The Design of Software Agents for a Network of PDE Solvers

Panagiota Tsompanopoulou, Ladislau Bölöni, Dan C. Marinescu, and John R. Rice
Computer Sciences Department
Purdue University, West Lafayette, In, 47907, USA
(giwta, boloni, dcm, jrr@cs.purdue.edu)

March 23, 1999

Abstract

The simulation of complex physical systems often involves solving a large system of partial differential equations (PDEs). We discuss how solving such a system of PDEs can be done by splitting a domain into several sub-domains and creating of a network of PDE solvers. In such a network several agents are used to: (a) control the execution of individual solvers on each sub-domain, (b) mediate between adjacent sub-domains, and (c) coordinate the execution of the ensemble. This paper presents the advantages of agent-based PDE solving and describes the implementation of a network of Partial Differential Equations, PDE, Solvers using **Bond** middleware for agent-based computing.

1 Overview

Consider the simulation of a complex physical device such as the engine shown in Figure 1. It has several physical phenomena (combustion, waterflow, structure,...) in a complex geometry. One could attack the simulation as follows: (1) separate the device into parts that have the same physical phenomena, (2) further separate each of these parts into pieces that have simple geometry. Then the simulation requires that partial differential equations (PDEs) be solved on each of the resulting pieces and that the solutions on the surfaces of the pieces agree (as dictated by the physics) with those of the piece's neighbors. Thus one has a large set of PDE problems along with interface conditions. One then assigns a PDE solver to each piece and connects them together so they can collaborate in solving the overall simulation problem. There is a mathematical method called *interface relaxation* which can be used to iteratively solve the separate PDE problems and relax the interface conditions collaboratively to achieve this overall solution.

Now visualize these PDE solvers made into software *solver agents* which collaborate in the simulation. Next create a set of *mediator agents*, one for each interface between two pieces, which carry out the interface relaxation process which is defined locally on each interface. We assume that the network of agents then has the mathematical capability to solve the PDEs of the simulations, i.e., these software agents have effective numerical methods which converge for the global problem at hand. Whether this assumption is true or not is a numerical analysis problem not discussed in this paper. These agents could compute the solution by a simple minded, synchronized iteration with all the agents on one machine or with each agent running on its own machine.

Once this network of collaborating agents is created, there are several other issues that can be addressed which exploit the properties of agent-based computing.

1. *Asynchronous Computing.* The computations proceed locally until the changes fall below a certain tolerance level. There is no need for all the agents to keep computing if only a

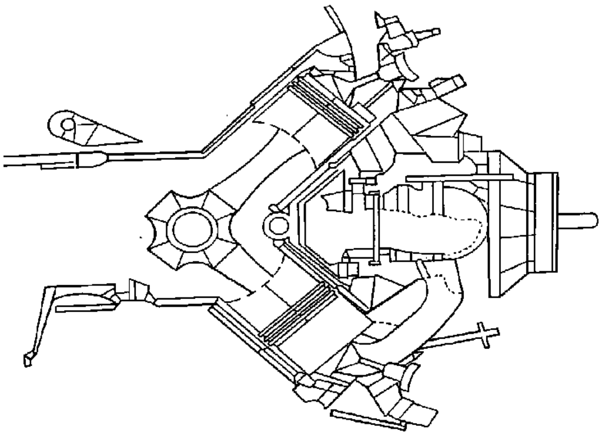

Figure 1: An automobile engine system consisting of many different parts.

few have failed to reach the target tolerance. Policies or goals can be given to the agents to guide their behavior during the collaboration to achieve this tolerance.

2. *Load Balancing.* Normally there are several (many) processors used because the computations are very large. However, there may be more agents than processors so decisions have to be made in their assignments to processors. The agents can time themselves for one PDE solution or one relaxation and their policies be established to associate processors and agents. For example, a PDE solver agent usually takes hundreds or thousands times as long for an iteration than a mediator agent and the difference in the PDE solving time can easily be a factor of 5 to 20. The agents might be mobile so they can truly migrate from machine to machine or load balancing might be accomplished by distributing PDE sub-problems among PDE solving servers. In any case, policies and goals can be given to the agents to guide their assignment to processors.

3. *Software Selection.* The PDE problem on a single piece can probably be solved by several, even hundreds, of different methods; methods for which software exists in repositories or which is used by network servers. Again, policies and goals can be given to agents to guide their selection of software.

In this paper we provide a software design for agents to implement the above collaborating PDE solvers network. This design is presented using the *Bond middleware* for the agents and the *PELLPACK problem solving environment* for the PDE solver agents. We begin with an example of a simple two-domain PDE problem as in Figure 2.

Here, the domain is split into two sub-domains and there is only one interface. Therefore there are two `PDESolver` agents created from PELLPACK [8], whose `solverid` are 1 and 2 respectively. The `mediatorid` for the `PDEMediator` agent is 1. The boundary of each sub-domain consists of pieces, identified by `pieceid`. So for example, the interface corresponds to the boundary piece with `pieceid` 4 for the `PDESolver` with `solverid` 1 and to the boundary piece with *pieceid* 3 for the PDESolver with *solverid* 2. Each local problem, i.e. the PDE problem in each sub-domain, is described in a PELLPACK file with a `.e` suffix. The preprocessing of this file, is done by Pelltool, and gives the corresponding `.f` file (a Fortran file that contains the main program of the local problem). Compiling the Fortran file one obtains an executable and a `.script` file that is used by the `ExecuteTool`. The discretization points on the local sub-domain are contained in a file, with `.mesh` or `.grid` suffix, when Finite Element Methods (FEM) or Finite Difference Methods (FDM) are used correspondingly.

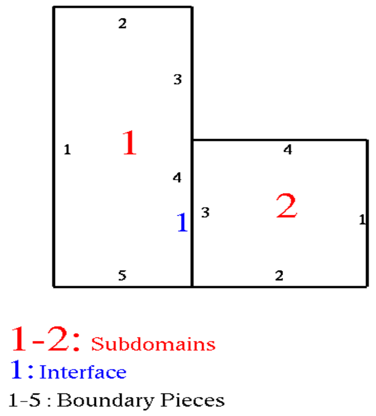

Figure 2: A simple PDE problem, whose domain is split into two sub-domains.

2 Bond Agents

Bond, is a Java-based, object-oriented middleware for network computing using KQML [4] as a communication language. Network computing is a paradigm that emphasizes the use of network resources, computing resources distributed across the network, over local resources. Network resources are hosts, communication links, programs, services and data. Middleware is a software layer that allows developers to mold systems tailored to specific needs from components and develop new components based upon existing ones.

Bond (`http://bond.cs.purdue.edu`) consists of a core package, providing the basic functionality of a distributed-object system and several frameworks for monitoring, security, [7], [6] and for software agents, [1], [2], [5], see Figure 3.

A Bond agent is an object with the following components [1], [2]:

- The **model of the world** is 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. There is no restriction of the format of this information: it can be a knowledge base or 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 function on the model and a distance function on the model. The boolean function shows if the agent accomplished its goal or not. The agenda acts as a termination condition for the agents, except for the agents marked as having a *continuous agenda* where their goal is to maintain the agenda as being satisfied. 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 programs. 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 of 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

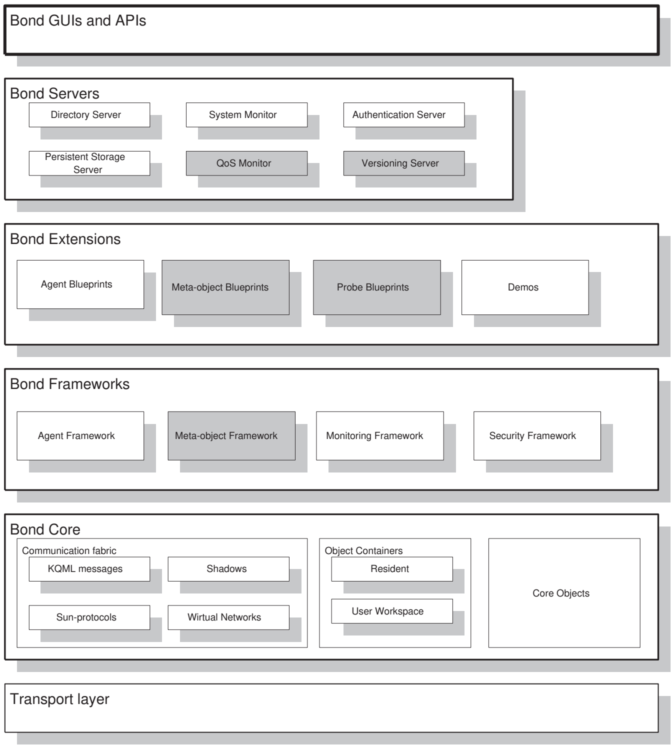

Figure 3: The architecture of the Bond middleware.

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.

The agent framework can be programmed at two levels. At the expert level, the developer can define new strategies and agendas by programming them directly in Java. At the *blueprint* level, the user can create new agents using the blueprint language of the Bond agent framework. With the blueprint a user can:

- specify the finite state machine of the new agent
- assign strategies from a strategy database to the states
- assign the agenda of the agent from the strategy database
- assemble new strategies following an *aspect oriented programming* approach using the strategy composition objects.
- assemble new agendas by applying boolean composition on predefined agendas
- create the control subprotocol of the agent
- initialize the variables of the model.

However a blueprint description cannot define new strategies or agendas which can not described as boolean conditions on the model.

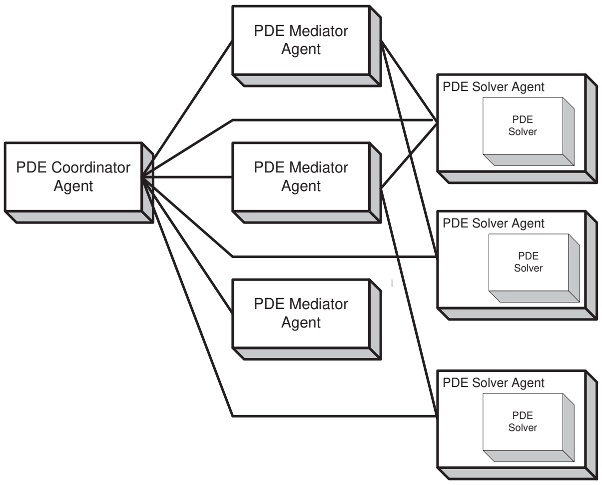

Figure 4: The network of agents used for distributed PDE solvers

3 PDE Network Agents

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.

The basic functionality of individual agents involved in a network of PDE solvers was the object of a one year old PhD dissertation. Thus we were able to identify with relative ease the strategies needed and then write them in Java.

The operation of the network of agents shown in Figure 4 is presented next. The `PDECoordinator` agent is started by means of a GUI. The agent reads and parses a problem description file and writes the information into its model. Then it starts up the other agents using the information provided in the configuration section of its model to decide where to start each agent. An agent is started by providing the blueprint for the new agent to the agent factory on the local system. Once all agents are started, the `PDECoordinator` agent updates the models of the other agents based upon the information stored in its own model. For example, a mediator agent learns the location of the two solver agents it is expected to mediate, a solver agent learns the domain it is expected to work on, as well as the identity of the mediators to work with. If necessary, data files are migrated to each solver agent.

Now all agents are ready to proceed and the `PDESolver` agents start up various components of the PDE Solver environment. First they start the `PELLPACK` tool. Once a solver finishes its computation it starts interractions with the mediators. Further understanding of the network of agents requires a more detailed presentation of the states and the strategies for the three types of agents, `PDESolver, PDEMediator`, and `PDECoordinator`.

The states and the strategies for the **PDESolver** agent are shown in Figure 5 and a brief description of the actions expected in each state follows.

- *Agent Started.* The agent is started by a message from the `PDECoordinator`. It replies acknowledging that is up and ready to work. By this time its model contains: (a) `solverid`, (b) the local working directory, (c) the location of the executables (i.e. `Pelltool, ExecuteTool, OutputTool`), (d) the location of the input files (the `.e` and `.mesh` or `.grid` files) on a remote host, (e) the location for the files that will keep error/warning messages, (f) the `PDEMediators` the `PDESolver` is connected with, and (g) mapping of boundaries with `PDEMediators`.

- *SetUp.* In this state the PDESolver fills its model with needed information (e.g. using paths already in model creates the full names of executables that will be executed in later states).
- *Start Pelltool.* In this state the agent starts-up the `Pelltool` which has as input the `.e` file. The `Pelltool` preprocess the `.e` file, creates the corresponding `.f` and then compiles it to create an executable and a script file that the `ExecuteTool` will use.
- *Supervise Pelltool.* After starting the `Pelltool`, the `PDEsolver` supervises its execution. When Pelltool finishes successfully it moves to next state.
- *Extract Mesh Points.* The agent prepares data files with the boundary points on each interface. It reads the `.e` file to find the name of the `.mesh` or `.grid` file which contains, among other information, all the discretization points of that particular sub-domain. Processes the `.mesh/.grid` file to extract the mesh points and creates an `bpoints.mesh.solverid` (or `fd-points.solverid.fdp` in the grid case) files. From the last file it extracts the mesh/grid boundary points for each particular boundary piece and creates separate files for each one of them named `boundary.points.solverid.pieceid`. After a successful parsing of the files, it informs the `PDEMediator` agents, that the boundary points from this particular sub-domain are ready.
- *Supervise Extraction.* Supervise the execution of the task described above, done by an independent program.
- *Wait for Boundary Points.* Expect a message from `PDEMediator` notifying that files named `interesting.points.mediatorid` with boundary points are ready.
- *Combine Boundary Points.* Since all files with the interesting boundary points are created, the agent parses them to combine boundary points and create a new file named `bpoints-in.solverid.bp`. If the file parsing is successful, it moves to next state.
- *Supervise CBP.* Supervise the execution of the previous step and proceeds to next state in case of success.
- *Wait for Boundary Values.* Message from the collaborating `PDEMediator` agents are expected notifying that files `boundary.values.mediatorid.solverid` with boundary values are ready.
- *Combine Boundary Values.* Read the files `boundary.values.`*mediatorid.solverid* with the boundary values, prepared by the collaboratoring `PDEMediator`. Combine them and create the file `bvalues-in`*solverid*`.bp`.
- *Supervise CBV.* Wait for the above execution to finish, then move to the next state.
- *Start ExecuteTool.* The `ExecuteTool` is started to solve the problem in the current sub-domain. Use the executable, the script file, the combined boundary points, and boundary values files obtained earlier. The id of the process is kept for the next state.
- *Supervize Execution.* Using the process id, the agent supervises the execution and waits for the termination code of the running process. If the `ExecuteTool` finishes with success the agent moves to next state. If the execution fails, the sends an error message and exits.
- *Iteration Terminated.* The agent sends a message to the collaborating `PDEMediator agents` that iteration is done and the output files are ready to be processed. If the message is sent properly, then it moves to next state.

- *Wait for Convergence.* Wait for a message from the collaborating `PDEMediator` agents signaling that they have finished the processing of the interface. The message contains information about local convergence. If the message is `no local convergence` the agent goes back to *Wait for Boundary Values.* If the message is `local convergence` then the agent waits for a message from the `PDECoordinator` regarding the global convergence. If the message is `global convergence` it moves to state *Plot Solution.* If the message is `noconvergence` the `PDESolver` remains idle until new messages from mediators about local convergence arrive.

- *Plot.* Since global convergence was achived, the `OutputTool` is called to plot the local solution.

- *Supervise Plot.* Supervise the execution of `OutputTool`. When finished go to *Exit.*

- *Error.* The agent reaches this state in case of an error. It prints out a message describing the error and then go to *Exit.*

- *Exit.* The agent is sent a termination message by the`PDECoordinator`, the problem was solved successfully.

The states of the `PDEMediator` agent are described below and shown in Figure 6.

- *Agent started.* The agent is stated by a message from the `PDECoordinator`. After that it sends a message to `PDECoordinator` acknowledging that is up and ready to work. By this time the agent has in its model information like: (a) *mediatorid* number as defined by PDE problem, (b) the local working directory, (c) the location of the input files on a remote host, (d) the location for the files that will keep error/warning messages, (e) the *solverid* of the two `PDESolvers` the particular `PDEMediator` is connected with, (f) the relaxation formula and (g) the *pieceid* of the boundary pieces of the neighbor `PDESolvers`, that work together with the `PDEMediator`. This information is written into ythe model by the `PDECoordinator`, just before sending the start-up message.

- *SetUp.* The `PDEMediator` fills-in its model with needed information (e.g. using paths already in model creates the full names of executables that will be executed in later states).

- *Wait for Boundary Points.* The `PDEMediator` waits for the `PDESolver` agents working with it to create several files files `boundary.points.solverid.pieceid`. Upon receiving this message it moves to next state.

- *Initialize Boundary Values.* Collects the two boundary points files (from the two collaborating `PDESolvers`) and combines them into one file named `interesting.points.mediatorid`. Calculate the normal vector and initialize the boundary conditions on the interesting points, and write the values in files named `boundary.values.mediatorid.solverid`.

- *Supervise Initialization.*

- *Notify Solvers.* Since all files are ready, sends messages to the two collaborating `PDESolver` agents to start solveing the problem locally.

- *Wait for Solvers.* Wait for the `PDESolvers` to finish the solution of the local problem.

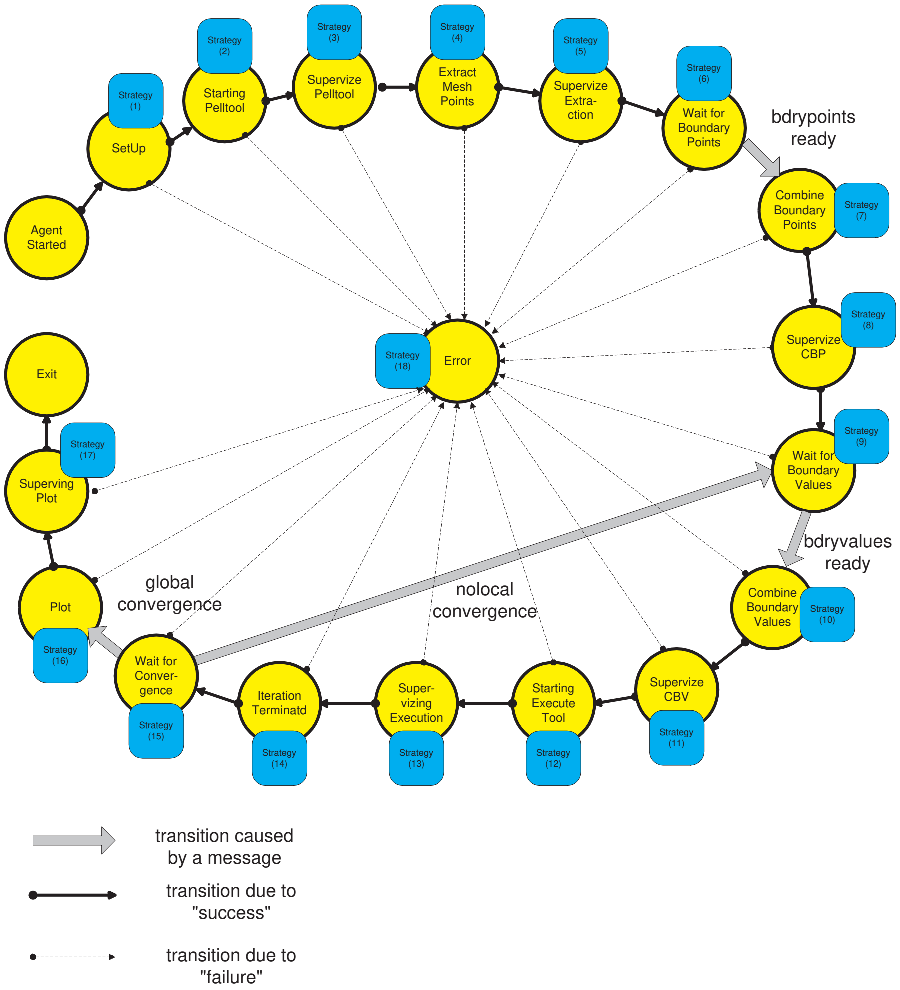

Figure 5: The state transition diagram of the PDESolver Agent, `PDESolver`. State transitions are caused by: (a) external messages, (b) the success/failure to attain the goal of the strategy in a given state.

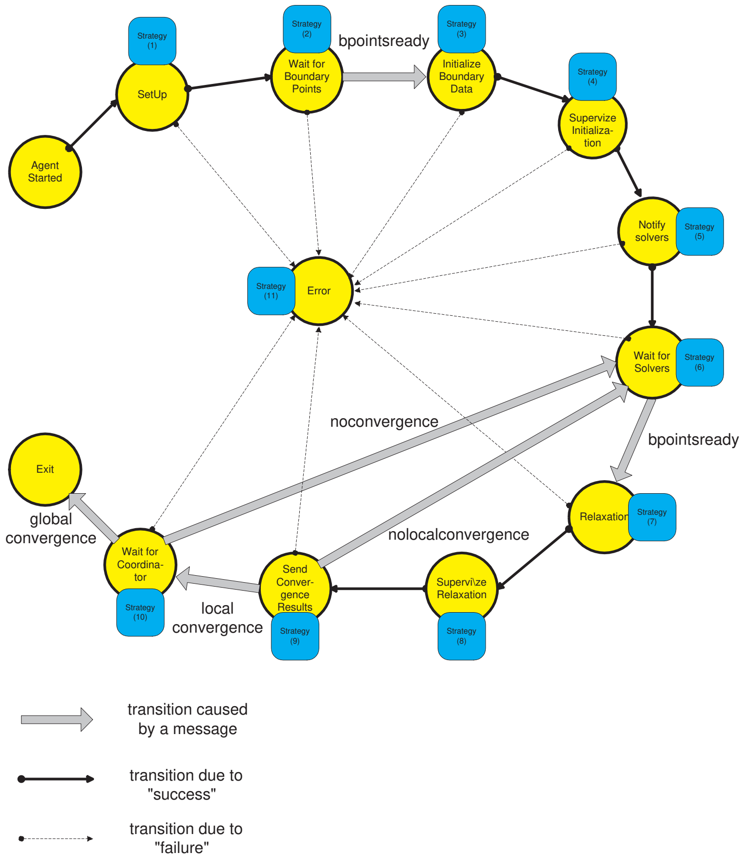

Figure 6: The state transition diagram of the PDEMediator Agent, `PDEMediator`. State transitions are caused by: (a) external messages, (b) the success/failure to attain the goal of the strategy in a given state.

- *Relaxation*. Since new solution is computed, the agent needs to read the output files (named `bvalues-out`*solverid*`.bv`) and calculate the difference between two successive iterations on its interface. Also the new boundary conditions are calculated by assigning the relaxation formula and are stored in the `bvalues-in`*solverid*`.bp` files.

- *Supervise Relaxation*. Here the agent supervises the above execution. When is done it proceed to the next state.

- *Send Convergence Results*. If the difference calculated in *Relaxation* is small enough, local convergence was obtained. Otherwise there is no local convergence. In each case the agent sends the right message to the collaborating `PDESolvers` and to the `PDECoordinator`. In case of local convergence it moves to state *Wait for Coordinator*, while in case of no local convergence in moves to state *Wait for Solvers*.

- *Wait for Coordinator*. Wait for instruction from the `PDECoordinator`. If global convergence is archived the agents moves to *Exit*, if no convergence it moves to *Wait for solvers*.

- *Error*. Here the agent prints out a message concerning the error and moves to state *Exit*.

- *Exit*. The agent terminates following a message from the `PDECoordinator`.

The state transition diagram of the `PDECoordinator` agent is similar with those in Figures 5 and 6. A brief description of these states and the corresponding strategies follows:

- *Agent started.* The agent is started using a GUI.
- *Read Input Filename.* The agent reads the name of the file that contains the input data for the PDE problem and the agent configuration.
- *Parse Input File* The agent parses the input file and fills its model with the information that later will share with other agents.

  The input file of the coordinator agent contains the following information: (a) the number of `PDESolvers`, and (b) the number of `PDEMediators`. Then for each `PDEMediator` there is (c) a left `PDESolver` id, (d) the boundary pieceid in the left sub-domain, that corresponds to the interface the particular `PDEMediator` handles, (e) the relaxation formula used to update the boundary conditions at each iteration, (f) the right `PDESolver id`, (g) the boundary `pieceid` of the right sub-domain, that corresponds to the interface that the particular `PDEMediator` handles, (h) the relaxation formula used to update the boundary conditions at each iteration, (i) the tolerance for convergence, (j) `x` and `y` coordinates of the upper point of the interface line, (k) an initial guess of the unknown (and wanted) function on that point, (l)`x` and `y` coordinates of the lower point of the interface line, (m) an initial guess of the unknown function on that point. (n) Machine name for the user interface and the display of all the tools (i.e. `PellTool, ExecuteTool` and `OutputTool`, (o) Information about the machine for the `PDECoordinator` agent, (p) Information about the machines that will host the `PDESolver` agents and (q) information about the machines that will be assigned the `PDEMediator` agents.
- *Create Solver Agents.* Send the blueprint for a PDESolver to the Bond residents on the hosts assigned to run `PDESolver` agents.
- *Create Mediator Agents.* Send the blueprint for a PDEMediator to the Bond residents on the hosts assigned to run `PDEMediator` agents.
- *Solver's Configuration.* Write into the model of the PDESolver agents information that they need.
- *Mediator's Configuration.* Configure PDEMediators.
- *Communication's Configuration.* The coordinator uses the *shadows* of PDESolvers and PDEMediators to inform each other about their addresses.
- *Start Solver Agents.* Send the "start" command to all `PDESolver` agents.
- *Start Mediator agents.* Send the "start" command to all `PDEMediator` agents.
- *Wait for Messages.* Wait for messages from mediator agents or from the user. A message from the user, may contain the change of the value of some parameters, or it may force the coordinator to terminate the execution. In the first case, the agent sends the proper messages to those agents that needs to know about this change. In the second case, if the message is *pause* the agent moves to *Exit*, if it is *quit* it moves to *CleanUp*. In both cases it lets every agent know that is time to stop its processes and then moves to *Exit*.

  If the message comes from a mediator agent we distinguish the following cases. First, the mediator did not achieve local convergence. This implies that no global convergence is possible, so the mediator sends messages to all agents to work on the next iteration. If the message indicates that the mediator achived local convergence, then the coordinator waits for the other mediators to report their convergence status. Then it decides if global convergence can be archived. Id so the coordinator sends messages to all agents to stop their computations and moves to *Exit*.

- *CleanUp*. The Coordinator cleans the local directories of unwanted files. If successful it moves to *Exit*.

- *Error*. The agent reaches this state only if an error occurred in any of the previous states. It prints a message indicating the cause of the problem and moves to *Exit*.

- *Exit*. The coordinator sends the termination messages to all agents.

4 Conclusions

Creating a network of PDE solvers requires the development of a group of software agents with relatively low level of inference capabilities. The agents use well-defined communication patterns to interact with one another.

Bond provides a convenient framework to construct agents as finite-state machines. The effort to develop the agents described in this paper was relatively modest and did not require a deep understanding of multi-threading and various communication protocols.

Several generic strategies available in strategy databases were used including the start-up of a group of agents, the start-up of a legacy application, synchronization of a group of agents, writing into the model of another agent.

We have experienced performance problems when the agents run on slow, out of date Unix systems. The Java Virtual machines running on old SPARC systems is simply too slow. We had to adjust the timeouts involved in various protocols to compensate for the large processing time.

Our agent framework has evolved since the original writing of this paper. We now support space agents consisting of multiple planes. Each plane implements a finite state machine and the planes communicate through shared-memory, the model of the agent. The PDE agent network is now being updates and tests are conducted with agents and PDE solvers running on a cluster of PCs running Solaris.

Acknowledgments

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] L. Bölöni and D.C. Marinescu *A Multi-Resolution Model for Software Agents* Technical Report, Department of Computer Science, Purdue University, 1999.

[2] L. Bölöni and D.C. Marinescu *An Object-Oriented Framework for Building Collaborative Network Agents*. Kluever Publishers, 1999 (to appear).

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

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

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

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

[7] 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).

[8] E.N. Houstis, J. R. Rice, S. Weerawarana, A. Catlin, M. Gaitatzes, P. Papachiou, and K. Wang, *PELLPACK: A Problem Solving Environment for PDE Based Applications on Multicomputer Platforms*, ACM Trans. Math. Software, 24, pp. 30-73, 1998.

[9] D. C. Marinescu, L. Bölöni, R. Hao, and K.K. Jun: *Structural Biolgy Metaphors Applied to the Design of a Distributed Object System*, Proc. Second Workshop on Bio-Inspired Solutions to Parallel Processing Problems, Springer Verlag, (1999), (in print).