximations, coupled by appropriate

boundary and interface conditions; see, for example, [26,27]. To be specific we consider the problem of modeling the propagation of acoustic waves in the area defined by the lines ϵ (sea level), γ (bottom of the sea) and $\delta \equiv \partial\Omega_3$ (the boundary of an obstacle that scatters incident waves) as those given on the left of Figure 3. The line γ represents the bottom and plays the role of the interface between water (the domain marked as Ω_1) and soil (domain Ω_2). For simplicity we assume that ϵ is a straight line and that Ω_2 consists of homogeneous isotropic soil. The propagation of waves in the area can be effectively modeled by the following PDE problem:

$$\Delta u(x) + k^2 u(x) = f(x), \quad x \in \Omega \tag{1}$$

$$B_\gamma u(x) = g_\gamma(x), \quad x \in \gamma \quad \text{and} \quad B_\delta u(x) = g_\delta(x), \quad x \in \delta \tag{2}$$

and

$$\frac{\partial u(x)}{\partial r} - iku(x) = o(r^{-1/2}) \quad \text{uniformly as } r = |x| \to \infty \tag{3}$$

Equation (1) is known as the Helmholtz (or reduced wave) equation, where $k = 2\pi/\lambda \in \mathbb{C}$ is called the number of a wave of frequency λ. k^2 is complex in the case of damped waves. Depending on the physical properties of the obstacle and the soil the interface/boundary conditions (2) can be of Dirichlet, Neumann or Robin type. To model the difference in the physical acoustic properties of soil and water the coefficients of these boundary conditions might be discontinuous implying a jump on the normal to γ derivative of the solution. Assuming that the obstacle is not penetrable by the waves there is no such jump on δ. The additional condition (3) is required to ensure mathematically the uniqueness of the solution of the above Helmholtz problem. Relation (3) is known as the Sommerfeld radiation condition and physically specifies an outgoing wave.

One usually restricts the unbounded domain of interest into a rectangular domain $\Omega \equiv$ABCD shown in the figure by introducing lines BC, CD and DA which are artificial boundaries that truncate the physical unbounded domain. The selection of optimum (with respect to modeling capabilities) conditions on these artificial boundaries is an open problem. Several choices are available ranging from simple Robin-type conditions to powerful non-local ones realized by appropriate integral equations on the artificial boundary lines.

In a typical application the computational domain is large, ranging from a few to several hundred miles. To ensure that the associated discrete model captures all physical phenomena this domain needs to be discretized using a relatively fine grid. Typical mesh sizes are no more than one tenth of the wavelength, which leads to very large linear systems of complex algebraic equations. Note that this PDE lacks certain important properties (symmetricity and positivity) and involves boundary conditions of mixed type. Therefore fast Poisson-type solvers like FFT are not readily applicable. Thus solving these linear systems in the case of high frequency waves can easily exhaust the capabilities of any modern uni-processor computer system. Therefore efficient parallel solvers are necessary for this problem and already significant research effort has been devoted to searching for them. Domain decomposition methods have the potential to provide such solvers. Nevertheless it was realized quite early that some of the most popular domain decomposition methods could not be used due to either divergence or slow convergence. One has to be careful in selecting appropriate conditions on the interface along subdomain lines so that the resulting local problems are well posed and have reasonable condition numbers. In addition, they should guarantee that waves can easily propagate from one subdomain to the other even in the case that they oscillate rapidly on an interface. The

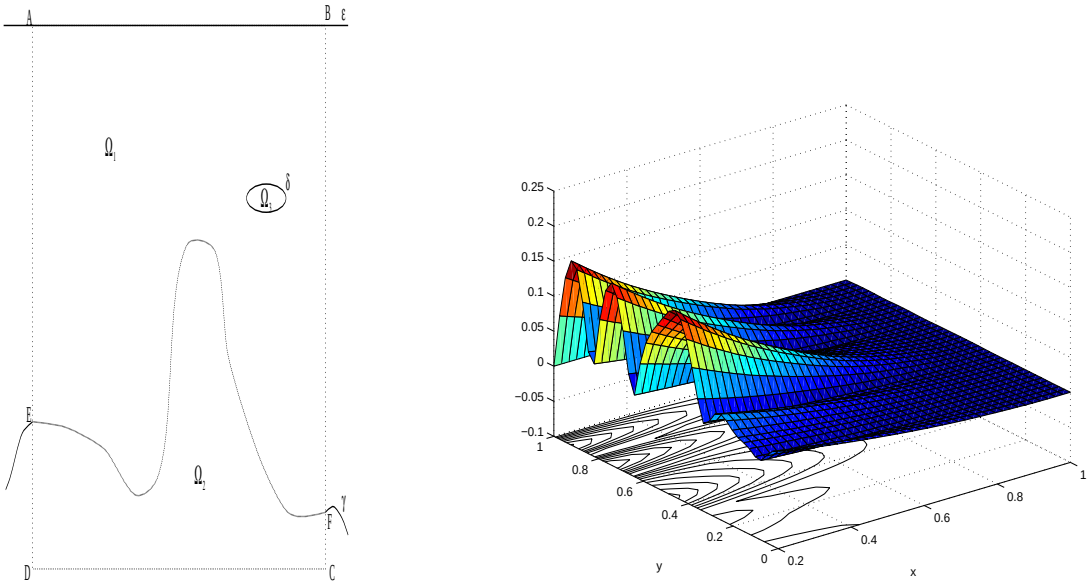


Figure 3. Plots of a cross-section of the 3-D acoustic domain (on the left) and of the amplitude of the computed solution for a 2-D simplified model with $k = 20$ (on the right).

possibility of imposing non-local conditions on the artificial boundary or interface lines simply make the domain decomposition approach more complicated and the analysis more challenging. Theoretical and experimental domain decomposition studies for the Helmholtz model have been published with an increasing frequency lately.

We use this application to illustrate the use of agent based implementation [28] of domain decomposition and interface relaxation. This problems is described in Figure 3 and has a known analytic solution. The numeric PDE solving method used to compute the acoustic pressure is a standard Galerkin/finite element method [29]. This model assumes that a point source is placed at $(0, y_s)$, that the sea bottom is horizontal, that there are no obstacles in the problem domain, and that the second order absorbing boundary condition apply on the side BC seen in Figure 3. This is a well-posed Helmholtz problem and, as seen from the two views plotted, there is considerable complexity in the solution. The computed solution agrees with the analytic solution very well.

Note that PDE models similar to the one described above are frequently used in general time-harmonic acoustic and electro-magnetic wave scattering problems. Most of the domain decomposition methods we consider can be readily used for such problems.

7. CONCLUSIONS

Software agents may provide an answer to the increased complexity of the software systems expected to intelligently anticipate and adapt to the needs of dynamically distributed applications. In this paper we discuss combining agents with legacy applications to solve data intensive problems.

Data parallelism is a common approach to reduce the computing time and to improve the quality of the solution for data-intensive applications. Often the algorithm for processing each data segment is rather complex, and the effort to partition the data, to determine the optimal number of data segments, and to combine the partial results, to adapt to a specific computing environment and to user requirements must be delegated to another program. Mixing control and management functions with the computational algorithm leads in such cases to brittle and complex software.

We discuss the complex choices faced by a user in the context of a parallel algorithm for determining the parallel orientation of virus particles in electron microscopy. We argue that an agent with inference abilities provides an optimal solution to the problem. Such an agent is developed as part of Virtual Laboratory for Computational Biology.

In some cases one can achieve parallelism using sequential legacy codes without the need to modify them at all. Whenever we can apply a divide-and-conquer methodology based upon the partitions of the data into sub-domains, solve the sub-problems independently in each sub-domain, and then resolve with ease the eventual conflicts between the individual workers, we have an appealing alternative to code parallelization. The agents coordinate the execution and mediate the conflicts, as in the network of PDE solvers discussed in Section 5 of the paper. The main advantage of the solution we propose is a drastic reduction of the development time from several months to a few weeks.

ACKNOWLEDGEMENTS

The work reported in this paper was partially supported by a grant from the National Science Foundation, MCB-9527131, by the Scalable I/O Initiative, by a contract from the Department of Energy ASCI Academic Strategic Alliance Program, LG 6982, and by a grant from the Intel Corporation.

REFERENCES

1. Franklin S, Graesser A. Is it an agent, or just a program?. *Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages*. Springer Verlag, 1996.
2. Bradshaw JM. An introduction to software agents. *Software Agents*. Bradshaw JM (ed.). MIT Press, 1997; 3–46.
3. Bölöni L, Marinescu DC. An object-oriented framework for building collaborative network agents. *Intelligent Systems and Interfaces*. Teodorescu NH, Mlynek D, Kandel A, Hoffmann H-J (eds.). Kluwer Publishing House, 2000; 31–65.
4. Marinescu DC. An agent-based design for problem solving environment, *Proceedings of Workshop on Parallel/High Performance Scientific Computing, POOSC'99*, 1999; 24–34.
5. Bölöni L, Hao R, Jun KK, Marinescu DC. Structural biology metaphors applied to the design of a distributed object system. *Proceedings Second Workshop on Bio-Inspired Solutions to Parallel Processing Problems* (*LNCS*, vol. 1586). Springer Verlag, 1999; 275–283.
6. Marinescu DC, Bölöni L. Biological metaphors in the design of complex software system. *Journal of Future Generation Computing Systems*. Elsevier, 2000 (in press).
7. Genesereth MR. An agent-based framework for interoperability. *Software Agents*. Bradshaw JM (ed.). MIT Press, 1997; 317–345.
8. Lynch RE, Marinescu DC, Lin H, Baker TS. Parallel algorithms for 3D reconstruction of asymmetric objects from electron micrographs. *Proceedings 13th International Parallel Processing Symposium*. IEEE Press, 1999; 632–637.
9. Quarteroni A, Pasquarelli F, Valli A. Heterogeneous domain decomposition: principles, algorithms, applications. *Proceedings Fifth International Symposium Domain Decomposition Methods PDEs*. Keyes D *et al.* (eds.). SIAM: Philadelphia, 1992.
10. Lions DL. On the schwarz alternating method III: a variant for non-overlapping domains. *Domain Decomposition Methods for PDEs*. Chan T *et al.* (eds.). SIAM: Philadelphia, 1990.
11. Rice JR, Tsompanopoulou P, Vavalis EA. Interface relaxation methods for elliptic differential equations. *Appl. Numer. Math.* 2000; 219–245.

12. Mu M. Solving composite problems with interface relaxation. *SIAM J. Sci. Comp*. 1999; **20**:1394–1416.
13. Rice JR, Vavalis EA, Yang D. Analysis of a nonoverlapping domain decomposition method for elliptic partial differential equations. *J. Computational Appl. Math.* 1997; **87**:11–19.
14. Palacz K, Marinescu DC. An agent-based workflow management system. *Proceedings AAAI Spring Symposium Workshop*. AAAI Press, 2000; 119–127.
15. Jun KK, Bölöni L, Yau DKY, Marinescu DC. Intelligent QoS support for an adaptive video service. *Proceedings International Resource Management Association*. IRMA, 2000; 1096–1099.
16. Jun KK, Bölöni L, Palacz K, Marinescu DC. Agent-based resource discovery. *Proceedings Heterogeneous Computing Workshop, HCW 2000*, vol. 1. IEEE Press, 2000; 43–52.
17. Tsompanopoulou P, Bölöni L, Marinescu DC, Rice JR. The design of software agents for a network of PDE solvers. *Proceedings Workshop on Agent Technologies for High Performance Computing, at Agents 99*, 1999; 57–68.
18. Finn T, Labrou Y, Mayfield J. KQML as an agent communication language. *Software Agents*. Bradshaw JM (ed.). MIT Press, 1997; 291–316.
19. Hao R, Jun KK, Marinescu DC. Bond system security and access control model. *Proceedings IASTED Conference on Parallel and Distributed Computing and Networks*, 1998; 520–524.
20. Hao R, Bölöni L, Jun KK, Marinescu DC. An aspect-oriented approach to distributed object security. *Proceedings of the 4th IEEE Symposium on Computers and Communications*. IEEE Press, 1999; 23–31.
21. Schneier B. *Applied Cryptography*. John Willey & Sons, 1996.
22. Bölöni L, Marinescu DC. Agent surgery: the case for mutable agents. *Proceedings Workshop on Biologically Inspired Solutions to Parallel Processing Problems* (*LNCS*). Springer Verlag, 2000 (in press).
23. Bölöni L, Marinescu DC. A component agent model—from theory to implementation. *Proceedings International Symposium, From Agent Theory to Agent Implementation.* IEEE Press, 2000 (in press).
24. Drashansky TT. An agent based approach to building multidisciplinary problem solving environments. *PhD Thesis*, Purdue University, 1996.
25. Houstis EN, Rice JR Weerawarana S, Catlin A, Gaitatzes M, Papachiou P, Wang K. PELLPACK: a problem solving environment for PDE based applications on multicomputer platforms. *ACM Trans. Math. Software* 1998; **24**:30–73.
26. Bayliss A, Goldstein C, Turkel E. The numerical solution of the Helmholtz equation for wave propagation problems in underwater acoustics. *Comp. Math. Applic.* 1985; **11**:655–665.
27. Bayliss A, Goldstein C, Turkel E. On accuracy conditions for the numerical computation of waves. *J. Comp. Phys.* 1985; **59**:396–404.
28. Rice JR, Vavalis E. Collaborative agents for modeling air pollution. *Syst. Anal. Modeling Simulation* 1998; **32**:93–101.
29. Dougalis V, Kampanis N. Tsompanopoulou P, Vavalis E. Linear systems solvers for finite element discretizations of the Helmholtz equation. *Proceedings of the 3rd European Conference on Underwater Acoustics*, 1996; 279–284.

