1995

SciAgents--An Agent Based Environment for Distributed, Cooperative Scientific Computing

Tzvetan T. Drashansky

Anupam Joshi

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

Report Number:
95-029

SCIAGENTS--AN AGENT BASED ENVIRONMENT FOR DISTRIBUTED, COOPERATIVE SCIENTIFIC COMPUTING

Tzvetan T. Drashansky
Anupam Joshi
John R. Rice

CSD-TR-95-029
April 1995

SciAgents — An Agent Based Environment for Distributed, Cooperative Scientific Computing*

Tzvetan T. Drashansky, Anupam Joshi, John R. Rice
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398
email: {ttd,joshi,jrr}@cs.purdue.edu

Abstract

Problem solving using complex mathematical models of the physical phenomena requires expert knowledge in a variety of fields of computer science, such as parallel computing and numerical methods. This often makes application scientists, who have the domain expertise to devise the mathematical models, unable to use the power of High Performance Computing (HPC) systems. *SciAgents* is a problem solving environment to allow these models and systems to become truly easy to use for the application scientists, much like PC-based systems. It is based on the agent-oriented model of computing. In this paper, we discuss the design and architecture of *SciAgents*. We present a set of artificial/computational intelligence techniques used by the cooperating agents that constitute *SciAgents* , which allows them to complete the program specification and to carry out the program execution with minimal need for user intervention. We describe the design in context of scientific computing models based on partial differential equations. *SciAgents* permits the non-expert user to cost-effectively and easily develop software for solving complex mathematical models. It is scalable and allows for extensive reuse of existing software.

1 Introduction

Scientific computing involves numerical models of real-world phenomena and, with advances in the physical sciences, these have become increasingly complex. A realistic model of any system contains a number of simpler models of subsystems that are part of it. These models interact with each other, reflecting the interaction taking place in the modeled system. The behavior of the entire complex system emerges from the combined actions of all parts.

The software systems for scientific computing reflect this complexity. The designers of the models and the consumers of the numerical results are usually application scientists or

*This work was supported in part by NSF awards ASC 9404859 and CCR 9202536, AFOSR award F49620-92-J-0069 and ARPA ARO award DAAH04-94-G-0010

engineers. With the increasing sophistication of HPC hardware and numerical software, it is becoming increasingly difficult for them to develop these complex software systems. Recognizing this problem, teams of experts have developed general problem solvers applicable to a relatively large set of homogeneous, relatively simple, and isolated models; these solvers encapsulate significant amount of knowledge from mathematics, scientific computing, parallel computing, scientific visualization, etc. A good example for such solvers is //ELLPACK [10, 22] which is designed to handle partial differential equations models.

It is generally accepted that universal solvers for the complex heterogeneous models described above cannot be built. Different software for solving each individual problem or small class of problems is necessary. Developing such software from scratch (even using libraries and object-oriented technologies) is a very slow and costly process. However, if the model is broken down to a collection of simple submodels, and if their interactions can be mathematically modeled, then a collection of simpler interacting problem solvers can solve the complex model. Such an approach has several advantages, including the possibility of reusing well-built and tested software, modularity and flexibility, low cost *inter alia*. It requires the resolution of issues like the management of the computations (processing the interactions between the submodels and deriving the global solution), the complexity of problem definition and computation specifications by the user, and the scalability of the approach.

In this paper we present and discuss the design and architecture of *SciAgents* – an agent-based programming environment for scientific computing which addresses these issues. The features of *SciAgents* make it a useful and powerful tool to make scientific computing more accessible to application scientists.Various components of this system have already been developed, we are now working on building a *SciAgents* prototype.

The trend of increased abstraction, encapsulation, and modularization in software technology has brought about in the recent years the concept of an *agent*. The agent-based paradigm is considered [29, 1, 8] a step beyond object-oriented computing. There is no universally accepted definition of an *agent* and the corresponding paradigm; some authors [29, 24] distinguish between a "weak" and a "strong" notion of agents. Our discussion here is close to the weak notion as presented in [29]. An agent usually denotes a system that possesses the following properties [29]:

- *Autonomy*: Agents operate without the direct intervention of humans, and have control over their actions and internal state.
- *Social ability*: Agents interact with other agents (and possibly humans) via some kind of *agent-communication language*.

- *Reactivity*: Agents perceive their environment and respond to changes that occur in it.
- *Pro-activeness*: Agents do not simply respond to their environment, they are able to exhibit goal-directed behavior and *take the initiative*.

Many agent-based systems have been developed [29, 24, 26, 23, 9], which demonstrate the advantages of the agent technology. One of their important aspects is their modularity and flexibility, it is very easy dynamically to add or remove agents, to move agents around the computing network, to organize the user interface. An agent based architecture provides a natural method of decomposing large tasks into self-contained modules, or conversely, of building a system to solve complex problems by a collection of agents, each of which is responsible for small part of the task. Agent-based systems can minimize centralized control.

Hitherto, the agent-based paradigm has not been used in scientific computing. We believe that using it in handling complex mathematical models of the type described earlier is natural and direct. It allows *distributed problem solving* [17] which is distinct from merely using distributed computing. The expected behavior of the simple model solvers, computing locally and interacting with the neighboring solvers, effectively translates into a behavior of a *local problem solver* agent. The task of relaxing the interface conditions between adjacent subdomains is given to *mediator* agents. The ability of the agents to autonomously pursue their goals can resolve the problems during the solution process without user intervention. This allows seamless derivation of the global solution. For similar techniques in a different context see Lesser *et al.* [14], Smith *et al.* [25], Cammarata *et al.* [2], Wesson *et al.* [28].

We develop the *SciAgents* approach in the context of solving models based on partial differential equations (PDE). Such models are among the more complex examples that arise in scientific computing. We next introduce PDE models, a traditional way to solve them, and mathematical modeling of the interactions between the submodels.

1.1 Solution of Partial Differential Equations

Many physical phenomena are modeled mathematically by partial differential equations (PDEs). When the model of the phenomenon is simple enough, then the resulting PDE problem consists of a single domain with a single PDE defined on it (together with appropriate boundary conditions and initial conditions). Solving such a PDE problem numerically involves specifying the geometry of the domain, the PDE, and the boundary conditions in proper data structures, discretizing the domain according to a selected numerical method, forming a (non)linear system of equations, and solving it. The user of the solution might also like to visualize it. It is clear that one needs a high level of expertise in scientific computing in order to solve such a single-domain PDE problem efficiently and accurately. There

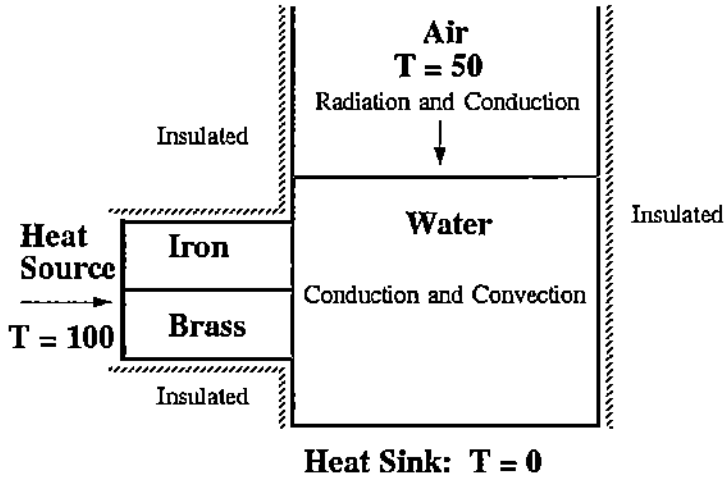

Figure 1: A simple heat flow problem.

are general solvers for this class of problems like //ELLPACK [10, 22] which has tools for defining the problem, a set of discretization methods for various problems, a set of linear equation solvers, and a set of routines for visualization of the solution. It also makes use of HPC hardware. The graphical and symbolic user interfaces allow quick definition of the problem and the entire solution process takes considerably less time and expert knowledge than needed for developing software for the particular problem.

The single-domain PDE problems, however, can not adequately model many of the physical world phenomena. With the increasing use of the computers for real-world scientific simulations there is a growing need to solve multiple-domain PDE problems. The present prototype of *SciAgents* can be used to solve the following class of PDE models.

- The physical phenomenon consists of a collection of simple connected parts.
- Each part obeys a single physical law locally (that can be modeled by a single PDE) in a single subdomain.
- The different parts influence each other and work together by adjusting interface conditions along the subdomain boundaries with neighbors.

Such features are common in models of physical events or processes. An example of such a problem is given on Figure 1. It models the temperature distribution in a small system of 4 different substances (with different laws for temperature distribution), a heater, and a sink.

Multiple-domain PDE problems also arise when a single-domain problem is broken down into smaller subdomains in order to distribute the computations among several computing units (i.e., when parallelizing the problem). This case can be viewed as an instance of the

more general case outlined earlier. Naturally, we may have a combination of the two if some subdomains in a multiple-domain problem are decomposed into still smaller subdomains. Multiple-domain PDEs often have complicated geometry and are highly non-homogeneous. As an example, consider simulating a vehicle engine. In addition to the above features, this problem also involves multiple time scales. Such problems require variable grid density and different discretization methods in different subdomains due to the different nature of the PDEs involved. The traditional domain decomposition methods consider and discretize an entire problem as a whole, before decomposing the resulting (huge) linear system for processing. This is necessary since these methods need to synchronize the grid points along the subdomain interfaces. The size of some important problems, however, is so big that considering the entire problem domain is itself an almost impossible task. For example, the engine simulation is estimated to require 100 million variables and the answer (the data set allowing the display of the accurate solution at any point) is 20 gigabytes in size. The problem contains about 10,000 subdomains with 35,000 interfaces [16].

Clearly, custom software is required for solving each multiple-domain PDE problem and it is not feasible to build it with the traditional software development technologies. On the other hand, it is easy to observe that if we have the usual boundary conditions, each subdomain problem can be considered a single-domain PDE problem for which we have efficient, existing software like //ELLPACK. Therefore, we can apply the method sketched before – we can build a network of single-domain solvers. One can view each problem solving process as an autonomous agent in a multi-agent system.

The main issue is how to obtain a global solution out of the local solutions. To do this, we use the interface relaxation technique [5, 4, 16, 15]. Important mathematical questions of the convergence of the method, the behavior of the solution in special cases, etc., are addressed in [16]. This technique uses physical relations among the parts of the model. The local phenomenon in each subdomain obeys a single physical law modeled by a PDE. The conditions between the subdomains may be obtained from knowledge of the physics which determines *interface conditions* that must be satisfied. In some cases the subdomain decomposition is created artificially and the interface conditions are derived from mathematics. Figure 2 shows the schematic of a PDE application and the structure of its decomposition into six subdomains which are handled by separate agents.

Along each interface there are conditions to be satisfied. Typically, for second order PDEs, there are two physical or mathematical conditions involving values and normal derivatives of the solutions on the neighboring subdomains. Examples for common interface conditions are given in [5, 16]. The interface relaxation technique can be described as follows.

Step 1. Choose initial information as boundary conditions to determine the PDE solutions

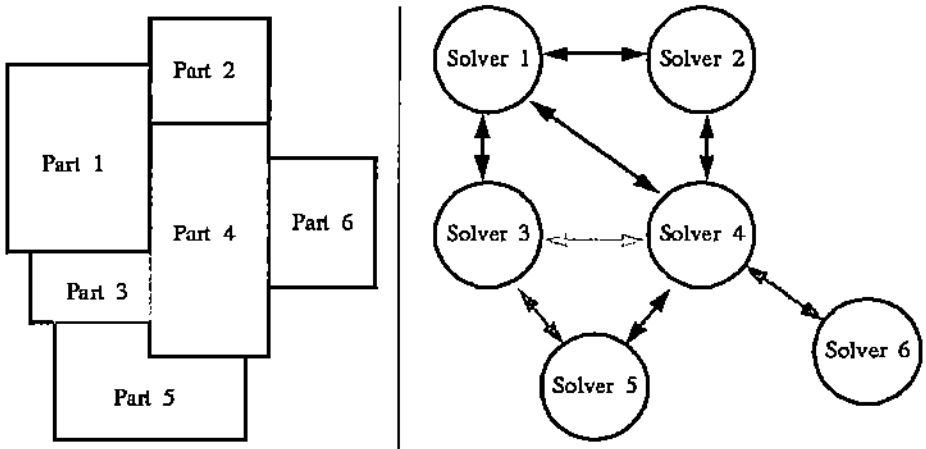

Figure 2: On the left is shown a schematic of the geometry of a physical phenomenon with six parts. The PDE model of this physical phenomenon can be represented by a network (right) of six solvers and eight interface conditions represented by arrows.

in each subdomain.

Step 2. Solve the PDE in each subdomain and obtain a local solution.

Step 3. Use the solution values to evaluate how well the interface conditions are satisfied along along the interfaces. Use a *relaxation formula* to compute new values of the boundary conditions.

Step 4. Iterate steps 1 to 3 until convergence.

This method is very convenient for the purpose of designing a mechanism that allows fast creation or assembly of a software system for solving multiple-domain PDE problems. It also allows the reuse of existing trusted software for solving single-domain PDEs.

2 User's View of *SciAgents*

In this section we discuss the design and the software architecture of *SciAgents* from the user's point of view. We also present computational/artificial intelligence techniques that free the user from the burden of specifying the computational details of the local solution process. Again, the details are given in the context of solving multiple-domain PDE problems. The framework, however, is applicable to any general mathematical model.

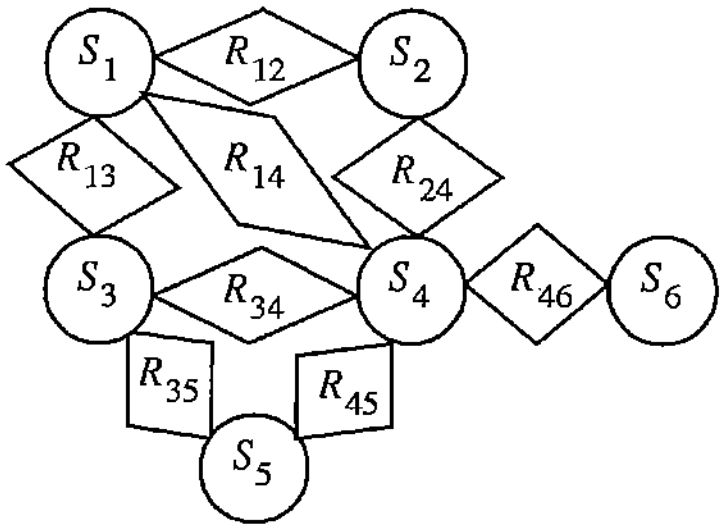

Figure 3: The user constructs a network of cooperating computing agents: solvers and relaxers. The network for the problem of Figure 2 is shown with six solvers, S_i, and eight relaxers, R_{ij}.

2.1 Defining the Problem by Building a Network of Cooperating Computing Agents

Consider a PDE problem like the one shown on Figure 2. It contains several subdomains with a single PDE to solve in each of them. The user needs to break down the geometry of the composite domain into simple subdomains with single PDE models. Then the interface conditions have to be defined in terms of the subdomain solutions and their derivatives. This preliminary work is primarily the responsibility of the individual PDE solver user interfaces but the *SciAgents* system coordinates this process. Then, a *network of computing agents* is created to solve the global problem by doing local computations and by exchanging data with other agents and with the user.

There are two major types of agents - *local problem solvers* and *mediators*. The *solvers* are responsible for finding local solutions of the single subdomain PDEs. The *mediators* are responsible for the interaction between different local problem solvers. In the PDE context, these agents are called *solvers* and *relaxers*. The *relaxers* are responsible for relaxing the interface conditions in a way that provides global convergence of the algorithm. A network for solving the problem in Figure 2 is given in Figure 3. Each relaxer agent controls a single interface between two subdomains, and each solver agent is responsible for a single domain.

The agent technology provides a natural and convenient way to hide the details of the actual algorithms and software involved in the problem solving. It allows one to use *SciAgents* to solve complex problems without being proficient in computer science (numerical methods, parallel computing, etc.).

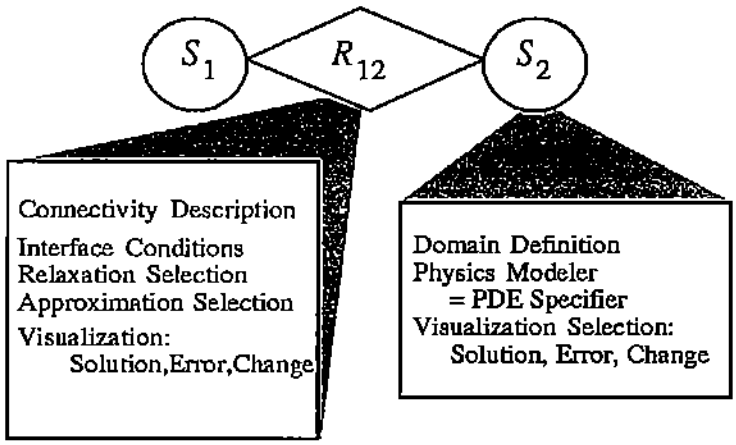

Figure 4: Computing agents: what the user sees in a solver and a relaxer.

The abstraction, which we call user's view of the agents (see Figure 4), serves two main purposes. It encapsulates the actual software architecture of *SciAgents* so that the user is presented with only the necessary details for an accurate definition of the problem and the solution criteria. It also provides the user with means to define the model and the corresponding environment and to actively participate, if desired, in the solution process (in other words, to provide an adequate user interface).

In the user's view, the individual solver agent offers:

- (sub)domain definition — in terms of the geometry of a single domain, the user has a way to define the domain for a particular PDE,
- PDE definition — the differential operator, the right side, and the proper (not actual interfaces with any other subdomain) boundary conditions,
- visualization — how the intermediate and the final solutions are to be displayed to the user, together with important parameters like the error and the convergence rate,
- parameter selection — various solution parameters such as the discretization scheme, the linear solver, etc.

The individual relaxer agents provide ways to define and control the following:

- connectivity description — which subdomains' interface they control and relax,
- interface conditions — on each side of the interface,
- relaxation scheme — the technique used to change the interface conditions,
- approximation algorithm for the values and derivatives along the interface,

- visualization — display of the solution process and the decisions taken by the relaxer along the way, display errors and convergence rates.

As we see, the user is presented with all the necessary features for problem definition and control of the solution process, if desired. No technical details of the actual computing configuration, for example, or of the computing process are required. In particular, the user need not know or care about the way the interaction between the agents is organized and implemented. As we shall see, the user may choose different levels of interaction with the agents depending on the desired level of involvement in the solution process.

Now we return to consider the main task the user faces — creating a proper network of computing agents. Initially, *SciAgents* presents to the user only *templates of agents* — structures that contain information about solver and relaxer agents and how to create (*instantiate*) them. These templates *do not and can not compute.* The user is provided with an *agent instantiator* — a process which displays information about the templates and creates active agents of both kinds, capable of computing. Once the complex problem is broken down into subdomains, individual PDEs, interfaces, and interface conditions, the network of agents that will solve the problem can be built.

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. This is one of the advantages of the agent technology — an agent is autonomous, it controls and manages the computations, and supplies its environment with the data and control it needs to support some global computations. The agents *actively* exchange partial solutions and data with other agents without outside control and management. In other words, each solver agent can request the necessary domain and PDE related data from the user and decide what to do with it (for example, should it start the computations or should it wait for other agents to contact it?). After each relaxer agent has been supplied with the connectivity data by the user, its task is to contact the corresponding solver agents and to request the information it needs — the geometry of the interface, the capabilities of the solvers with respect to approximating values and derivatives along its interface, visualization capabilities (for example, should the relaxer display some data or can the solvers do it themselves?), etc. All this can be and is done without the involvement of the user. In a way, by instantiating the individual agents (concentrating on the individual subdomains and interfaces from the global problem only) the user builds the highly interconnected and interoperable network that is going to solve that problem, by the *cooperation* between the individual agents. Note that the user defines the communication strictly in terms of the model — by assigning interfaces to relaxer agents and subdomains to solver agents.

The user needs some high-level way of looking at *SciAgents* to monitor the processes

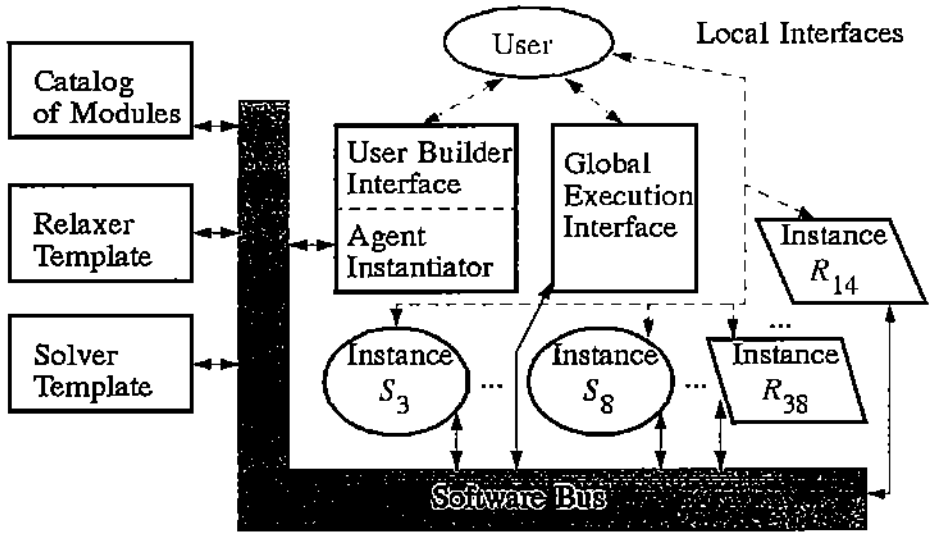

Figure 5: Software architecture of *SciAgents*: user's view. The user initially interacts with the User Builder Interface to define the global PDE problem. Later the interaction is with the Global Execution Interface to monitor and control the solution of the problem. Direct contact with individual solvers and relaxers is also possible. The agents communicate with each other using the *software bus*.

involved in solving the problem. Figure 5 depicts the abstraction we use. There is a global communication medium which is used by all entities called a *software bus* [27]. The agent instantiator communicates with the user through the user builder interface and uses the software bus to communicate with the templates in order to instantiate various agents. Agents communicate with each other through the software bus and have their own local user interfaces to interact with the user. The order of instantiating the agents is irrelevant. If a solver agent is instantiated and it does not have all boundary values it needs to compute a local solution (i.e., a relaxer 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 consecutive iterations). If a relaxer agent is instantiated and a solver agent on either side of its interface is missing, then it suspends its computations and waits for the solver agents with the necessary characteristics (the right subdomain assigned) to appear. This "intelligent" synchronization is, we believe, an important advantage of *SciAgents*. We go into more detail of the inter agent communication during the solution process below.

Since agent instantiation happens one agent at a time, the data which the user has to provide (domain, interface, PDE, etc.) is strictly local, and the agents collaborate in building the computing network. The user actually *does not even need to know the whole model.* We can easily imagine (recall the engine example) a situation when the global problem is very large. Different specialists may model parts of it. They might only know and care about

their local set of subdomains. In such a situation, a user may instantiate a few agents and leave the instantiating of the rest of the cooperating agents to colleagues. A user may even request access to the user interface of agents instantiated by others in order to observe the modeled process better. 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 relaxation schemes along the "borders" between different users.

The agent technology makes it natural to provide each agent with a user interface. It is convenient to think of the user as another agent. Then the protocol or the language used in the communication between the (computing) agents can be used to interact with the user (of course, properly translated and tuned to the human nature of the "user agent" by some small "talking agent"). The approach has the advantage of using the same commands, requests, and data structures that the agents already use. In this way we can build the user builder interface (see Figure 5), the global execution interface, and the local interface of the agents. The interfaces that we create make extensive use of visual programming techniques [3, 20] to aid the non-experts. In fact, by instantiating agents and building the problem solving network, the user creates a program which is then executed. In this process, the visual programming languages and systems 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 agent instantiator, or when the user wants to request some action or data. The semantic interpretation of the user's manipulations of icons and other objects on the screen provides the necessary base for the program to construct the answer of the system. Another useful technique from the visual programming is the so-called "zooming" of different objects. Zoom-in displays more data, zoom-out, reduces the amount of data shown. In *SciAgents* the user may zoom agents and their representations. The zooming allows to the agent to display different amount of the available data which helps the user to control the problem of too much data on the screen at one time.

2.2 Intelligent Completion of the Problem Specification

We now describe the data required to specify and solve a problem. These data include the functional problem specification as well as the internal parameters needed for the solution process. We posit that the user need only provide the functional specification. We also introduce the tools used by *SciAgents* to intelligently complete the user's specification by deducing the rest of the parameters required for the computation.

The complete functional (mathematical) description of the problem includes

- definition of the subdomains

- definitions of the PDEs in each subdomain
- definition of the boundary and initial conditions
- definition of the interfaces between subdomains and interface conditions (the latter are part of the mathematical model)

In addition, the user selects a visualization method (an aesthetic or pragmatic issue, not requiring any special "computing" expertise) and global solution criteria – say, solution accuracy (which also does not require any additional knowledge). Note that all of the above items are entirely defined in the terms of the user's problem domain and do not require any scientific computing expertise.

The agents, however, need lots of additional data, parameters, and configuration values in order to proceed with the solution process. We provide three representative examples next. First, the local solver agents need a set of computational parameters for the single-domain problem they have to solve at each iteration. These include the discretization method for the domain and the equation, grid/mesh sizes and configurations, linear solvers, etc. One of the good features of our interface relaxation technique is that the solvers do not need to coordinate the values of the parameters among themselves – each solver has complete independence in its decision about the values of these parameters. This is not the case with any other problem decomposition techniques used in PDE problem solving [21, 5].

Second, the relaxer agents have to select a set of algorithms related to the interface relaxation technique and to inform the solvers what data they need to provide after each iteration.

Third, the agents have to make use of the available hardware. It is possible that multiple computing units will be available for this particular problem. The agent instantiator will try initially to distribute the agents evenly among the computing units but it has very little information in order to make an intelligent decision – it knows only the pairs of agents that communicate with each other. The relaxer agents are distributed in an obvious manner described in the next section, and the relaxer agent computations are a small fraction of the computations necessary to obtain the local subdomain solution. The main issue is then the correct distribution of the solver agents to balance the load. This can be done by the global execution interface in several ways. One is to reassign agents [19] to appropriate computing units; another is to split some subdomains further and distribute them to separate computing units. A third possibility is to allow the individual solvers to use more than one computing unit and to do the decomposition of their subdomain internally, without affecting the interactions with the corresponding relaxer agents. These actions require reliable estimates

of the computational loads caused by the solvers. At this point we do not handle dynamic migrations and decomposition of agents.

2.2.1 The PYTHIA system

PYTHIA [11] is a system to automatically obtain the data and the parameters described above. Its objective is to advise the user of the "right", or at least "good", selections of various solvers, their parameters and the computational resources for solving a particular single-domain PDE problem.

The basic premise in PYTHIA's reasoning strategy is that performance data gathered during the solution of a set of problems using different solution methods can be used to estimate the performance of these methods on a new problem, provided of course that the problem is "similar" to the members of that set. The goal of the reasoning process then is to recommend a solution method and applicable parameters that can be used to solve the user's problem within the given computational and performance objectives. This goal is achieved by the following steps: Analyze the PDE problem and identify its characteristics. From the previously solved problems, identify the set of problems similar to the new one. Extract all information available about this set of problems and the applicable solvers and select the best method. Use performance information of this method to predict its behavior for the new problem. In order for the above procedure to produce accurate results, the database of previously seen problems must be large and growing. This however can lead to prohibitively large times for identifying similar problems. Our solution is to divide the total database into a collection of (possibly overlapping) subsets that have similar characteristics. The extraction process is done in two stages; firstly the subset to which the new problem belongs to is identified and then only its members are examined. Clearly, the quality of the answer obtained by *PYTHIA* depends on how many and what kind of problems it has previously seen. This can prove a significant problem in some cases where the *PYTHIA* databases contains few problems overall, or few (or no) problems of a specific type.

This deficiency can be partially overcome by using a cooperative agent based approach. We consider each instance of *PYTHIA* to be an agent so that different users have different *PYTHIA* agents. Different problems are solved by different users, so each *PYTHIA* agent can be expected to have seen different problems. Whenever a problem is posed to *PYTHIA*, it tries to locate similar problems in its own database. Failing that, or having a low confidence in its estimates, it queries other *PYTHIA* agents. Each agent then replies, and encloses information about "how much confidence it has in the estimate" and "how much it knows about the kind of problem". The agent receives all these answers and then uses them to make a prediction [12].

The classification of problems into subsets and determining which subset a particular problem belongs to can be implemented in several ways. We have used bayesian belief networks [27], neural networks [13], and fuzzy systems [18] in our work.

As an example for the use of the *PYTHIA* system, in *SciAgents* the solvers ask the available *PYTHIA* agent for a recommendation for each of the required parameters given the equation, the domain, and the desired accuracy. After the *PYTHIA* agent consults its knowledge base (and, possibly, other *PYTHIA* agents) it delivers back to the solvers values for the parameters and some additional information like the time estimation of the solution process (for one iteration). A similar scheme, *mutatis mutandis*, is used to obtain the other required parameters and the estimates for the amount of the solver's computing load.

3 Software Architecture of *SciAgents*

In this section we discuss details of the actual software architecture of *SciAgents* and the artificial/computational intelligence techniques used.

3.1 Actual Agent Configuration and Architecture

The software architecture of the local problem solver agents reflects our desire to reuse the existing software for solving general single-domain PDE problems. Each solver consists of a *core* implementing the functionality of the PDE solving process and the local user interface and a *wrapper* which gives the solver the behavior and the appearance of an agent. *SciAgents* is designed as an open system – it is relatively easy to add new solver agent templates with different core solvers to the set of templates in the agent instantiator's database. In *SciAgents* at the highest level communication is done using the Knowledge Query and Manipulation Language (KQML [6, 7]) from ARPA's knowledge sharing initiative. We adhere to the declarative approach in the agent interaction due to the heterogeneous environment of *SciAgents* . The contents of the messages is in the high-level language S-KIF for scientific computing. This is based on a language we developed for PDE data called PDESpec [27]. Using KQML for the inter agent communication in *SciAgents* ensures portability, compatibility, and better opportunities for extensions and the inclusion of agents built by others.

The core solvers have to be able to complete the local PDE problem definition (by making use of PYTHIA, for instance). Requirements of the minimal and the desired functionality of an existing solver before it is made available to *SciAgents* are discussed in detail in [5, 4]. During the solution process, the wrappers are responsible for employing the intelligence techniques described below.

The architecture of the relaxers facilitates the even distribution of the computations and

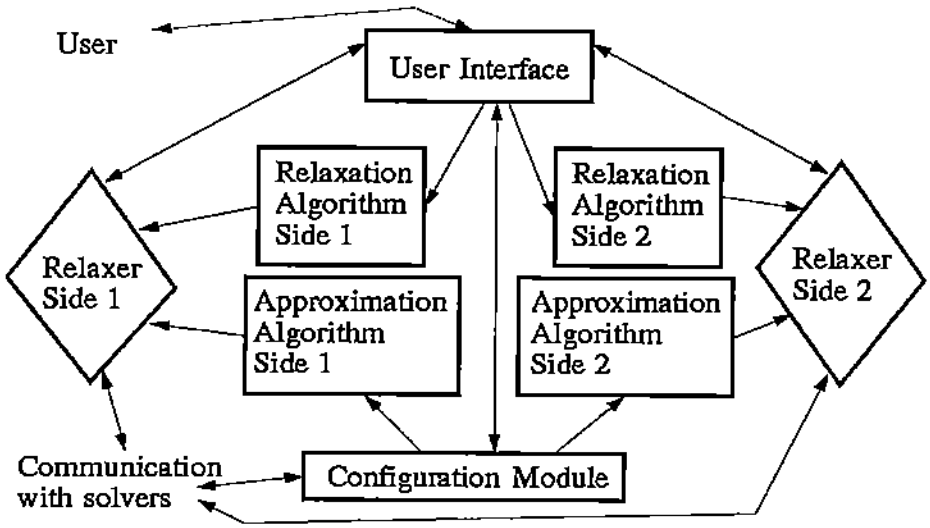

Figure 6: Software architecture of *SciAgents*: relaxer agent. The relaxer agent is divided internally into two subrelaxers — each subrelaxer controls and supplies data to and from one solver on one side of the interface. Each subrelaxer uses its own relaxation and approximation algorithms and it communicates relatively independently with the solver agent on its side of the interface. There are two shared modules — the user interface module (responsible for the interaction with the user) and the configuration module (responsible for "orienting" the agent in its environment).

efficient implementation. The interface conditions on the two sides of the interface may differ; the relaxation scheme may require different handling of the data; the approximation algorithms for the values and derivatives along the interface may be different. All this suggests that the two sides of the interface should be handled somewhat separately. This partitioned view of a relaxer agent is detailed on Figure 6. Each of two subrelaxers controls and supplies data to and from one solver on one side of the interface. Each subrelaxer uses its own relaxation and approximation algorithms and communicates relatively independently with the solver agent on its side of the interface. These subrelaxers are the processes that do the actual computation and initiate the consecutive iterations during the problem solving process. The two subrelaxers share the user interface and the configuration module. The user interface module presents the relaxer agent as a single entity to supply and request user information. It also handles requests for dynamic changes of the parameters.

The configuration module is responsible for "orienting" the agent in its environment. After the relaxer has been instantiated, the configuration module requests connectivity information (which interface am I responsible for?) and then attempts to locate the corresponding solvers. If they have been instantiated, the configuration module communicates with them in order to establish their capabilities and other necessary parameters, otherwise it suspends its activity until the required solver agents become available. It is responsible

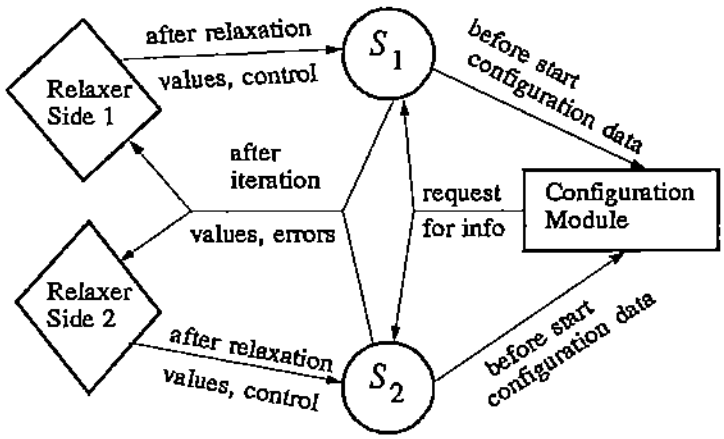

Figure 7: Relaxer agent: communication with the solvers. After the initial exchange between the solver agents and the configuration module, the information flow is very simple. In the direction from the relaxer to the solvers it can be entirely split between the two subrelaxers and their solvers. In the opposite direction the data has to be delivered to both subrelaxers.

for determining the parameters of the relaxation scheme necessary to complete the problem definition. The configuration module monitors the subrelaxers in order to terminate the iterations (locally) if convergence has been reached.

The user interface and the configuration modules are combined into a single process that exercises dynamic control over the subrelaxers. Its interface with them follows the inter agent communication protocol valid for the entire *SciAgents* . Effectively, the relaxer agent is in fact a "meta agent" consisting of 3 actual agents with significantly overlapping goals and a somewhat centralized control.

Figure 7 shows the information flow between a relaxer agent and its two solvers. After the initial exchange between the solver agents and the configuration module, the information flow is very simple. In the direction from the relaxer to the solvers it can be entirely separated between the two subrelaxers and their solvers. In the opposite direction the data has to be delivered to both subrelaxers. It it important to note that the pattern of the communication between the agents is completely local — each relaxer agent communicates with two solver agents and each solver agent communicates with the relaxers for the interfaces of its subdomain. This locality is an advantage for *SciAgents* since it allows for good scalability.

The architecture of the relaxer agents allows us to distribute N subdomain solvers and M interface relaxers among N computational units (if available) in a natural and efficient way. When the relaxers compute, the solvers are idle and vice versa due to the nature of the interface relaxation technique. We use this to build the *SciAgents* software architecture as shown in Figure 8 where each rectangle represents a computing unit. All computing units use the software bus as the communication medium. Each computing unit has a message

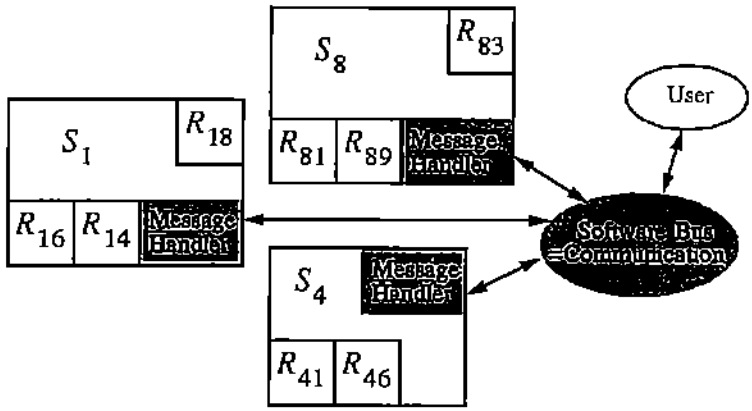

Figure 8: Software architecture of *SciAgents*: designer's view. Each rectangle represents a computing unit. There is a single subdomain solver running on one computing unit and it has all relevant parts of the relaxers for its interfaces "attached" to it. The two subrelaxers can be split between the two solvers, with the configuration module and the user interface partly duplicated. The software bus is the communication medium. Each computing unit has a message handler which may be considered a part of the software bus.

handler which may be considered a part of the software bus. There is a single subdomain solver running on one computing unit and it has all relevant parts of the relaxers for its interfaces "attached" to it.

Finally, the agent instantiator and the global execution interface are grouped together in a single agent that provides the communication with the user concerning global data and requests (composing the network of agents, defining the global constraints of the solution, etc.) and exercises the necessary global coordination among the agents during the solution process. The agent instantiator is responsible for instantiating of all computing agents. The solver agent template contains a database of the various existing solvers available at the moment. The instantiator decides where to start an agent, activates the necessary code and announces the existence of a new agent.

3.2 Intelligent Interagent Cooperation

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

The local computations are governed by the relaxer agents – the solvers simply solve PDEs – the decision making is concentrated in the mediator agents. The relaxer agents collect the errors after each iteration and, when the desired accuracy is obtained, they *locally* suspend the

computations and report the fact to the global execution interface. The suspension is done by issuing an instruction to the solvers on both sides of this interface to use the boundary conditions for the interface from the previous iteration in any successive iterations they may perform (the other interfaces of the two subdomains may still not have converged). The solvers continue to report the required data to the subrelaxers and the subrelaxers continue to check whether the local interface conditions are satisfied with the required accuracy. If a solver receives instructions to use the old iteration boundary conditions for all its interfaces, then it stops the iterations. The iterations may be restarted if the interface conditions relaxed by a given relaxer agent are no longer satisfied (even though they once were). In this case, the relaxer issues instructions to the two solvers on both sides of its interface to resume solving with new boundary conditions. If the maximum number of iterations is reached, the relaxer reports failure to the global execution interface and suspends the computations. The only global control exercised by the global execution interface is to terminate all agents in case all relaxers report local convergence or one of them reports a failure.

The above scheme provides a robust mechanism for cooperation among the computing agents. Using *only* local knowledge, they perform only local computations and communicate only with "neighboring" agents. They *cooperate* in solving a global, complex problem, and none of them exercises centralized control over the computations. The global solution "emerges" in a well-defined mathematical way 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, and the system architecture allows this to be done easily — by distributing the control policy to all agents involved. Such global policies may include continuing the iterations until the last interface conditions converge, recomputing the solutions for all subdomains if the user changes something (conditions, method, etc.) for any domain, etc.

References

[1] G. Agha, P. Wegner, and A. Yonezawa (eds), *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 1993.

[2] S. Cammarata et al., *Strategies of Cooperation in Distributed Problem Solving*, Readings in Distributed Artificial Intelligence (Bond and Gasser, eds.), Morgan Kaufmann, 1988, pp. 102–105.

[3] McWriter J. D. and G. J. Nutt, *Escalante: An Environment for the Rapid Construction of Visual Languages Applications*, Proc. 1994 IEEE Symposium Visual Languages, IEEE, IEEE Comp. Soc. Press, 1994, pp. 15–22.

[4] T. T. Drashansky, *A Software Architecture of Collaborating Agents for Solving PDEs*, Tech. Report TR-95-010, Dept. Comp. Sci., Purdue University, 1995, (M.S. thesis).

[5] T. T. Drashansky and J. R. Rice, *Processing PDE Interface Conditions – II*, Tech. Report TR-94-066, Dept. Comp. Sci., Purdue University, 1994.

[6] T. Finin et al., *Draft Specification of the KQML Agent-Communication Language*, DARPA Knowledge Sharing Initiative, External Interfaces Working Group, 1993.

[7] ———, *KQML as an Agent Communication Language*, Proc. III Intl.Conf. on Information and Knowledge Management, ACM, ACM Press, 1994.

[8] M.R. Genesereth and S.P. Ketchpel, *Software Agents*, Comm. ACM **37** (1994), no. 7, 48–53.

[9] B. Hayes-Roth et al., *Guardian. A Prototype Intelligent Agent for Intensive-care Monitoring*, Artif. Intell. Med **4** (1992), no. 2, 165–185.

[10] E. N. Houstis and J. R. Rice, *Parallel ELLPACK: A Development and Problem Solving Environment for High Performance Computing Machines*, Programming Environments for High Level Scientific Problem Solving, North Holland, 1992, pp. 229–243.

[11] E. Houstis et al., *The PYTHIA projet*, Proc. First Intl. Conf. on Neural, Parallel and Scientific Computing, 1995, (to appear).

[12] Anupam Joshi, *To Learn or Not to Learn ...*, Proc. IJCAI'95 Workshop on Adaptation and Learning in Multiagent Systems, 1995, (to appear).

[13] A. Joshi et al., *The Use of Neural Networks to Support Intelligent Scientific Computing*, Proc. IEEE Intl. Conf. Neural Networks, IEEE, IEEE Press, July 1995.

[14] V. R. Lesser, *A Retrospective View of FA/C Distributed Problem Solving*, IEEE Transactions on Systems, Man, and Cybernetics **21** (1991), no. 6, 1347–1363.

[15] S. McFaddin and J. R. Rice, *RELAX: A Platform for Software Relaxation*, Expert Systems for Scientific Computing (Houstis, Rice, and Vichnevetsky, eds.), North Holland, 1992.

[16] Mo Mu and J. R. Rice, *Modeling with Collaborating PDE Solvers — Theory and Practice*, Tech. Report TR-94-056, Dept. Comp. Sci., Purdue University, 1994.

[17] T. Oates et al., *Cooperative Information Gathering: A Distributed Problem Solving Approach*, Tech. Report TR-94-66, UMASS, 1994.

[18] N. Ramakrishnan et al., *Neuro-Fuzzy Systems for Intelligent Scientific Computing*, Tech. Report TR-95-026, Dept. Comp. Sci., Purdue University, 1995.

[19] V. Rego et al., *Process Mobility in Distributed Memory Simulation Systems*, Proc. Winter Simulation Conference, 1993, pp. 722–730.

[20] A. Repenning, *Agentsheets: A Tool for Building Domain-Oriented Dynamic, Visual Environments*, Ph.D. thesis, Dept. Comp. Sci., Univ. Colorado at Boulder, 1993.

[21] J. R. Rice, *Processing PDE Interface Conditons*, Tech. Report TR-94-041, Dept. Comp. Sci., Purdue University, 1994.

[22] J. R. Rice and R. F. Boisvert, *Solving Elliptic Problems Using ELLPACK*, Springer-Verlag, 1985.

[23] J. C. Schlimmer and L. A. Hermens, *Software Agents: Completing Patternsand Constructing User Interfaces*, Journal of Artificial Intelligence Research **1** (1993), no. 61-89.

[24] Y. Shoham, *Agent-Oriented Programming*, Artificial Intelligence 60 (1993), no. 1, 51–92.

[25] R. G. Smith and R. Davis, *Frameworks for Cooperation in Distributed Problem Solving*, Readings in Distributed Artificial Intelligence (Bond and Gasser, eds.), Morgan Kaufmann, 1988, pp. 61–70.

[26] L. Z. Varga et. al., *Integrating Intelligent Systems into a Cooperating Community for Electricity Distribution Management*, International Journal of Expert Systems with Applications **7** (1994), no. 4.

[27] S. Weerawarana, *Problem Solving Environments for Partial Differential Equation Based Systems*, Ph.D. thesis, Dept. Comp. Sci., Purdue University, 1994.

[28] R. Wesson et al., *Network Structures for Distributed Situation Assessment*, Readings in Distributed Artificial Intelligence (Bond and Gasser, eds.), Morgan Kaufmann, 1988, pp. 71–89.

[29] M. Wooldridge and N. Jennings, *Intelligent Agents: Theory and Practice*, (submitted to Knowledge Engineering Review), 1994.