Purdue University
Purdue e-Pubs

Department of Computer Science Technical Reports | Department of Computer Science

1996

Multi-Agent Simulation of Complex Heterogeneous Models in Scientific Computing

Anupam Joshi

Tzvetan Drashansky

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

Sanjiva Weerawarana

Elias N. Houstis
Purdue University, enh@cs.purdue.edu

Report Number:
96-025

Joshi, Anupam; Drashansky, Tzvetan; Rice, John R.; Weerawarana, Sanjiva; and Houstis, Elias N., "Multi-Agent Simulation of Complex Heterogeneous Models in Scientific Computing" (1996). *Department of Computer Science Technical Reports.* Paper 1281.
https://docs.lib.purdue.edu/cstech/1281

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.

MULTI-AGENT SIMULATION OF COMPLEX HETEROGENOUS MODELS IN SCIENTIFIC COMPUTING

Anupam Joshi
Tzvetan Drashansky
John R. Rice
Sanjiva Weerawarana
Elias Houstis

Purdue University
Department of Computer Sciences
West Lafayette, IN 479807

CSD TR-96-025
May 1996

Multi-Agent Simulation of Complex Heterogeneous Models in Scientific Computing*

Anupam Joshi, Tzvetan Drashansky, John Rice, Sanjiva Weerawarana and Elias Houstis
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907–1398 USA
Phone: 317-494-7821, Fax: 317-494-0739
email: {joshi,ttd,jrr,saw,enh}@cs.purdue.edu

Abstract

Electronic prototyping is becoming a part of every scientific inquiry and product design, and is the focus of research in the new scientific field of Computational Science and Engineering. The new grand challenge here is the rapid prototyping of manufactured artifacts and the rapid solution to problems with numerous interrelated elements. This, in turn, requires the fast, accurate simulation of physical processes and design optimization using knowledge and computational models from multiple disciplines in science and engineering. In this paper we formulate a mathematical and software framework for complex rapid prototyping. Its design utilizes the current computer network infrastructures and High Performance Computation technologies. Its functionality includes adaptability and intelligence with respect to end-users and hardware platforms. We present the architecture and implementation of this framework, named *SciAgents*, using a multi-agent software model encapsulating a collaborating mathematical method. The design of *SciAgents* allows wholesale reuse of scientific software and provides a natural approach to parallel and distributed problem solving.

Keywords: Distributed Problem Solving, Agent-Based Computing, Simulation, Heterogeneous Models, Software Reuse and Evolution.

1 Introduction

The growth of computational power and network bandwidth suggests that computational modeling and experimentation will continue to grow in importance as a tool for big and small science. The design process will operate at the scale of the whole physical systems with a large number of components that have different shapes, obey different physical laws and manufacturing constraints, and interact with each other through geometric and physical interfaces. We consider multi-agent problem solving systems in scientific computation and, in particular, strategies that allow scientific computing systems to solve problems cooperatively. Scientific computation involves numerical models of real world phenomenon, and these models are becoming

*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

increasingly complex. Heretofore, scientific computing systems have been developed as stand-alone systems targeted to a particular class of applications modeled in a somewhat homogeneous, generic way. However, the real world consists of physical objects that interact with each other. The overall behavior of a physical system is a result of these interactions. Consider an automobile engine. Its design involves the domains of thermodynamics (gives the behavior of the gases in the piston-cylinder assemblies), mechanics (gives the kinematic and dynamic behaviors of pistons, links, cranks, etc.), structures (gives the stresses and strains on the parts) and geometry (gives the shape of the components and the structural constraints). The engine behavior emerges from the interaction of phenomena from these different domains. They share common parameters across interfaces but each has its own parameters and constraints. An optimal engine design requires the simulation and prototyping of these multi-component physical systems.

Complex prototyping requires the development of new algorithmic strategies, as well as software for managing the complexity and harvesting the power of high performance computing and communication (HPCC) resources. It requires technology to support programming-in-the-large and to reduce the overhead of HPCC computing. Our research aims to identify the framework for the numerical simulation of multidisciplinary applications and to develop the enabling theories and technologies needed to support and realize this framework in specific applications. Our design objective is to allow the "natural" specification of complex physical systems and their simulation with interacting software components through mathematical and software interfaces across networks of heterogeneous computational resources.

The behavior of each component of a complex physical system is modeled using various formulations for the geometry, some mathematical equations for the physics, and conditions or constraints for the interfaces or linkages between components. It is difficult to imagine creating a monolithic software system to model accurately a complicated real problem with hundreds of diverse parts and dozens of physical phenomena. Therefore, one needs a software framework which is applicable to a wide variety of practical problems and allows for software reuse. Most physical systems and manufactured artifacts have a global model which is a mathematical network whose nodes represent the physical components in a system or artifact. Each node has a mathematical model of the physics of the component it represents and a solver agent[2] for its analysis. Individual components are chosen so that each node corresponds to a simple mathematical problem defined on a regular geometry. There exist many standard, reliable solver systems that can be applied to these local node problems. Some nodes in the network correspond to interfaces that model the interaction of the parts in the global model. To solve the global problem, the local solvers collaborate with each other to relax (i.e., resolve) the interface conditions. An interface controller or mediator agent[2] collects parameters from neighboring subdomains and adjusts these to better satisfy the interface conditions. This "network" abstraction of a physical system allows us to build a software system which is a network of well defined collaborating software parts using interfaces. Some of the theoretical issues of this methodology have been

addressed in [15, 16, 18] for the case of collaborating PDE models. The results obtained so far verify the feasibility and potential of network-based prototyping.

Our view of the computing environment is a network [4] where servers export to the user's machine an agent that provides an interactive user interface built on top of the standard network services. The bulk of the software and computing is done at the server's or third party sites using software tailored to a known and controlled environment. The server site can, in turn, request services from specialized resources it knows, e.g., a commercial PDE solver, a proprietary optimization package, a 1000 node supercomputer, an ad hoc collection of 122 workstations, a database of physical properties of materials. Each of these resources is contacted by an agent with a specific request for problem solving or information service. All of this can be managed without involving the user, without moving software to arbitrary platforms, and without revealing source codes. This approach also allows software reuse for easy software update and evolution, things that are extremely important in practice. For example, each new automobile engine design normally results in a new software system. Recreating such a system could easily take several months or years. In contrast, the execution time to perform the electronic prototyping might only be a few days. Since a new engine design often incorporates parts from old designs, the physical changes correspond to replacing, adding, or deleting a few nodes in the network with a corresponding change in interface conditions. In such applications each physical component is viewed both as a physical object and as a software object. In addition, this mathematical network approach is naturally suitable for parallel and distributed computing as it exploits the parallelism in physical systems. One can handle issues like data partition, assignment, and load balancing on the physics level using the structure of a given physical system. We believe that this multi-agent approach is natural and direct. It is facilitated by the existence of a multitude of stand-alone scientific problem solving agents that can effectively model and solve for the behavior of fairly simple, homogeneous physical phenomenon. Some of these agents are no more than subroutine libraries in the classical sense, others are very much larger and more sophisticated *problem solving environments* [7]. We believe that this approach will allow locally interacting problem solving agents to decompose a complex computation into a distributed collection of self contained computations. It also allows high scalability.

2 Background and Related Work

Many agent-based systems have been developed[9, 21, 22, 24, 27], which demonstrate the advantages of the agent-oriented paradigm. One of their important aspects is modularity and flexibility. It is very easy to dynamically add or remove agents, to move agents around the computing network, and 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 widely in scientific computing. We believe that using it in handling complex mathematical models 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 mediating interface conditions between adjacent subproblems 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.

Several researchers have addressed the issue of coordinating multi-agent systems. For instance Smith and Davis [23] propose two forms of multi-agent cooperation, task sharing and result sharing. Task sharing essentially involves creating subtasks, and then farming them off to other agents. Result sharing is more data directed. Different agents are solving different tasks, and keep on exchanging partial results to cooperate. They also proposed using "contract nets", to distribute tasks. Wesson *et. al* showed[26] how many intelligent sensor devices could pool their knowledge to obtain an accurate overall assessment of the situation. The specific task presented in their work involved detecting moving entities, even though each "sensor agent" saw only a part of the environment. They reported results using both an hierarchical organization, as well as an "anarchic committee" organization, and found that the latter was as good as, and sometimes better than the former. Cammarata *et. al* [1] present strategies for cooperation by groups of agents involved in distributed problem solving, and infer a set of requirements on information distribution and organizational policies. They point out that different agents may have different capabilities, limited knowledge and resources, and thus differing appropriateness in solving the problem at hand. Lesser *et. al* [14] describes the FA/C (functionally accurate, cooperative) architecture in which agents exchange partial and tentative results in order to converge to a solution.

Joshi [12] presents a learning technique which enhances the effectiveness of such coordination. It combines neuro-fuzzy techniques with epistemic utility criterion.

2.1 SciAgents

The software systems for scientific computing reflect the underlying complexity of the intricate, interacting mathematical models. The designers of the models and the consumers of the numerical results are usually application scientists or engineers. With the increasing sophistication of high performance computing (HPC) hardware and numerical software, it is becoming difficult for them to develop these complex software systems. Recognizing this problem, teams of experts have developed general problem solvers, each one applicable to one of 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 visu-

alization, etc.. A good example for such solvers is //ELLPACK [11, 20] which is designed to handle Partial Differential Equations (PDE) models.

It is generally accepted, however, that universal solvers for the complex heterogeneous models described earlier 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 reuse of well tested software, modularity and flexibility, low cost, etc. We are developing an agent-based, distributed, collaborative environment *SciAgents* [2] which addresses these issues and allows disparate pieces of scientific software to collaborate in solving a problem.

3 Design and Architecture of SciAgents

To develop a complex scientific computing system, we need to develop strategies for coordination amongst heterogeneous agents. These strategies have to operate under the "black box" constraint, i.e. make no assumption of the internal mechanisms of other agents. In this section we present the design and the architecture of *SciAgents.* Specifically, we present a scenario where the agents are divided into two broad types - solvers and relaxers. Each solver agent is attuned to solve some particular problem, and the relaxer agents try to "mediate" between the solvers to bring their solutions to conformity at the interface regions. Internally, of course, each solver and relaxer agent can be different in how it achieves these broad goals. We show in this section how, in a multi-agent system, these agents adapt to their changing environment and each others activities by exchanging messages. The syntax and semantics of these messages are given in appendix A. When we need to be more specific, we use as examples PDE based models. Such models are among the more complex that arise in scientific computing. Various components of this system have already been developed, we are now working on building a *SciAgents* prototype. We first introduce the user's view of *SciAgents* and then concentrate on the communication and the coordination between the agents.

3.1 User's Abstraction of the Architecture

SciAgents is suitable for multiple-domain models with the following properties:

- Physical phenomenon consisting of a collection of simple, connected parts.
- Each part obeys a single physical law (that can be modeled by a simple mathematical submodel) locally in a simple subdomain.

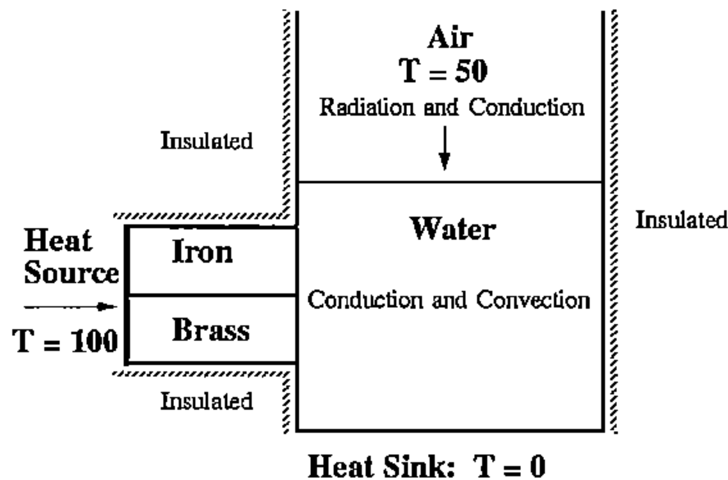

Figure 1: A simple heat flow problem.

- 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.

In [2, 3, 4] we show how model solvers (for a single PDE defined on a single domain, for example) like //ELLPACK [11] can be used to solve the submodels. They compute locally and interact with the neighboring solvers, so they are natural *local problem solver* agents. 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 allows seamless computation of the global solution. The overall behavior of the agents is based on the interface relaxation technique. For PDE based models it is described in detail in [3, 4, 16]. It uses the physical relations between the parts of the system modeled by mathematical formulas involving the solutions of the submodels in the individual neighboring subdomains and their derivatives. 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 [4, 7]. The interface relaxation technique can be described briefly as follows.

Step 1. Choose initial information as boundary conditions to determine the submodel solutions in each subdomain.

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

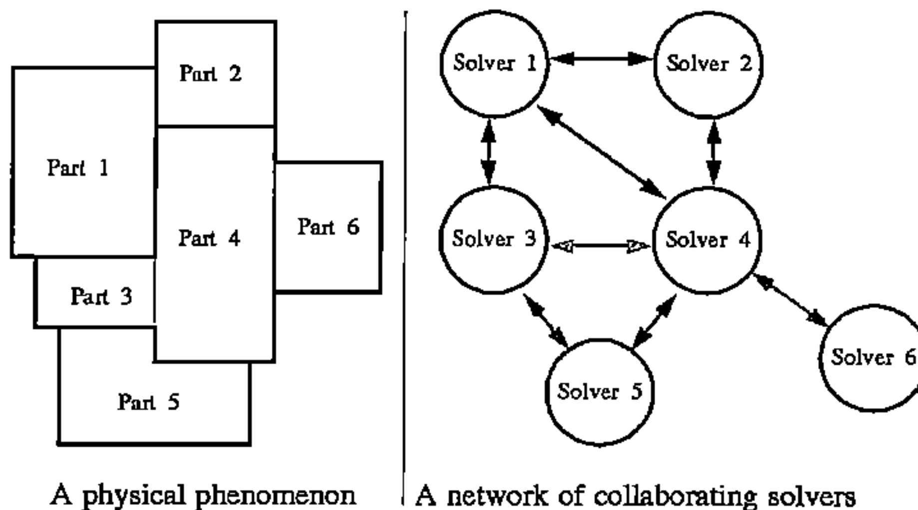

Figure 2: A schematic of the geometry of a physical phenomenon with six parts is shown on the left. The mathematical model of this physical phenomenon can be represented by a network (right) of six solvers and eight interface conditions represented by arrows.

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 2 and 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 problems.

We now describe how the users build a prototype using SciAgents and their view of the software architecture. Consider the problem in Figure 2 which contains several subdomains with a single submodel in each of them. The user breaks down the global geometry into simple subdomains with simple models. Then the interface conditions have to be defined in terms of the subdomain solutions and their derivatives. This preliminary work is done through the user interfaces of the individual solvers, but the *SciAgents* system helps coordinate this process. Then, a *network of computing agents* is created with two major types of agents - *local problem solvers* and *mediators.* 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 framework provides a natural and convenient way to hide the details of the actual algorithms and software involved in the problem solving. Users need not know the internals of the software parts, or how they are pieced together.

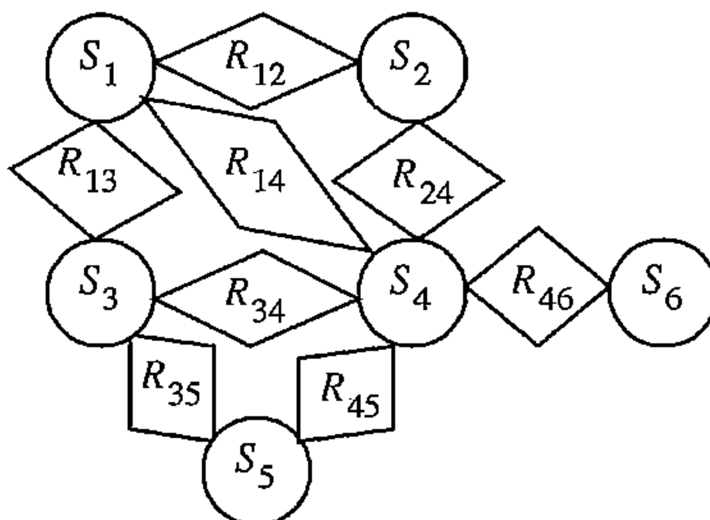

Figure 3: The user constructs a network of cooperating computing agents: solvers and relaxers. The network for the problem of Figure 2 is shown with six solvers, S_i, and eight relaxers, R_{ij}.

The user constructs the proper network of computing agents by simply *instantiating* various 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. The user is provided with an *agent instantiator* which displays information about the templates and creates active agents of both kinds, capable of computing.

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. The agents *actively* exchange partial solutions and data with other agents. 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, etc. All this is done without user involvement. In a way, by instantiating the individual agents (concentrating on the individual subdomains and interfaces) the user builds the highly interconnected and interoperable network that will solve the problem by *cooperation* between individual agents.

The user's high-level view of the *SciAgents*' architecture is shown in Figure 4. There is a global communication medium which is used by all entities called a *software bus*[25]. The agent instantiator communicates with the user through the user interface builder 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

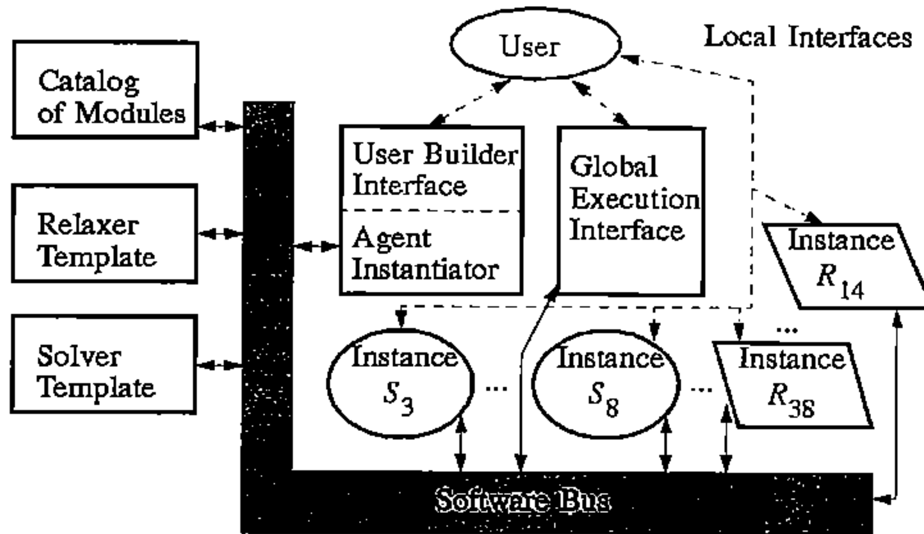

Figure 4: Software architecture of *SciAgents*: user's view. The user initially interacts with the User Interface Builder to define the global PDE problem. Later the interaction is with the Global Execution Interface to monitor and control the solution of the problem. Direct interaction with individual solvers and relaxers is also possible. The agents communicate with each other using the *software bus.*

have their own local user interfaces to interact with the user. The order of instantiating the agents is not important. 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 built in synchronization, a result of the agents adapting their behavior, is an advantage of *SciAgents.*

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 global model.* We can easily imagine a situation when the global problem is very large. Different specialists may model different parts. 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 relaxation schemes along the "borders" between different users.

3.2 Inter-agent Cooperation and Coordination of the Solution Process

Setup of Computations. One of the goals of *SciAgents* is to allow the user, who has minimal expertise in computer science, to construct the software system that will solve the mathematical model. This is achieved in part by requiring the user to provide only the functional (mathematical) specification of the problem (subdomains, PDEs, boundary and initial conditions, interfaces and interface conditions between neighboring subdomains). The agents, however, need lots of additional data, parameters, and configuration values in order to proceed with the solution process. These include:

- computational parameters for the single-domain problem the local solvers have to solve at each iteration – discretization methods for the domain and the equation, grid/mesh sizes and configurations, linear solvers, etc.;
- set of algorithms related to the interface relaxation technique that are applied by the relaxers;
- parameters depending on the hardware resources of the network.

The last bullet needs additional explanation. The network will normally have many resources available for solving a particular problem. The agent instantiator will try initially to distribute the agents evenly among the appropriate computers. However, it has very little information on which 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.

All the above parameters need to be deduced automatically by the corresponding solvers and relaxers. Solver agents contact *PYTHIA* for this purpose. *PYTHIA* [10, 13] is a system to automatically obtain the data and these parameters. 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. For example, the solvers ask an available *PYTHIA* agent for a recommendation for each of the required parameters given the equation, the domain, and the desired accuracy. *PYTHIA* 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.

Inter-agent Communication and Agent Architecture. In *SciAgents* at the highest level communication is done using the Knowledge Query and Manipulation Language (KQML [5, 6]) from ARPA's knowledge sharing initiative. 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 [25]. 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 software architecture of the local problem solver agents reflects our desire to reuse 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 – our aim is to make it relatively easy to add new solver agent templates with different core solvers to the set of templates in the agent instantiator's database. In one agent network the user may include solver agents obtained from different simple-problem solvers.

The architecture of the relaxers facilitates the even distribution of the computations and leads to an efficient implementation of the computational model. 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 in Figure 5. 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 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)

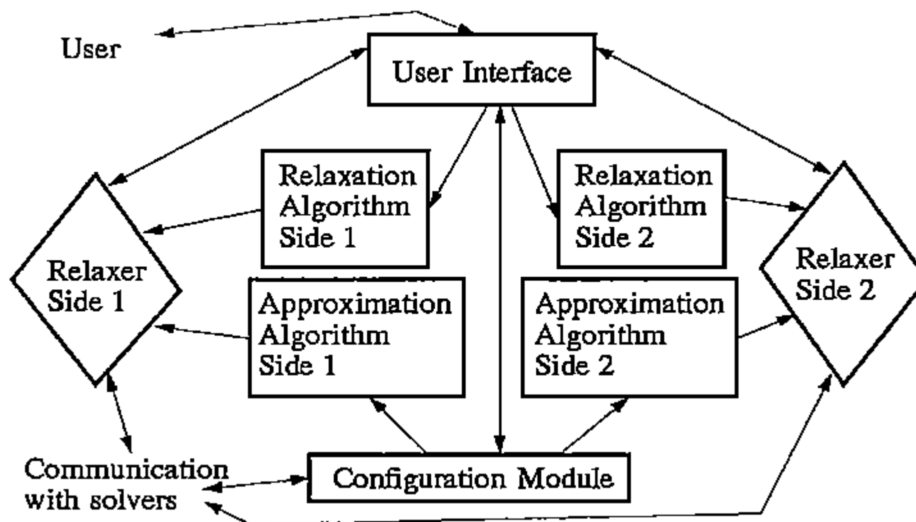

Figure 5: Software architecture of a relaxer agent. The relaxer agent is divided internally into two subrelaxers — each subrelaxer controls and supplies data to and from one solver on one side of the interface. Each subrelaxer uses its own relaxation and approximation algorithms and it communicates relatively 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).

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 three actual agents with significantly overlapping goals and a somewhat centralized control.

Figure 6 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 7 where each rectangle represents a computing unit. All computing units use the software bus as the communication medium. Each computing

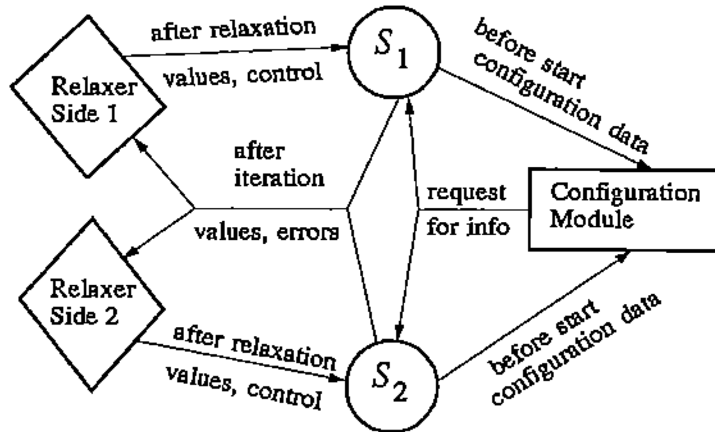

Figure 6: Relaxer agent's communication with the solvers. After the initial exchange between the solver agents and the configuration module, the information flow is very simple. In the direction from the relaxer to the solvers it can be entirely split between the two subrelaxers and their solvers. In the opposite direction the data has to be delivered to both subrelaxers.

unit has a message 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 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.

Coordination of the Solution Process. The format and the semantics of all inter-agent and some intra-agent messages in *SciAgents* are given in Appendix A. The algorithms governing the behaviour of the computing agents are shown in Figures 8, 9, 10, and 11. We denote the interface pieces of a subdomain boundary by Γ_i, the local subdomain solution by $u(x)$, the grid/mesh points on either side of a given Γ_i by P_{ij}, and the values of the boundary condition at such point by $bc(P_{ij})$. Note that these algorithms only illustrate the process of solution, and do not detail the messages exchanged between agents, nor their responses to the user's requests. Even though such messages play an important role in the control and management of the computations, they arrive and are serviced asynchronously with respect to the flow of the computations for any given agent. For simplicity, we present the relaxer algorithm as a single-entity algorithm, even though it consists of three separate processes.

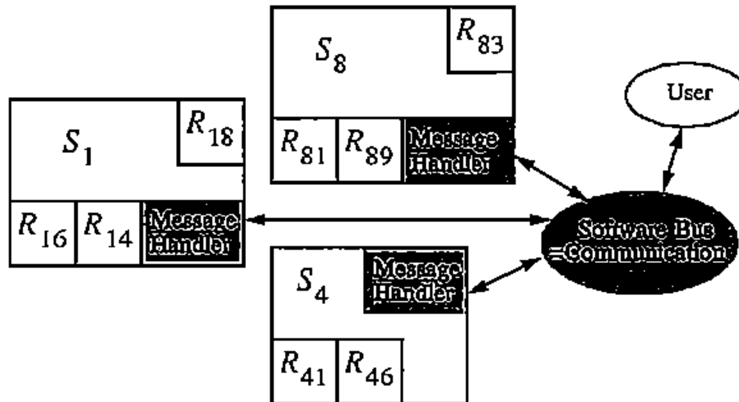

Figure 7: Software architecture of *SciAgents*: designer's view. Each rectangle represents a computing unit. There is a single subdomain solver running on one computing unit and it has all relevant parts of the relaxers for its interfaces "attached" to it. The two subrelaxers can be split between the 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.

```
Display user interface if required;
Receive the user definition of the local subdomain problem;
Ask PYTHIA agent for the missing parts of the specification;
Determine the interface pieces Γ_i of the boundary;
for (each Γ_i)
    Determine the set of points P_ij which its relaxer will use;
end for
do /* this loop is given in more detail in Figure 10 */
    Get the current boundary values bc(P_ij) for all interfaces;
    Compute local subdomain solution u(x);
    Send the relevant parts of it to all relaxers;
while (not converged)
Send u(x) to the global execution interface;
```

Figure 8: Pseudo code for the solver agent's actions after it has been started by the agent instantiator. The do loop (the algorithm for a single local iteration) is further detailed in Figure 10

```
Receive the geometry of Γ_i from the two solvers;
Display user interface;
Receive the user input on the relaxation formulas/algorithms and accuracy;
Determine the approximation capabilities of the solvers;
Determine the coordinates of all P_ij on Γ_i;
Inform the solvers of the P_ij s;
do /* this loop is given in more detail in Figure 11 */
    Calculate bc(P_ij) for both solvers;
    if (not yet converged locally)
        Send them to the solvers;
    else
        Inform the global execution interface about local convergence;
    end if
    Get the necessary u(P_ij) values from the solver agents;
while (not converged)
```

Figure 9: Pseudo code for the relaxer agent's actions after it has been started by the agent instantiator. The do loop (the algorithm for relaxing the boundary conditions after each subdomain iteration) is further detailed in Figure 11

```
for (each Γ_i)
    if (the relaxer has stopped the computations locally)
        Use the boundary condition bc(P_ij) from the previous iteration;
    else
        Get bc(P_ij) from the relaxer;
        /* wait for them if necessary */
    end if
end for
Compute u(x);
for (each Γ_i)
    Compute u(P_ij);
    /* may be done by the solver directly, by approximation formula, or by data
      sent to the relaxer routines */
    Send them to the relaxer;
end for
if (converged for all Γ_i)
    Stop computations; /* termination criterion reached */
    Listen for new commands from the relaxers to resume;
else /* no local convergence reached yet */
    Continue iterations;
end if
```

Figure 10: Pseudo code for a single iteration from the solver agent's point of view. The body of the do loop from Figure 8 is given in greater detail, as well as the computation of the loop terminating criterion

```
for (each side of Γ_i)
    if (local convergence reached)
        Do not wait for u(P_ij) from the solver;
        /* if such arrive, take them into account in the next local convergence
          check */
    else
        if (initial iteration)
            Initialize boundary condition;
        else
            Compute bc(P_ij);
            /* i.e., apply the relaxation formula */
        end if
        Send bc(P_ij) to the solver;
        Wait for u(P_ij) from the solver;
    end if
end for
when ((u(P_ij) arrive) or (local convergence and needed u(P_ij) arrive))
    Estimate the current error;
    if (converged)
        if (converged previously)
            Do nothing;
        else
            Send ''local convergence'' message to solvers;
            Inform the global execution interface;
        end if
    else
        if (converged previously)
            /* resume computations */
            Inform the global execution interface;
        else
            Check global termination criteria;
        end if
    end if
end when
```

Figure 11: Pseudo code for a single iteration from the relaxer agent's point of view. The body of the do loop from Figure 9 is given in greater detail, as well as the computation of the computation terminating criterion

Next, we discuss some important aspects of the cooperation between the agents during the solution process, and then we consider in detail an example of the communication between the agents in a sample software system built using *SciAgents*.

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 and the computations may be suspended locally if local convergence is achieved.

The local computations are governed by the relaxers which collect the errors after each iteration and, when the desired accuracy is obtained, *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 might 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 – 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.

Let us consider now as an example a part of the network shown in Figure 3. In the rest of this section we follow the actions of the solvers S_3, S_4, and S_5, and the mediators R_{35} and R_{45} as well as the messages

issued by them and by the global execution interface (GEI). We abbreviate the message formats here; their precise arguments and semantics can be found in Appendix A. When talking about the two subrelaxers of a relaxer agent R_{ij} we denote them R^i_{ij} and R^j_{ij}. The messages are given in term of the name performative and its content, which is an *S-KIF* statement.

Initially, the user instantiates the agents using the agent instantiator. The solvers use their user interface to obtain the problem specification from the user. If necessary, they send an `ask_one(<solution parameters>)` query to the available PYTHIA agent which replies with the requested data (after consultation with other PYTHIA agents). To give a better idea of the *SciAgents* inter-agent communication we show in more detail this message and the reply of the PYTHIA agent.

```
(ask-one :sender S_3
         :content(get_solution_parameters(<format>,<query code>,
                                          <problem specification>))
         :receiver PYTHIA_1
         :reply-with solution-parameters
         :language S_KIF
         :ontology PDE-solving)

(reply  :sender PYTHIA_1
        :content(set_solution_parameters(<format>,<query code>,
                                         <solution parameters>))
        :receiver S_3
        :in-reply-to solution-parameters
        :language S_KIF
        :ontology PDE-solving)
```

The messages provide information about the ontology (the context) of the message, the language of the content, and the query message the response is a reply to. If the receiver (e.g., the PYTHIA_1 agent) does not understand the ontology `PDE-solving` or the language `S_KIF`, it still can provide a reasonable reply message with the possible goal to negotiate a new language/ontology. At the same time, the content of the message can be as specific to the application domain (solving PDEs, in this case) as necessary for efficient communications between agents understanding the content language.

When the relaxers get instantiated they inform their solvers about their existence. R_{35} sends `tell(<relaxer_id>)` messages to S_3 and S_5 (which solvers a relaxer mediates is supplied by the user by building the network). Then the mediators use the `ask_one(<get boundary>)` query to get from the solvers the pieces of the subdomain boundaries the user has defined in each of them. The user instructs each mediator about the

particular interface it is responsible for. After the input from the user has been processed, the mediators send `tell(<interface>)` messages to the solvers to inform them of the user's decisions. They may also send `tell(<new coordinate system>)` and `tell(<new boundary>)` messages if the solver has to change its coordinate system and/or its data about the boundary.

The next message exchanged between the mediators and the solvers synchronizes the solution process. The mediators send a query `ask_one(<which interface points>)` to the solvers asking for the coordinates of the interface points for which the solvers need boundary conditions in order to start each iteration (these coordinates do not change during the solution process). They also send a `ask_one(<get approximation capabilities>)` query to obtain information about the solver's capabilities of approximating the solution at the boundary. After they get a reply, the mediators send a `tell(<solution values requested>)` message to each solver informing it of the coordinates of the points for which it has to supply the solution values after each iteration.

Now the agents are ready to start the solution process. Each of the solvers starts when it has all necessary data. In particular, the solver S_5 waits until it gets both messages `tell(< next iteration>)` from the subrelaxers R^5_{35} and R^5_{45}. They contain the new values of the boundary conditions along the corresponding interface which the solver uses in solving the local submodel in the next iteration. When the solver is ready with the solution, it sends to its mediators the messages `tell(<done iteration>)` which supplies the solution values. Each mediator collects the data from its solvers (for example, R_{35} waits for messages `tell(<done iteration>)` from both S_3 and S_5. Then it checks the interface conditions for convergence. If the required accuracy (communicated at the beginning of the computations by the GEI through a `tell(<global parameters>)` message to all mediators) is achieved (say, along the interface between S_3 and S_5), the mediator sends two types of messages – a message `tell(<done computations>)` to the GEI to inform it about the local convergence achieved, and two messages `tell(<next iteration>)` to the solvers with arguments telling them not to wait next time for new boundary conditions along this interface but to use these ones in consecutive iterations. If S_5 gets such message from R^5_{35} but not from R^5_{45}, it continues calculating more iterations sending back messages `tell(<done iteration>)` to both mediators. R_{35} continues to check the interface conditions. If meanwhile all mediators report local convergence to the GEI, then it issues a `tell(<stop job>)` message to every agent since global convergence has occurred. If, however, R_{35} detects error greater than the required accuracy, it issues to S_3 and S_5 a `tell(<resume waiting>)` message. It tells the solvers to resume waiting for new boundary conditions from the mediator since the convergence is no longer observed. Also, a message `tell(<resume computations>)` is sent to the GEI to inform it that there no longer a local convergence along this interface.

In the preceding example, we have skipped some message exchanges designed to better tune the computations in the interests of readability.

4 SciAgents Implementation and Software Reuse/Evolution Issues

We describe the current implementation of *SciAgents* and use it as an example for successful reuse, evolution, and incorporation of existing complex stand-alone software systems into *SciAgents* . The highest level communication in *SciAgents* is done using the public domain KQML implementation KAPI by EIT Corp. and Lockheed, Inc. Since this KQML implementation also provides agent locator facilities (through registering with and querying an HTTP server) we rely on it for the low-level integration of the *SciAgents* environment – as soon as the agents are instantiated by the agent instantiator, they register with the "locator server" and query it frequently to find out whether their peers are already available. This allows the agent instantiator to be flexible with respect to the computers on which it instantiates the agents. The relaxer implementation separates the interfacing, communicating, etc. capabilities of the agents from the relaxation formula and even in part from the algorithm. Thus, we are able to use one and the same relaxer template to instantiate relaxers with different relaxation formulas (the formulas have to be available, i.e., programed, before the instantiation). Each relaxer must be assigned to a part of the interface that is considered a "single" curve. Naturally, two subdomains' interface may consist of several pieces – in this case, there are several relaxers between them – each relaxing one piece of the interface. Note that this requirement does not expect any additional knowledge from the users – they have to know the exact form of the geometry of the subdomains when they define the local subproblems. The only real restriction is that the boundaries of neighboring subdomains have to be described consistently – if the local solver on one side of an interface thinks that the boundary consists of three pieces, then the local solver on the other side of this interface should also have it divided into the same three pieces.

Probably the most interesting aspect of the *SciAgents* implementation are the local solvers. As we mentioned before, they consist of a core and an agent wrapper. One of our major goals in designing *SciAgents* is to propose and investigate a feasible way to reuse and evolve the vast amount of scientific problem solving software that is available, tested, and popular among the application domain users. This is achieved by using such systems as cores of the solver agents. The important issue is the ease and cost of implementing the wrapper that is required for an existing problem solver to run as a local solver agent in *SciAgents* . Our work with //ELLPACK indicates that the developers of the existing system can design and implement the wrapper for a very small fraction of the time and cost spent for the existing system. In the rest of this section we describe in detail how the wrapper //ELLPACK [11] has been designed and implemented. We also give some insights on the effects on the entire system produced by our implementation.

//ELLPACK is the only solver template available in the current implementation of *SciAgents* . However, //ELLPACK contains many actual PDE solvers and two runs of it in different modes can differ substantially

from each other – therefore, the users can get the different solvers they need in the different subdomains according to the physical characteristics of the local submodels. The agent wrapper of //ELLPACK is less than 1000 lines of code long, while //ELLPACK itself contains close to a million lines of code. No more than 300 lines of code have been changed in the original code of //ELLPACK in order to provide the wrapper with all necessary data. The wrapper communicates with //ELLPACK through files containing the necessary data and it is able to run the //ELLPACK user interface and its computational part separately. The major part of the wrapper translates data to and from the S_KIF format, and receives and sends the appropriate messages to other agents. When the user instantiates a local solver, the agent instantiator starts up a copy of the wrapper. The wrapper then invokes the //ELLPACK user interface in order to obtain the subproblem definition from the user. During the problem specification, the //ELLPACK interface outputs various data to files. It includes the geometry of the interface parts of the boundary (how many pieces, the definition of them, etc.), the coordinates of the mesh/grid points on the interface (the handling of the adaptive grid is beyond the scope of this paper), and the approximation capabilities of the solver in the mode the user is running it in. After the problem is defined, an //ELLPACK program [20] is generated which is then translated into a Fortran program containing the information where (on which hardware platform) the resulting executable will be run. //ELLPACK is able to use a variety of hardware platforms, including several parallel and distributed computer architectures. No restrictions on the choice of a platform to run the local solver has been made. Thus, one subproblem may be solved on a single SPARC workstation, another – on an Intel Paragon, and third – on a cluster of workstations. The choice of the platform is left to the user or to the available PYTHIA agent. Then the program is compiled and linked to produce an executable, after which the //ELLPACK user interface exits and the wrapper takes over. It reads and parses the files left from //ELLPACK and communicates the necessary data (such as the coordinates of the boundary mesh points) to the relaxers. When the relaxers send it the new boundary conditions for the interfaces, the wrapper prepares new files for the //ELLPACK executable. After all data is ready, the wrapper invokes the executable. While running, it accesses the data and computes the solution into a file. When the executable exits, the wrapper accesses the solution data and sends it to the relaxers, waiting for new boundary conditions from them. Thus, at the next iteration, no new compilation and user actions are necessary, since the same executable is run by the wrapper. This approach leads to greater efficiency of the global solution process but does not allow dynamic migration of solver agents across machines. The ability to handle migration is a part of our ongoing research. Naturally, if the users wish to change something during the computations, they may do so by invoking the //ELLPACK interface again. Then, a new executable is compiled and an attempt is made to continue the computations. In the current version, however, a change to the discretization of the subdomain results in restarting the computations locally (using as boundary conditions for the interfaces the latest solutions from the surrounding subdomains) since the mediators cannot evaluate the interface

conditions if they do not have the coordinates of the interface mesh points from both sides of their interface before each iteration starts. Other user interventions (short of changing the equation, the geometry, or the proper boundary conditions) have minimal effect on the computational process outside the solver agent. Such changes may include specifying different linear solver, different visualization routine, different hardware architecture, or different machine to run. Note that these choices, including the discretization choice, should be made only by expert users — otherwise the available PYTHIA agent advises on most of their values.

It should be clear now why developing the wrapper for //ELLPACK has been so straightforward and low-cost — the wrapper does only the necessary minimum of translating and communicating actions and the changes in the //ELLPACK code are needed only to output already available computed data into files. We envision two possible difficulties in supplying other problem solvers with agent wrappers for running in *SciAgents* . First, //ELLPACK is highly modular system. As a result, we have been able to separate its user interface, together with the preprocessing and the compilation of the //ELLPACK program from its computational parts (the executable in this case). In systems where this is not so easy, the developers might have problems with the control of the solution process and supplying the necessary data at the right moment. Second, there exist solvers that are too "closed" and locating the data that has to be output (made available to the wrapper) might not be an easy task. For example, there might not be a good way to locate and output the grid before the actual solution process starts. This problem could be solved by "extracting" the grid-computing routine from the solver and running it alone in order to produce the grid when it is needed. However, we believe that even with these potential problems in mind, the authors of an existing problem solver should be able to build a wrapper for it at low cost and for short time based on the communication and other specifications and requirements of *SciAgents* . The prospect of multiplying the use of their solver can be considered as an additional incentive to invest the necessary effort of developing the wrapper.

Appendix A *SciAgents* Communication Format

This appendix defines the context and the semantics of messages to be used for inter-agent (and, in some cases, intra-agent) communication within *SciAgents*. The messages are given in a somewhat abbreviated KQML [5, 6] format. Only the fields that contribute to understanding the message semantics are given, – others, e.g., the `:ontology` field are skipped. The content of the messages is given in *S-KIF* [8] – a knowledge exchange language for scientific computing being developed at Purdue. The `:language` field is, therefore, the same for all messages and is not included in the messages. The format of the parts in "< >" is not given in detail. While the desired functionality of the programming environment has been covered extensively, there may be a need for additional messages for the error handling and user interaction. Such messages will be added to this list as the situations are discovered, the user options are defined, and a mechanism for their handling is developed. The messages directed to agents not directly involved in the solution process (like *PYTHIA* agents) are not presented here. The messages are grouped according to the agents that service them. Within the messages serviced by a given type of agent there is no particular order although messages with common topic are likely to be found together.

The software bus [25] is a guaranteed delivery communication system, hence, the messages do not require explicit acknowlegements. In a number of cases, however, the sender needs more than just information whether its message has been delivered. For these cases, special reply messages with specific formats have been designed and they are described in the section for their receiver.

Appendix A.1 Messages Serviced by the Agent Instantiator

```
(tell :content(resume_computations(<mediator id>)))
```

Sent by the mediator's CIPs when they start the first iteration or when the computations resume after a "local convergence" message has been sent. The mediator ids are assigned by the agent instantiator when the agents are instantiated.

```
(tell :content(done_computations(<mediator id>, <end code>, <error>)))
```

Sent by the mediators' CIPs when they decide to stop the computations (at least temporarily). The end code describes the reason for stopping – local convergence achieved, total number of iterations exceeded, possible crash of some solver, etc. The error field shows the current value of the norm of the error (e.g., the maximum absolute value of the error vectors). In case of requirement for computing until global convergence, the message indicates simply local convergence.

```
(reply :content(job_done(<agent id>, <end code>)))
```

Sent by solvers and mediator's CIPs when they have killed all the subprocesses after the global execution interface has issued a "stop_job" message. The end code indicates error conditions, if any.

```
(reply :content(get_solution(<solver id>, <reply code>, <format>,
                              <solution file>)))
```

Sent by the solvers as a reply to a "get_solution" message issued by the global execution interface. The solutions are then used to present a global view of the solution by the global execution interface. The format is what the solver has been able to provide.

```
(reply :content(save(<agent id>, <reply code>, <data file>)))
```

Sent by all primary processes of the agents as a reply to a "save" message. The filename is saved in the central data file saved from the session. When the central file is loaded into the global execution interface, it starts all other primary processes with the filenames recorded in the central file.

Appendix A.2 Messages Serviced by the Solver Agent

```
(ask-all :content(get_solution(<format list>)))
```

Sent by the global execution interface in order to obtain the values of the local solution. The format list contains a list of formats ordered according to the preference of the global execution interface. The solvers are supposed to send back in "get_solution_reply" the first possible for them format from that list.

```
(ask-all :content(stop_job()))
```

Sent by the global execution interface when the agents are about to be terminated. The primary process kills all child processes and sends a "job_done" message back.

```
(tell :content(display_solution(<display mode>)))
```

Sent by the global execution interface when the latest solution is to be displayed locally. The display mode determines the frequency of the display (say, after each iteration, or once), the viewing parameters, etc.

```
(tell :content(mediator(<mediator id>)))
```

Sent by the mediators' CIPs to inform the solver about a new mediator assigned to relax one of this subdomain interfaces.

```
(ask-one :content(get_approximation_capabilities(<mediator id>)))
```

Request from the mediators' CIPs to describe the capabilities of the solver to approximate the solution values/derivatives along the interface.

```
(tell :content(solution_values(<mediator id>, <format>,
                                <list of point coordinates>, <requested data>)))
```

Gives the solver the values along the interface whose solution values/derivatives it has to supply after each iteration. The actual data supplied by the solver will depend on its approximation capabilities – e.g., if the solver can only supply the solution values for points "near" the interface then it will send a set of points for each point of the interface (in this case, the requested data field contains the minimal number of points the solver must supply).

```
(ask-one :content(get_boundary(<mediator id>, <format list>)))
```

Request from the mediators' CIPs to supply the pieces of the subdomain boundary. The list of formats ordered by preference is specified in the format list. The solver may skip the already assigned interface pieces as well as proper boundary pieces, if the user has the opportunity to define them clearly in the solver's interface.

```
(tell :content(change_coordinate_system(<mediator id>, <transformation matrix>)))
```

Request from the mediators' CIPs to change the coordinate system of the solver. The transformation matrix is the matrix to apply to all point coordinates. The transformation is linear since we assume Cartesian systems initially in all solvers.

```
(tell :content(change_boundary(<mediator id>)))
```

Request from the mediators' CIPs to display the user interface so that the user may change some of the boundary pieces in event of non-match between the interfaces of the two subdomains.

```
(tell :content(new_boundary(<mediator id>, <format>, <old boundary>,
                            <new boundary>)))
```

Request from the mediators' CIPs to replace the old boundary piece with the new boundary. The new boundary may contain a list of pieces; the format is given in the format field.

```
(tell :content(interface(<mediator id>, <format>, <interface pieces>,
                         <subrelaxer id1>, <subrelaxer id2>)))
```

Informs the solver that the (list of) boundary pieces will be the interface governed by this mediator. The two subrelaxer ids are needed by the solver for the communication during the solution process.

```
(ask-one :content(get_interface_points(<mediator id>, <format list>)))
```

Request for the coordinates of the boundary points for which the solver will need boundary condition values.

```
(tell :content(next_iteration(<mediator id>, <boundary data>,
                              <computation status>)))
```

Sent by the subrelaxer for this solver. Supplies the new boundary data to be used in the next iteration. The computation status field commands whether the solver will wait until the new data for this interface arrives before starting the next iteration. Effectively, "no more data for this interface" means "local convergence"; if the solver gets messages with "no more data" from the mediators of all its interfaces, then the solver suspends the computations.

```
(tell :content(resume_waiting(<mediator id>)))
```

Sent by the subrelaxer for this solver to indicate resuming of the iterations. The solver is required to wait again for the "next_iteration" message from this subrelaxer after it sends the solution data from the next iteration. This message is necessary to synchronize the subrelaxers and the solver after temporary suspension of the computations.

```
(ask-one :content(more_points(<mediator id>, <format list>, <interface point>,
                              <requested data>)))
```

Sent by the mediators' CIP process and serves as a "local" "solution_values" message for the case when the set of approximation points supplied by the solver do not guarantee the required accuracy. The solver replies with two reply messages "more_points" to the two subrelaxers.

```
(ask-all :content(save()))
```

Sent by the global execution interface after a user's request to save the current state of the session. The solver saves its current state and sends back a "save" reply message.

Appendix A.3 Messages Serviced by the Mediator's CIP Process

Since the mediator is a complex agent, part of the messages serviced by the CIP process are intra-agent communication between the CIP process and the two subrelaxer processes.

Appendix A.3.1 Messages from Other Agents

```
(ask-all :content(stop_job()))
```

Sent by the global execution interface when the agents are about to be terminated. The CIP kills all child processes and sends a "job_done" message back.

```
(tell :content(assign_solver(<solver id>)))
```

Informs the mediator about a solver assigned to it. The message comes from the global execution interface.

```
(tell :content(global_parameters(<number of iterations>, <error>, <time limit>,
                                 <local mode>)))
```

Informs the mediator about some global parameters given by the user such as: the maximum number of iterations to perform, desired norm of the error, time limit to find a solution, whether to continue to perform iteration after the computations converge locally, etc.

```
(tell :content(global_stop(<stop code>)))
```

Requests termination of the computations and is sent by the global execution interface. The stop code indicates the reason for that request – usually global convergence but may also be a user request, etc. The purpose of the information in the stop code is to be displayed (if requested by the user) on the mediator's local interface.

```
(tell :content(global_reset(<reset code>)))
```

Requests resetting the computations and possibly starting over again. The reset code indicates the reason for the reset and the parameters of the reset. This message is a result of the user's intervention.

```
(reply :content(get_approximation_capabilities(<solver id>, <reply code>,
                                               <capabilities description>)))
```

Sent by the solver as a reply to a "get_approx_capabilities" message.

```
(reply :content(get_boundary(<solver id>, <reply code>, <format>,
                             <list of boundary pieces>)))
```

Sent by the solver as a reply to a "get_boundary" or a "change_boundary" message. The format field indicates the actual data format used by the solver. In the reply code, the solver indicates whether its coordinate system has been examined (and, maybe, changed) by another mediator. This will help this mediator to decide which coordinate system to change in event of non-match.

```
(ask-all :content(save()))
```

Sent by the global execution interface after a user's request. The CIP process saves its current state and issues a "save" message to the subrelaxer processes. After receiving their reply it includes their filenames in its file and sends back a "save" reply message.

```
(reply :content(save(<agent id>, <reply code>, <data file>)))
```

Sent by the subrelaxers as a reply to a "save" message by the CIP process. The CIP process uses the files as command line arguments when starting the subrelaxers by loading its session file.

Appendix A.3.2 Messages from the Subrelaxer Processes

The messages between the CIP process and the subrelaxer processes are not the only one way these processes may communicate. When the CIP starts the subrelaxers, it may supply them (through files, if necessary) with many initial items. The message interface, therefore, is needed mainly for the dynamic communication during the computations and for control.

```
(tell :contents(stop_point(<subrelaxer id>, <stop code>, <error>)))
```

Sent by the subrelaxers when a stop condition is satisfied. The condition is explained in the stop code and may be satisfied interface conditions on this part of the interface, maximum number of iterations reached, etc. Issuing this message does not indicate that the corresponding subrelaxer has suspended the computations. Only the CIP may initiate such a suspension (by issuing a "stop_iterations" message). Until the subrelaxers receive such a message, they continue to work if possible.

```
(tell :content(new_iteration_necessary(<subrelaxer id>, <error>)))
```

Sent by the subrelaxers when the new solutions communicated by the solvers do not satisfy the interface conditions. The CIP process may decide to issue a "resume_iterations" message.

```
(tell :content(iteration(<subrelaxer id>, <# iteration>, <error>)))
```

Sent by the subrelaxers after each iteration only if the user requests detailed information during the solution process.

```
(ask-one :content(more_data_requested(<subrelaxer id>, <request>)))
```

Sent by the subrelaxers when more static data is requested from the solver – e.g., more approximation points for a given interface point. The request has to go through the CIP process since only one such message must be sent to the solver. The solver then can send the reply message directly to the subrelaxers.

Appendix A.4 Messages Serviced by the Subrelaxer Processes

Appendix A.4.1 Messages from Other Agents

```
(reply :content(solution_values(<solver id>, <reply code>, <format>,
                                <solution values data>)))
```

Sent by the solver as a reply to a "solution_values" message. The purpose of the reply is to communicate static data which (probably) will not change during the iterations like formulas for computing the values/derivatives of the solution along the interface, coordinates of points "near" the interface points, etc. If such data changes during the computations, the solver sends another message of this type, even though there hasn't been a "solution value" message before it. This message also establishes the order in which the solver will communicate the values/derivatives of the solution after each iteration. It goes to the both subrelaxers (since both will receive the data from the solver).

```
(reply :content(get_interface_points(<solver id>, <reply code>,  <format>,
                                     <list of interface points>)))
```

Sent by the solver as a reply to a "get_interface_points" message. This is part of the static data supplied by the solver before any iterations begin. The order of the interface points is the order the subrelaxer should supply the values. This message is only to the proper subrelaxer.

```
(tell :content(done_iteration(<solver id>, <end code>, <format>,
                              <solution data>)))
```

Sent by the solver to communicate the solution data after each iteration to both subrelaxers. The end code may indicate problems and errors – e.g., non-convergence of the solver's iterative method, user's intervention, etc. This message is sent every time an iteration is performed by the solver, no matter whether the iterations have been suspended locally by the CIP (and, subsequently, by the subrelaxers).

```
(reply :content(more_points(<solver id>, <reply code>, <format>,
                            <interface point>, <list of point coordinates>)))
```

Sent by the solver as a reply to a "more_points" message. Supplies the new points' coordinates. They replace the old set of points for the interface point in question. Both subrelaxers receive the message.

Appendix A.4.2 Messages from the CIP Process

```
(tell :content(stop_iterations()))
```

Requests suspension of the computations (possibly a temporary one). The subrelaxers indicate the request to the solvers in the "next_iteration" computation status field as "no waiting".

```
(tell :content(resume_iterations()))
```

Requests resuming the iterations (at the beginning and after a temporary suspension). The subrelaxers issue "resume_waiting" messages.

```
(tell :content(parameter(<parameter id>, <parameter value>)))
```

Communicates changes in the solution parameters – acceptable error, maximum number of iterations, time limits, interface conditions, relaxation formula, etc. Basically, this message reflects the user input into the global execution interface and into the mediator's user interface (run by the CIP process).

```
(tell :content(reset_iterations(<reset code>)))
```

Requests resetting of the computations (a consequence of the "global_reset" message). Both subrelaxers reset their parameters and send to the solvers initial boundary conditions (if the solution process is to restart from the beginning). The solvers need not know that the computations have started again – they simply compute iteration after iteration.

```
(ask-one :content(save()))
```

Requests saving of the current state into a file for later loading. After saving its state, the subrelaxer sends back a "save_reply" message.

References

[1] 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.

[2] T. Drashansky, A. Joshi, and J.R. Rice, *SciAgents – An Agent Based Environment for Distributed, Cooperative Scientific Computing*, Proc. 7th Intl. Conf. Tools with Artificial Intelligence (Los Alamitos, CA), IEEE Computer Soc., 1995, pp. 452–459.

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

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

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

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

[7] E. Gallopoulos, E. Houstis, and J.R. Rice, *Computer as Thinker/Doer: Problem-Solving Environments for Computational Science*, IEEE Computational Science and Enginerring 1 (1994), 11–23.

[8] M. R. Genesereth and R. E. Fikes, *Knowledge Interchange Format, Ver. 3.0 Reference Manual*, Comp. Sci. Dept., Stanford University, 1992.

[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. Houstis, S. Weerawarana, A. Joshi, and J. R. Rice, *The PYTHIA project*, Neural, Parallel, and Scientific Computations (S. K. Aityan et al., ed.), Dynamic Pub., 1995, pp. 215–218.

[11] 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.

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

[13] A. Joshi, 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.

[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, *Collaborating PDE Solvers*, Appl. Num. Math 10 (1992), 279–295.

[16] Mo Mu and J. R. Rice, *Modeling with Collaborating PDE Solvers — Theory and Practice*, Computing Systems in Engineering 6 (1995), 87–95.

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

[18] A. Quarteroni, F. Pasquarelli, and A. Valli, *Heterogeneous Domain Decomposition: Principles, Algorithms, Applications*, Proc. of Fifth Intl. Symp. Domain Decomposition Methods for PDEs (Philadelphia) (D. Keyes et al., ed.), SIAM Publications, 1992, pp. 129–150.

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

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

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

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

[23] 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.

[24] 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.

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

[26] 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.

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