Purdue University

Purdue e-Pubs

Department of Computer Science Technical Reports

Department of Computer Science

1996

Agent Based Network Systems for Multi- Physics Problems

Tzvetan Drashansky

John R. Rice
Purdue University, jrr@cs.purdue.edu

Report Number:
96-068

Drashansky, Tzvetan and Rice, John R., "Agent Based Network Systems for Multi- Physics Problems" (1996). *Department of Computer Science Technical Reports.* Paper 1322.
https://docs.lib.purdue.edu/cstech/1322

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries.
Please contact epubs@purdue.edu for additional information.

Agent Based Network Systems for Multi-Physics Problems

Tzvetan Drashansky and J.R. Rice
Computer Sciences Department
Purdue University
and
Anupam Joshi
University of Missouri

CSD-TR-96-068
November 1996

Agent Based Network Systems for Multi-Physics Problems

Tzvetan Drashansky and John Rice
Purdue University
and
Anupam Joshi
University of Missouri

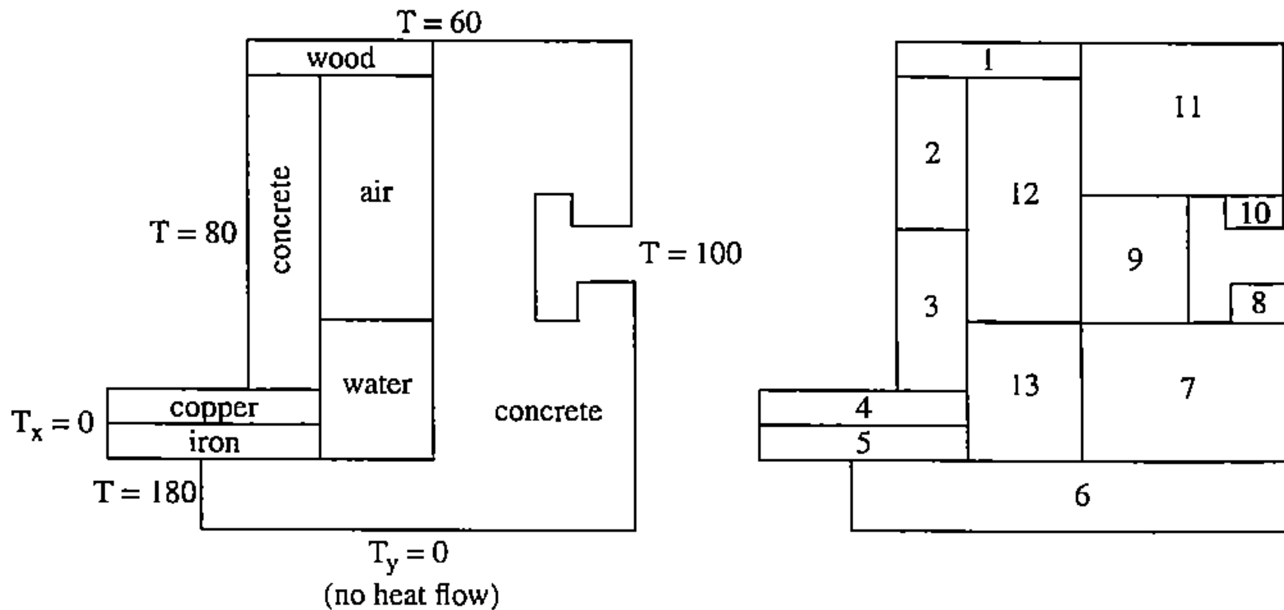
November 18, 1996

We consider a new approach to solving complex physical simulation problems, called *multi-physics problems*, which is naturally suited to agent based computing over a network of machines. Multi-physics problems are best visualized as one physical system with many components involving a variety of physical phenomena. e.g., heat flows, stresses and strains, fluid flows, mechanical motion, and gas flows. A lawn mower motor or an air conditioner involves all these phenomena. Each of these phenomena is modeled by differential equations or Newton's laws of motion. For simplicity we omit modeling mechanical motion and focus on the other phenomena. A simple example of a multi-physics problem is shown in Figure 1a where the objective is to compute the temperature distribution everywhere plus the air and water circulation flows. The internet, with every increasing capacity, opens up the possibility of combining software and hardware resources spread across it into a single system to solve very complex multi-physics problems. This approach can permit the heavy reuse of legacy software, allow heterogeneous distributed resources to be harnessed. and provide a dramatic reduction in software development time. This is the vision that guides this work.

The approach used for solving this problem is the *collaborating solvers* method (described in [9] for the case of partial differential equations – PDEs) which can be described in non-mathematical terms as follows. Assume that we have a collection of solvers that can accurately simulate any one of the physical phenomena on a component with a simple geometric shape given values around the boundary of the component with

- only one phenomenon per component
- a simple shape

An additional requirement, needed to achieve a satisfactory solution time for the problem, is that the amount of work to simulate the physics on each component not be too large. The multi-physics problem is solved when one has all the equations satisfied on the individual components and these



(a) The multi-physics problem. (b) The computational problem.

Figure 1: (a) The physical problem with seven components and three physical phenomena. The objective is to compute the temperature T everywhere. (b) The computational problem with 13 components. The concrete components were subdivided to simplify the geometry and to reduce the work for the simulation on some pieces.

solutions "match properly" on the interfaces between the components. The term match properly is defined by the physics if the interface is where the physics changes. For heat flow, for example, this means that temperature is the same on both sides of the interface and that the amount of heat flowing into one component is the same as the amount flowing out of the other. If the interface is artificial (introduced to make the geometry simple or the work smaller) then "match properly" is defined mathematically and means that the solutions join smoothly (have continuous values and derivatives).

This approach replaces the multi–physics problem by a set simulation problems which must be satisfied simultaneously along with a set of interface conditions. We have a set of solvers that can solve simple simulation problems defined on individual components and it is natural to define collaborative methods to solve the total problem. A number of methods based on relaxation formulas or *relaxers* for the interface conditions have been proposed for matching the interface conditions. The underlying mathematical problem can be analyzed only in extremely simple cases, but there is substantial experimental evidence that these relaxers are broadly applicable and can provide rapid convergence, see [1] and the references therein for further information. Figure 1b shows the computational problem that might result from the multi-physics problem of Figure 1a.

We have developed the *SciAgents* [2] system which implements this approach. We note two things first. Multi-physics problems can require enormous processing power, giving the component

calculations to different machines provides this naturally. Further, the computations on different components can be made by autonomous and loosely coupled software systems so an agent based approach is natural. The iterations on components are loosely synchronized (even the number of iterations made on each can – and does – vary somewhat). Other factors influencing the design of SciAgents are given below.

The Distributed Problem Solving Process. Each of the simple components is associated with a *solver agent* which includes a software module to simulate the physics on this component. Each of the interfaces between two components is associated with a *mediator agent* which includes a software module to implement the relaxers used. There are usually more mediators than solvers (there are 13 solvers and 21 mediators for the situation in Figure 1b). There is also a global control agent which sets goals for the agents (e.g., obtain three significant digits of accuracy in the solvers). The computation is described roughly as follows:

- Compute an initial estimate of the solution on all interfaces.
- Start all the solver agents.
- **For the solvers**

  While goals not met, solve local simulation problem else wait for input from mediators.
- **For the mediators**

  Wait for new interface values from both solvers then use relaxer to compute new interface values. If goals not met send interface values to solvers

The global control terminates the computation when all agents become idle. In practice, the global control sets more intricate goals and policies and it can accept input from a user to modify the policies dynamically. Note that a solver agent is much bigger than a mediator, perhaps by a factor of 100 to 1000 in lines of code. Solvers also typically take much longer to execute by a factor of 1000 to 100,000; a single component solution time can range from seconds for toy problems to minutes for realistic two-dimensional problems to several hours for three-dimensional problems.

Technical Networking Issues

The SciAgents system makes all of the usual demands on networks concerning bandwidth, reliability, response time, etc., and we do not discuss any of these here; we assume the network has adequate performance. The more interesting network questions involve resource acquisition; SciAgents needs machines, software, and knowledge. We are careful to distinguish between what SciAgents does today and what future implementations should do. Today SciAgents requires the user to obtain permission to use particular machines, to assign agents to them, and to provide pointers to the //ELLPACK software [3, 4] and knowledge base. In the future it will be standard for agent based systems like SciAgents to search for machines available on the network. Also, there will be software servers on the network that provide problem solving power, with or without accompanying machine

power. This will substantially increase the power of SciAgents as it will obtain access to solvers for models of physical phenomena that the Purdue group's solvers cannot handle. Further, network servers will appear that handle parts of the solver tasks, e.g., geometric modeling, visualization, symbolic analysis, or large linear algebra problems. The SciAgents system will then evolve toward dynamically creating specialized distributed solvers as well as using its own or other monolithic solvers. It will also exploit specialized hardware resources that make their compute power accessible over the network.

So far no mention has been made of the fact ultra high level problem solvers like SciAgents must have a wide range of knowledge and expert assistance (often hidden) for users. Recall that these solvers may involve millions of lines of code and involve many sophisticated algorithms and methods which most users neither can nor want to understand. SciAgents now uses the PYTHIA expert system [14] which provides some of this assistance. We envisage that knowledge about the performance and use of solvers will be distributed widely over the network and thus SciAgents will invoke sub-agents that gather the knowledge required for a particular application.

Technical Problem Solving Issues

The principal technical problems associated with the problem solving process are discussed in four areas. All these directly involve the network structure of SciAgents.

Multi-Physics Applications. To specify the the computation for the 13 solvers in Figure 1b one must specify the geometry, equations, conditions, etc. The solver //ELLPACK of SciAgents has elaborate facilities to do this that use a number of interactive windows for each component. Thus defining the problem might involve 50–100 windows for the solvers plus another 21 for the mediators. Window management is a problem here but it is tractable because they are normally used in a rather sequential manner. As the size of applications grows, this aspect of SciAgents will require a more elaborate management facility.

During the solution process and the postprocessing stage one wants to be able to see the solutions and examine the behavior along the interfaces. For Figure 1b this requires 34 windows and 13 of these are to display complex graphics about the component solutions, displays that are changing with every solution made during the iteration. It is clearly easy to overload a single workstation's communication and display capacity with even modest size application. The current SciAgents system only displays simple numerical summary values (e.g., maximum change of a component solution from one iteration to the next, maximum error in satisfying interface constraints) during the iterations. Graphics for final solutions can be displayed during the postprocessing phase.

The most sensitive part of the problem solving process is the choice of relaxers. The mediators provide a library of standard relaxers for the user to select. During the solution process the user can open a mediator's window and modify this selection or change the parameter of a particular relaxer. The current modest level understanding and experience with this problem solving process requires that the knowledge base system of SciAgents needs substantial enhancement for this task.

Resource Location. We envisage heterogeneous solvers running as "agents" or "servers" on networked, heterogeneous computing platforms. Discovering what resources are available in the

network, what kind of performance (and price) commitments we are getting from them, are all important tasks for *SciAgents*. Presently, we employ a simple solution using *Agent Name Servers (ANS)*. An ANS runs as an HTTP server, and allows queries about specific server names. Though not currently fully implemented in our system, ANSs can talk to one another, much like the DNS system for IP name to IP address binding. However, this is a simple first step, and will not scale up to real networks. This is in part due to the fact that ANS does not "understand" the semantics of the service needed. The required service can be called different names, and reliance solely on name matches (be it for servers or services) is not likely to succeed in the scientific computing domain. Clearly, there must be some format for agents to advertise their capabilities, and for other agents to reason about these capabilities. We hope to develop a language and ontology to enable this. PDESpec [13] is an primitive version of such a language for PDE based systems. This problem has analogs in the world of distributed OO systems which usually use well defined interfaces to couple components. To achieve language independence, there are a variety of interface description languages associated with systems such as OMG CORBA, Matchmaker, IBM SOM, and Microsofts COM. These allow clients to express operations on remote objects and they enforce type conformance. Loosely speaking, this corresponds to invoking the appropriate method from a server. For most systems, this is done via a class based mechanism. In other words, the client specifies its requirements (interface) as a class (say PDESolver class, which can solve any PDE). Servers which are instances of this class, or its subclasses (say 2 class, which has special methods to solve 2D PDEs more efficiently) will conform to the requirements. However, this technique has several disadvantages and leads to the creation of a complicated class hierarchy even in simple systems. A recent approach based on signatures has been proposed [10] which is a cleaner implementation of conformance, and is also more suited for large scale networked systems that we envisage for scientific computing. The basic idea here is for each object to export its signature, composed of the operations it can perform. For example, an object could say that it is of *File* class, and has signature *(Read, Write)*. However, even in this method, there is no obvious way to clarify the semantics of the operation. For example, in the context of scientific computing, the signature *(solve)* would not make sense unless one could identify what kind of a system the object claimed to solve.

Several other issues of interest in networked scientific computing are also not addressed by distributed OO models. For example, take the issue of format translation. Say object A expects a matrix represented as an array, but object B produces the matrix as a linked list. Another issue is one of performance commitments. For example, the agent coordinator would need to know not only that a particular linear system solver can solve linear systems, but how much time it would take and how much cost would be incurred, whether there was a tradeoff which could be achieved, etc. We are attempting to address these problems using the ontology we develop in conjunction with KQML. Very recently, there has been some work done by Fox *et. al.* (`http://www.npac.syr.edu/techreports/`) to use Java and the Web as a means of networked resource location. This work is related to our initial efforts with systems like Web //ELLPACK, but relies on RPC like mechanisms such as JAVA RMI.

Distributed Load Balancing. The SciAgents approach needs to have the loads balanced on the various machines as the iteration proceeds roughly at the speed of the slowest component processor. Currently SciAgents depends on the user to partition the application to balance the

loads. The //ELLPACK system has several tools to subdivide a given component into pieces with nearly equal computational work. These can be used to help with the load balancing but they are not integrated with SciAgents in any way.

We believe that the best strategy for the future is decouple load balancing (and component partitioning) from the assignment of solvers to machines. This can be done with a threads of control approach: Define 100 different computational domains and solver agents, then estimate roughly their computing needs, and then allocate the 100 solvers to, say, 11 actual machines to nearly equalize the work per machine. This approach easily accommodates heterogeneous collections of machines and the allocation process is a very simple calculation. The disadvantages of this approach are that more large software systems are launched, more components and windows must be managed by the user, and network resource use is higher. These disadvantages suggest that the threads of control approach should be used in moderation, experiments are needed to evaluate better the trade-offs between the two approaches.

Distributed Knowledge Acquisition. An important task of *SciAgents* is to select the appropriate software components and hardware platforms given a problem. We refer to this process as the *algorithm selection problem* [12], and implement it in the form of *consulting agents.* The basic idea is to use prior experience in solving similar problems to find the right hardware and software to solve a new one. The selection process is constructed in a modular fashion so that different or even multiple selection schemes can be used, e.g., neural nets or fuzzy logic. The success of these selectors depends on having a "knowledge base" about how various solvers perform on various problems. Such knowledge bases are being systematically generated using batteries of test problems at Purdue. These knowledge bases will be further augmented by *SciAgents* by allowing performance data to be dynamically captured. This data will be added to the knowledge base and, in the long run, the consulting agents will become better in selecting algorithms because of the experience they gain. In an actual setting, the knowledge bases will be distributed across users on the network, and each consulting agent will have only some of the information needed to correctly determine the hardware and software choices.

Our approach to selection searches the data base for previously seen problems close to the new one by classifying problems into meaningful classes. This would imply that finding the closest problem would be a two stage process, where one would first find the appropriate class for a new problem, and then look for close problems amongst the members of that class. The classification of problems into subsets and determining which subset a particular problem belongs to can be implemented in several ways. While there are some problem classes where there is a completely deterministic (and simple) way to determine class membership, for other classes such a priori determination is not possible. We have to determine the class structure based on samples seen thus far. In other words, the class structure has to be learned given the examples. We have developed architectures for addressing the learning issue using a variety of "intelligent" approaches, and experimented with a variety of connectionist, fuzzy and hybrid approaches to learning. It has been our experience [6, 8], that specialized selection techniques developed for particular application domains tend to outperform more general, conventional techniques.

With the selection mechanism outlined above, the quality of the answer obtained depends on the quality of the database – i.e., on how many and what kind of problems it has. This can be

a significant problem if the database contains few problems overall, or few (or no) problems of a specific type. This deficiency can be alleviated by using a cooperative agent based approach. Thus different users, possibly on different machines across the network, have different consulting agents. Due to the difference in the kind of problems addressed by different users, each agent can be expected to have seen different problems. Whenever a selection problem is posed to a consulting agent, it tries to answer it using its own database. If it finds that it cannot find an answer, or that it has a low confidence in its answer, it queries other agents. Each agent then replies with an answer, and encloses information about "how much confidence it has in the answer" and "how much does it know about the kind of problem". The agent receives all these answers and then uses them to make a selection. Clearly, this technique is inefficient, since it requires a broadcast of the query to all other agents. We have developed and experimented with new techniques which *learn* the capabilities of consulting agents from these initial broadcasts, and then direct queries appropriately. The learning uses a variety of single pass neuro-fuzzy techniques that we have recently developed [8, 6, 11, 7]. Preliminary results show that these techniques do as well as the best known algorithms on a variety of data sets, and yet promote fast, single pass learning. Our approach is be unsupervised, and includes techniques based on epistemic utility theory that we have explored to automatically generate exemplars. In [5], we describe this approach applied to a collection of PYTHIA [14] agents.

SciAgents as a Network of Computing Agents

Recall that SciAgents employ two major types of computing agents – *solvers* and *mediators*. The solver is considered a "black box" by the other agents and it interacts with them only using an inter-agent language for the specific problem. This feature allows all computational decisions for solving the individual subproblem to be taken independently from the decisions in any other subproblem – a major difference from the traditional approaches to multi-physics simulations. Each mediator agent is responsible for adjusting an interface between two neighboring subdomains. Since the interface between any two subdomains may be complex itself, there may be more than one mediator assigned to adjust it, each of them operating on separate piece of the whole interface. Thus the mediators control the data exchange between the solvers working on neighboring subproblems by applying mediating formulas and algorithms to the data coming from and going to the solvers. Different mediators may apply different mediating formulas and algorithms depending on the physical nature of their interfaces. The mediators are also responsible for enforcing global solution strategies and for recognizing (locally) that some goal (like "end of computations") has been achieved. The solvers and mediators form a network of agents that solves the given global problem. A schematic view of the functional architecture of SciAgents is given in Figure 2.

We now describe how the user builds ("programs") this network for the problem in Figure 1. We see that SciAgents provides a natural abstraction to the user in the problem domain and hides the details of the actual algorithms and software involved. The user first breaks down the geometry of the composite domain into simple subdomains with simple models to define the problems for each subdomain. Then the physical interface conditions between the subdomains are identified. All that can be done in the terms of the user's problem domain. Then the user constructs the proper

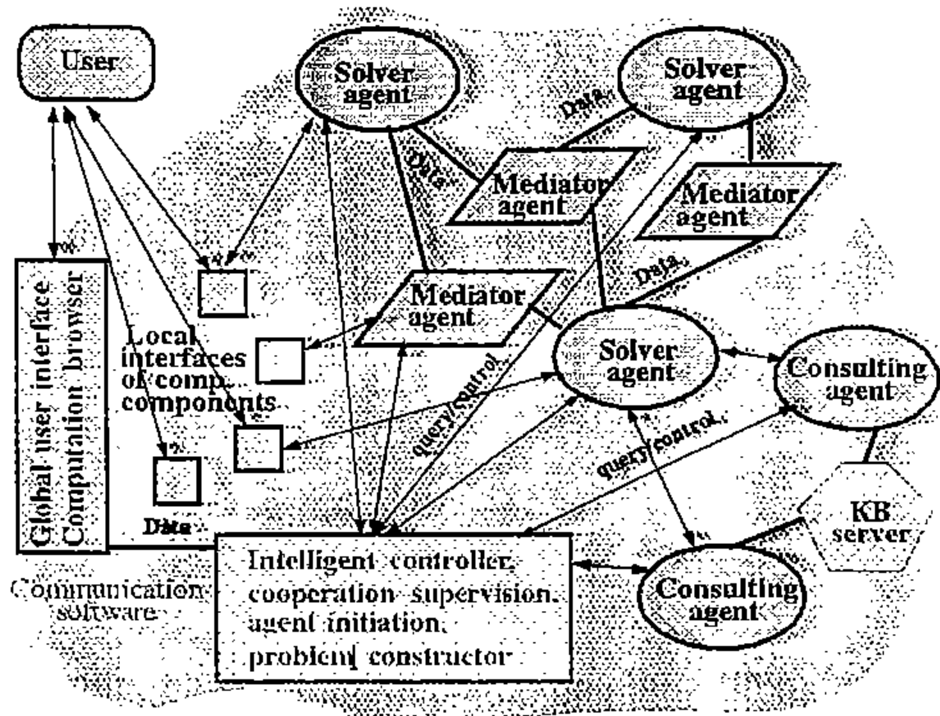


Figure 2: Functional architecture of SciAgents. The computations (and the major data exchange) are concentrated in the network of solver and mediator agents. The computing agents communicate with the consulting ones through queries to obtain "advice" on computation parameters. The user interacts with the system through the global and local user interfaces which send queries and receive replies from the various agents. The intelligent controller and the problem constructor are integrated into a single "agent" which controls the global state of the computations and instantiates, queries, and manages the other agents.

network of computing agents by simply *instantiating* various agents. The user is provided with an *problem constructor* (*agent instantiator*) — a process which displays templates for defining solvers (e.g., giving geometry, equations) and for defining mediators (e.g., giving connections, relaxers). This creates active agents of both kinds, capable of computing. Initially, only *templates of agents* — structures that contain information about solver and mediator agents and how to *instantiate* them, are available. The user selects solvers that are capable of solving the corresponding component problems and mediators that are capable of mediating the physical conditions along the specific interfaces, and provides appropriate information to them. The user interacts with the system using a visual programming like approach, these have proved useful in allowing the non-experts to "program" by manipulating images and objects from their problem domain. In our case, a visual environment is useful for the constructor, or when the user wants to request some action or data.

Once an agent has been instantiated, it takes over the communication with the user and with its environment (the other agents) and tries to acquire all necessary information for its task. Each agent retains its own interface and can interact with the user. It is convenient to think of the user as another agent in these interactions. The user defines each component problem independently, using the solver agent user interface. then interacts similarly with the mediators to specify the physical conditions holding along the various interfaces.

The agents *actively* exchange partial solutions and data with other agents without outside control and management. Thus, each solver agent can request the necessary domain and problem related data from the user and decide what to do with it (should it, for instance, start the computations or should it wait for other agents to contact it?). After each mediator agent has been supplied with the connectivity and mediating data by the user, it contacts the corresponding solver agents and requests the information it needs. By instantiating the individual agents (defining individual components and interfaces) the user builds the highly interconnected and interoperable network that solves the problem, by *cooperation* between individual agents.

The user's high-level view of the SciAgents architecture is shown in Figure 3. The global communication medium used by all entities in the SciAgents is called a *software bus* [13]. The constructor (user builder interface) uses the software bus to communicate with the templates in order to instantiate various agents. The order of instantiating the agents is not important. If a solver agent is instantiated and it does not have all data it needs to compute a local solution (i.e., a mediator agent is missing), then it suspends the computations and waits for some relaxer agent to contact it and to provide the missing values (this is also a way to "naturally" control the solution process). The mediator agents act similarly. This built in synchronization is, we believe, an important advantage of the SciAgents architecture.

Since agent instantiation happens one agent at a time, the data which the user has to provide (domain, interface, problem definition, etc.) is strictly local, and the agents collaborate in building the computing network. The user actually does not even need to know the global model. We can easily imagine a situation when the global problem is very large. Different specialists might only model parts of it. In such a situation, a user may instantiate a few agents and leave the instantiating of the rest of the cooperating agents to colleagues. Naturally, some care has to be taken in order to instantiate all necessary agents for the global solution and not to define contradictory interface conditions or mediation schemes along the "borders" between different users.

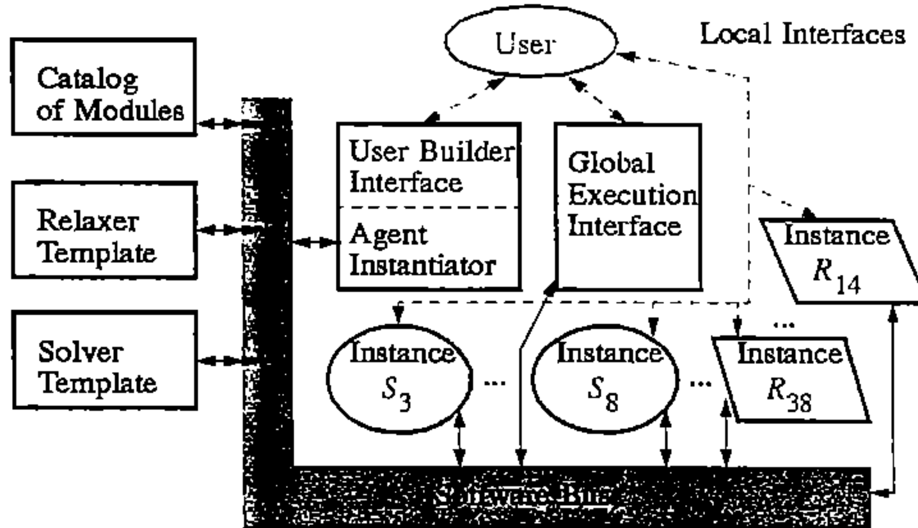


Figure 3: User's abstraction of the SciAgents software architecture. The user initially interacts with the User Interface Builder to define the global composite problem. Later the interaction is with the Global Execution Interface to monitor and control the solution of the problem. Direct interaction with individual solvers and mediators is also possible. The agents communicate with each other using the *software bus*.

The collection of agent interfaces that a user interacts with is the only software the user actually needs to run locally in order to solve the problem. Everything else can be done over the network.

This user view of the SciAgents architecture is too abstract for an actual implementation. For this one has to specify the internal architecture of each agent and the detailed communication among the agents, see [2] for these important details. The implementation utilizes the locality of the communication patterns and the fact that whenever a mediator is active (computing), the corresponding solvers are idle and vice versa. Also, the asynchronicity of the communication and the need of implementing the "pro-active" feature of the agents prompts us to employ many active threads in a single agent (multithreading).

Coordination of the Solution Process. There are well-defined global mathematical conditions for terminating the computations, for example, reaching a specified accuracy, or failing to converge. In most cases, these global conditions can be "localized" either explicitly or implicitly. For instance, the user may require different accuracy for different subdomains and the computations may be suspended locally if local convergence is achieved.

The local computations are governed by the mediators (the solvers simply solve the mathematical problems). The mediator agents collect the errors after each iteration and, when the desired accuracy is obtained, *locally* suspend the computations and report the fact to the global controller. The solvers continue to report the required data to the mediators and they check whether the local interface conditions continue to be satisfied with the required accuracy. If a solver receives information that convergence has been achieved for all its interfaces, then it stops the iterations.

They may be restarted if the interface conditions relaxed by a given mediator agent are no longer satisfied (even though they once were). The only global control exercised by the controller is to terminate all agents in case all mediators report local convergence or one of them reports a failure. The global solution "emerges" from the local computations as a result of intelligent decision making done locally and independently by the mediator agents. The agents may change their goals dynamically according to the local status of the solution process – switching between observing results and computing new data.

Other global control policies can be imposed by the user if desired – the system architecture allows this to be done easily by distributing the control policy to all agents involved. Such global policies include continuing the iterations until the all interface conditions are satisfied, and recomputing the solutions for all subdomains if the user changes something (conditions, method, etc.) for any domain.

Software Reuse and Evolution. One of the major goals of the SciAgents concept is to design a system that allows for low-cost and less time-consuming methods of building the software to simulate a complex mathematical model of physical processes. This goal cannot be accomplished if the existing rich variety of problem solving software for scientific computing is not used. More precisely, there are a number of well-tested, powerful, and popular solvers for use after breaking the global model into "simple" subproblems defined on a single subdomain. These can easily and accurately solve such a "simple" submodel and it is natural to use them. However, our architecture requires the solvers to behave like agents (e.g., understand agent languages, communicate data to other agents), something no solvers in scientific computing are able to do to the best of our knowledge.

Thus we provide an *agent wrapper* for solvers and associated software to take care of the interaction with the other agents and with the other aspects of emulating agent behavior. The wrapper encapsulates the original functionality and is responsible for running it and for the necessary interpretation of parameters and results. This is not simply a "preprocessor" that prepares input and a "postprocessor" that interprets the results, since mediation between subproblems requires communicating to the mediators some intermediate results and/or accepting some additional data from them. Designing the wrapper is sometimes complicated by the "closed" nature of the solvers — their original design is not flexible or "open" enough to allow access to various parts of the code and the processed data. However, we believe that the solver developers can design and build such a wrapper for a very small fraction of the time and the cost of designing and building the entire solver. The wrapper, once written, will enable the software's reuse as a solver agent in different MPSEs, thus amortizing the cost further. An additional task is to evaluate the solver's user interface — since the user defines the local problems through it, it is important that the interface facilitates the problem definition in user's terms well.

References

[1] Drashansky, T., M. Mu, J. Rice, and M. Vavalis, Collaborating PDE solvers, http://www.cs.purdue.edu/people/jrr/slides/Collaborating/index.html.

[2] Drashansky, T., *An Agent-Based Approach to Building MPSEs*, Ph.D. Thesis, Department of Computer Sciences, Purdue University, 1996.

[3] Houstis, E.N. and J.R. Rice, Parallel ELLPACK: A development and problem solving environment for high performance computing machines, in *Programming Environments for High-Level Scientific Problem Solving*, (P. Gaffney and E. Houstis, eds.), North Holland, Amsterdam, (1992), 229–241.

[4] Houstis, E.N., J.R. Rice, S. Weerawarana, A.C. Catlin, P. Papachiou, K.-Y. Wang, and M. Gaitatzes, Parallel (//)ELLPACK: A problem solving environment for PDE based applications on multicomputer platforms, submitted for publication, (1996).

[5] Joshi, A., To Learn or Not to Learn ..., *Adaptation and Learning in Multiagent Systems*, (Weiss, G. and Sen, S.), Springer Verlag, Lecture Notes in Artificial Intelligence, **1042**, (1996).

[6] Joshi, A., S. Weerawarana, E.N. Houstis, J.R. Rice, and N. Ramakrishnan, Neuro-fuzzy support for problem solving environments, *IEEE Computational Science and Engineering*, **3**, (1996), 44-56.

[7] Joshi, A. N. Ramakrishnan, J.R. Rice, and E.N. Houstis, A neuro-fuzzy approach to agglomerative clustering, in *Proc. IEEE Intl. Conf. on Neural Networks*, IEEE Press, **2**, (1996), 1028–1033.

[8] Joshi, A., N. Ramakrishnan, J.R. Rice, and Houstis, E., On neurobiological, neuro–fuzzy, machine learning and statistical pattern recognition techniques, *IEEE Trans. Neural Networks*, **8**, (1997), (to appear).

[9] Mu, M., and J.R. Rice, Modeling with collaborating PDE solvers – Theory and practice, *Comp. Syst. Engr.*, **6**, (1995), 87–95.

[10] Muckelbauer, A., and V.F. Russo, The Rennaisance Distributed Object System, `http://www.cs.purdue.edu/Rennaisance/overview.ps`

[11] Ramakrishnan, N., A. Joshi, S. Weerawarana, E.N. Houstis, and J.R. Rice, Neuro-fuzzy systems for intelligent scientific computing, in *Proc. Artificial Neural Networks in Engineering ANNIE '95*, (1995), 279–284,

[12] Rice, J.R., Methodology for the algorithm selection problem, *Performance Evaluation of Numerical Software*, (L. Fosdick,ed.), North Holland, (1979), 301–307.

[13] Weerawarana, S., Problem Solving Environments for Partial Differential Equation Based Applications, Department Computer Sciences, Purdue University, (1994).

[14] Weerawarana, S., E.N. Houstis, J.R. Rice, A. Joshi, and C.E. Houstis, PYTHIA: A knowledge based system to select scientific algorithms, *ACM Trans. Math. Software*, **23**, (1997), to appear.