1995

On Learning and Adaptation in Multiagent Systems: A Scientific Computing Perspective

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:
95-040

On Learning and Adaptation in Multiagent Systems: A Scientific Computing Perspective

Anupam Joshi
Tzvetan Drashansky
John R. Rice
Sanjiva Weerawarana
Elias Houstis
Computer Sciences Department
Purdue University
West Lafayette, IN 47907

CSD-TR-95-040
June, 1995

On Learning and Adaptation in Multiagent Systems: A Scientific Computing Perspective*

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

Systems with interacting agents are now being proposed to solve many problems grouped together under the "distributed problem solving" umbrella. For such systems to work properly, it is necessary that agents learn from their environment and adapt their behaviour accordingly. We investigate such systems in the context of scientific computing. The physical world consists of interacting system, and its overall behaviour emerges from the interacting local behaviours of its constituents.In this paper we present a system which uses a combination of neuro–fuzzy learning and static adaptation to coordinate the activity of multiple agents. An epistemic utility based formulation is used to automatically generate the exemplars for learning, making the process unsupervised. We illustrate how these techniques can be used to convert a standalone, single agent system into a collaborative, multiagent one, and present some results from a preliminary implementation. We also present the design and architecture of a multiagent system, named *SciAgents*, for cooperative and distributed scientific computing.

1 Introduction

We consider multiagent 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 increasingly complex. Heretofore, scientific computing systems have been developed as standalone 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 behaviour of a physical system is a result of these interactions. Thus we are motivated to explore modelling complex physical phenomenon by the activities of multiple, autonomous software agents.

We believe that this approach is natural and direct. It is facilitated by the existence of a multitude of standalone scientific problem solving agents that can effectively model and solve for the behaviour 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

*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

Problem Solving Environments [11]. 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. The interactions between the problem solving agents are handled by mediator agents whose function is to enforce the interaction (or interface) conditions that are familiar in scientific computing.

For such agent based systems to be developed, there is a need to study the modalities and mechanisms used by individual agents in interacting with each other and in reacting to their environment. Since the environment is affected by the activities of multiple autonomous agents, it seems evident that an interaction strategy that adapts to the changing circumstances would be better than a static, non adaptive one. It also seems intuitive that the adaptive behaviour would occur as a product of *learning*. Intuition, however, is not always correct, and it is our thesis that such is the case here. Specifically, we will argue that multiple strategies are needed for efficient coordination. Some of these strategies use learning, others follow predetermined formats. We thus propose a combination of nativist and empiricist approaches to the problem. We should perhaps elaborate here on our use of the term agent, since it has been much (ab)used in literature. To paraphrase what Shoham said in his seminal work[34], "agenthood lies inthe mind of the programmer". For us, agents are software (and hardware) systems that were designed to solve some task in a standalone fashion. Thus they accept some input data, and produce specific results. The aim of our research is to allow such disparate systems to collaborate to solve some large problem by solving components of it. This should not be confused with a simple partioning of the tasks, for the solution of the individual components in our case requires interactions amongst the solvers.

We begin this paper by providing a brief summary of some of the related work done in the area of multiagent coordination. We also provide some basic information about the specific scientific computing environment that we are transforming from a standalone to a multiagent system. We detail our overall coordination strategy, and its three components in section 3, and discuss results from a preliminary implementation in section 4. In section 5 we introduce the design and architecture of a Multidisciplinary Problem Solving Environment that we are currently developing.

2 Background and Related Work

Many agent-based systems have been developed[43, 34, 40, 33, 13], which demonstrate the advantages of the agent technology. One of their important aspects is their 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* [28] which is distinct from merely using distributed computing. The expected behaviour of the simple model solvers, computing locally and interacting with the neighboring solvers, effectively translates into a behaviour 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 multiagent systems. For instance Smith and Davis [37] propose two forms of multiagent cooperation, task sharing and result sharing. Task sharing essentially involves creating subtasks, and then farming them off to other agents. In this sense, it is closer to pure distributed computation. 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[42] 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" organisation, and found that the latter was as good as, and sometimes better than the former. Cammarata and coauthors[1] espouse strategies for cooperation. They analyze the problems faced by the groups of agents involved in distributed problem solving, and infer a set of requirements on information distribution and organizational policies. They point out that in a DPS scenario, different agents may have different capabilities, limited knowledge and resources, and thus differing appropriateness in solving the problem at hand. Lesser *et. al* [19] describes the FA/C (functionally accurate, cooperative) architecture in which agents exchange partial and tentative results in order to converge to a solution.

Most of these coordination and cooperation techniques however, are static in nature. They do not "learn" as their environment changes, which in our opinion is an important aspect of coordination. We describe our coordination strategies in the context of two systems that we are developing: SciAgents and PYTHIA.

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 visualization, etc.. A good example for such solvers is //ELLPACK [14, 31] 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 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. We are developing an environment which addresses these issues and allows disparate pieces of scientific software to collaborate in solving a problem. We call this system SciAgents[4]. SciAgents will help in the realization of Multidisciplinary Problem Solving Environments (MPSEs), where individual solution software running on networked, heterogeneous software will interact to provide solutions of complex mathematical models.

2.2 PYTHIA

In the PYTHIA[15] project, our aim is to develop a system that will accept a description of a problem from the user, and then automatically select the appropriate numerical solver and computing platform, along with values for the various associated parameters. While the theoretical framework underlying PYTHIA is being developed in the generic context of scientific computing, our specific implementation deals with PDE based systems.

PYTHIA was originally conceived as a stand alone system, a single agent in this context. Its input is a description of a PDE in a special format that we have developed[1]. In response, PYTHIA produced a recommendation for the best solution method, and its confidence in ranking this method as the best. This recommendation was based on the information contained in its knowledge base regarding similar problems, and what methods had worked best for them. Clearly, in order to recommend a good method for a given problem, PYTHIA must have seen a similar problem before.

Recently, we have begun to move towards making PYTHIA a collaborative multiagent system. This is, as we shall illustrate, a more natural implementation. PDEs can be widely varying. Most application scientists tend to solve only a limited kind, and hence any PYTHIA agent they are running is likely to be able to answer questions effectively only about a limited range of problems. If there were mechanisms that allowed PYTHIA agents of various application scientists to collaborate, then each agent could share knowledge and potentially answer a broader range of questions – they could call upon the collective wisdom of all agents, as it were. In this paper, we refer to the standalone PYTHIA system as I-PYTHIA and the collaborative, multiagent version as C-PYTHIA.

3 Learning and Adaptation

In this section, we present our scheme for cooperation amongst multiple agents. We argue for a combination strategy that involves both learning from the changing environment, as well as adaptation based on *a priori* knowledge where available. We describe the strategy in the context of C-PYTHIA.

3.1 When to learn, ...

The question for C-PYTHIA is the following. Given a problem, and user imposed constraints on it (like error bounds, max time allowable), which method should be used to solve the problem, and on which hardware platform? Each individual I-PYTHIA agent is therefore learning a mapping from *(problem, constraints)* to *(method, platform, parameters)*. The parameters term here refers to things like how long computing the solution is expected to take, what error can be expected, as well as the parameters of the methods chosen. Clearly, some parameters of the solutions reflect the effort by the I-PYTHIA agent to conform to the user's constraints. Other parameters reflect confidence measures of the agent in proposing the method and hardware. In recent work, we have

[1]The format is a characteristic vector, whose elements denote various PDE properties.

applied various learning techniques to this single agent problem. Specifically, we have used Bayesian belief nets[41], neural networks[16] and fuzzy systems[29]. We now apply learning to the multiagent scenario. Specifically, what happens if the I-PYTHIA agent, to which a query has been directed, discovers that it does not have "enough" confidence in the prediction it is making ? We propose that the agent initially use some broker/agent name server to find out which other I-PYTHIA agents are available to answer queries, and send them all the query. Presumably, it will get a set of answers from the its peer I-PYTHIA agents, and will need to decide which one is "correct." It is likely that one of the PYTHIA agents knows a great deal about a certain type of problem. Thus from the answers received by an agent, it should be able to get an answer with high confidence and "learn" a mapping from the (peer) I-PYTHIA agent which is most likely to have a correct answer for this type of problem. In future, thus, it could also direct queries more effectively, rather than using a broadcast technique to seek answers.

We are using a neuro-fuzzy method of learning to this end. The reason to use a fuzzy system is that in the PYTHIA scenario, classification of problems is not crisp. For instance, the solution of a given PDE could have a singularity, and also show some oscillatory behaviour on the boundaries. Thus the given PDE would have membership (to different extent) in the classes representing "solution–singular" and "solution–oscillatory". A conventional, binary membership function would not model this situation accurately. We feel that such fuzziness will be inherent in the learning task whenever agents model complex, real world problems.

The basic idea of the method we use was proposed by Simpson[35, 36], and is a variation of the leader cluster algorithm, enhanced with the notion of fuzziness. Similar methods have been proposed by Newton[25] and Grossberg *et. al.*[2]. Simpson describes a supervised learning neural network classifier that uses fuzzy sets to describe pattern classes. Each fuzzy set is the fuzzy union of several n-dimensional hyperboxes. Such hyperboxes define a region in n-dimensional pattern space that have patterns with full-class membership. A hyperbox is completely defined by its min-point and max-point and also has associated with it a fuzzy membership function with respect to these min-max points. This membership function helps to view the hyperbox as a fuzzy set and such "hyperbox fuzzy sets" can be aggregated to form a single fuzzy set class. This provides degree–of–membership information that can be used in decision making. Thus each pattern class in the given space is represented by an aggregate of several fuzzy sets and the resulting structure fits neatly into a neural network assembly. Learning in the fuzzy min-max network proceeds by placing and adjusting the hyperboxes in pattern space. Recall in the network consists of calculating the fuzzy union of the membership function values produced from each of the fuzzy set hyperboxes. The fuzzy min-max network provides good accuracy, facilitates *single pass learning*(i.e., does not require repeated presentation of the exemplars to learn), has few parameters to tune and, most importantly, provides on-line adaptation.

However, the method as proposed by Simpson does not allow for classes that are mutually non exclusive. It would thus fail to account for a situation where more than one agent might be expected to provide a correct answer. We have enhanced the method to allow it operate under this situation. Essentially, any example which belongs to multiple classes is presented to the learning system multiple times - once for each class it belongs to, with an appropriate class label. This forces the system to include this example with full membership in multiple classes. The mechanism which adjusts the class hyperboxes is altered to allow for class overlaps where the data demands it. Initial results from this approach have been very promising, both on I-PYTHIA data, as well as classical test data such as the IRIS [9]. The details regarding this method, as well as these results can be found in [29]. We are also studying other improvements to this method using techniques from computational geometry. The method as it stands tries to form classes by using isothetic hyperboxes. Clearly, this approach is extremely naive, since it would cover regions of space that did not belong to a class. We are trying to study improvements that can be obtained by allowing the boxes to have arbitrary orientation, as well as by using hyperspheres/hyperellipsoids as our space covering primitives.

3.2 ..... what to do while learning, ...

Once an agent has learned a mapping from problem types to (other) agents which are likely to know the answer, it can direct queries to other agents appropriately. However, learning in this instance requires some known exemplars. Since the I-PYTHIA agent is assumed to be a *tabula rasa* at start, it does not have any such exemplars. The straightforward approach would be to provide the system with a list of agents to query for each problem type. While this would allow the system to direct its queries, it would still not provide labeled exemplars of the type "agent *a* provides the correct/best answer for problem type *p*." Another option would be to involve the user in the process. PYTHIA could present all the answers obtained from peer agents, and the user could select the best. Given this kind of a scenario, Lashkari *et. al.*[17] have developed a trust function which each agent uses to measure its belief that a peer agent has a correct answer. Obviously, involving the user is self–defeating in our case since the aim of the system is to allow a non–expert (in HPC) to use it. Expecting an application domain expert to chose amongst numerical solution techniques and parallel hardware is, to say the least, naive. Clearly an unsupervised learning approach is needed, which learns without any user intervention.

We propose an alternate approach formulated in terms of *epistemic utility*. We summarize the ideas here following Lehrer's presentation [18] of internal coherence and personal justification in humans. The basic idea here is that each agent has an *acceptance system*, which it uses to accept certain hypothesis as true. This system is based on two principles, obtaining truth and avoiding error. It informs an agent that it is more reasonable to accept some things than others. It enables the agent to judge which sources of information to trust and which not. Adapting Lehrer's definitions of acceptance and coherence to the agent scenario, we have

Definition 1 *Agent $\mathcal{A}$ is justified in accepting proposition $\mathcal{P}$ at time t if and only if $\mathcal{P}$ coheres with the acceptance system of $\mathcal{A}$ at t.*

Definition 2 *$\mathcal{P}$ coheres with the acceptance system of $\mathcal{A}$ if and only if it is more reasonable for $\mathcal{A}$ to accept $\mathcal{P}$ than any other competing claim $\mathcal{Q}$.*

In effect, we are saying that of all the competing hypotheses, an agent should accept the one which is more reasonable. Since we introduce time as a factor in the definitions, we leave open the possibility that an agent's acceptance system, and hence its notion of what is reasonable, will evolve over time. Note that reasonableness is not the same as the probability of being true. Consider, for instance, the following statements:

It looks like there is snow on the ground.

There is snow on the ground.

Now, it could appear that there is snow on the ground for many reasons (white confetti, TP-ing by a fraternity party, hallucination, ...) other than there actually being snow on the ground. Accepting the first statement therefore is less likely to make us err. On the other hand, it does not really tell us quite as much as the second statement, so we do not gain in the area of obtaining truth. To obtain a quantitative measure of *reasonable*–ness, we need to combine two factors, one which denotes the probability of a proposition q being true, and the other which denotes its utility. Specifically, let $U_t(q)$ denote the positive utility of accepting q if it is true, $U_f(q)$ denote the negative utility of accepting q if it is false. Further, let $p(q)$ be the probability that q is true. Then, the reasonableness of accepting q can be defined[18] as:

$$r(q) = p(q)U_t(q) + p(not(q))U_f(q).$$

Such formulations of reasonableness derive from the work of Issac Levi [21], and from the work by Neyman & Pearson on rational decisionmaking [27, 26]. The work of Neyman & Pearson deals with decision making when prior probabilities of the truth of hypothesis are not available. Consider a binary decision problem, with H_0 and H_1 as the two competing hypotheses. They suggested that in such a case, the objective of the decision making process should be to maximize the power probability (the probability of deciding in favour of H_1 when it is active) , subject to some constraint on the false alarm rate (the probability of deciding in favour of H_1 when H_0 is active). The complement of power probability, and the false alarm rate are also refered to as errors of type I and type II, respectively.

Levi's work[21, 20] is a philosophical treatise, delaing with the process of human decision making in general, and scientific enquiry in particular. He has developed a theory of epistemic utility which deals with decisionmaking by a single rational agent. It provides ideas on how an agent interacts with the environment to acquire "error free" knowledge, and how it revises its beliefs. His work provides mechanisms which enable one to develop measures of reasonableness. Reasonableness quantifies a combination of the two aims of an agent, to acquire new knowledge and to avoid error. Like much of decision theory, it is based on the notion of maximizing utility, but differs in as much as its formulation of what utility is. Based on his work, Stirling & Morrow have recently proposed schemes that are used for coordinated intelligent control [38, 39].

In the case of C-PYTHIA, each I-PYTHIA agent produces a number denoting confidence in its recommendation being correct, so $p(q)$ is trivially defined, and $p(not(q))$ is simply $1 - p(q)$. The utility is a more tricky measure. We have earlier posited that the more problems of a type that an agent has seen, the more likely it is to recommend an appropriate method, etc., for a new problem of the same type. Moreover, the reason for the epistemic module is to provide exemplars for learning. Thus the utility of accepting an agent's recommendation (and using it as an exemplar for learning) should reflect the number of problems of the present type that it has seen. In this instance, we chose to make the positive and negative utilities the same in value, but opposite in sign. This is done since the value of utility is measuring the amount of knowledge that an agent appears to have. Thus $U_t(q) = -U_f(q) = f(N_e)$, where f is some squashing function mapping the domain of $[0, \infty)$ to a range of $[0, 1)$, and N_e is the number of exemplars of a given type(that of the problem being considered) that the agent has seen. We chose $f(x) = \frac{2}{1+e^{-x}} - 1$. Note however that this is just one possible formulation of the utility measure. In fact, such a formulation suffers from a problem if it is asked to chose between two alternative hypotheses with low probabilities of being correct. Specifically, assume that the probabilities of hypothesis h_1 and h_2 being true are both p, and their utilities are u_1 and $u_2, u_1 \geq u_2$ respectively. When $p \leq 0.5$, the second hypothesis will be assigned a greater *reasonableness* measure, even though the first hypothesis has a greater utility. An alternative which does not suffer from this problem is to use $\frac{1}{f(N_e)}$ as the measure of utility of accepting a recommendation in the formulation for r whenever the probability of being true is less that half, and use $f(N_e)$ otherwise. This however tends to bias reasonableness towards truth, and away from utility.

3.3 ..... and when not to learn

An important component of scientific computation is the optimal use of the available, heterogeneous HPC hardware. We view each hardware platform as an agent, with bounded (computational) capability. Part of this capability is inherent in the hardware. The other part is a function of the amount of load on it. The platform configurations (for instance, the number of processors) can be changed, so the mapping to be considered is from *(problem, method, hardware, config)* to *time*. It would be naive, in our opinion, to throw a learning mechanism at this problem. We believe that direct learning is not required in this case, and the system can be adaptable without it.

The adaptability can be achieved by a combination of learning and modeling. PYTHIA can estimate the time required for a standard configuration of a serial hardware. It is straightforward (but not trivial) to transform this estimate for other serial hardware but we need also to transform it to parallel machines or heterogeneous clusters. Accurate analytic methods for this do not exist and learning about how all PDE solvers run on all such configurations is impractical. However, a few relatively simple models of parallel computations are useful in predicting speedup. The logP[3] and E(P)[22] models are the realistic of them. We briefly discuss how to use the E(P) model, which uses the following parameters:

- w(1) : work to solve the problem on 1 processor (provided by PYTHIA)
- f : fraction of strictly sequential work (algorithm dependent)
- P : number of processors (hardware dependent)
- E(P) : number of communication events as function of P (algorithm dependent)
- Θ : average work for a communication event (hardware dependent)

The work using P processors is then estimated to be

$$w(P) = w(1)(1 + Pf) + \Theta E(P)$$

If the processors are of unequal power, then somewhat messier formulas are required. The two algorithm or code dependent quantities, f and E(P), can be estimated from general algorithm properties or measured directly. Then, of course, they can be recorded as knowledge about the solvers for use in estimating time on parallel or distributed hardware configurations. The logP model has a more refined hardware model for Θ.

Using such an approach, the PYTHIA agent can model the behaviour of hardware agents to predict what number of processors would be optimal for a given problem on a given platform.

The speedup estimate is given in the context of a single program executing on a given platform. However, it provides the requisite framework for extension to the case where each hardware agent is actually a multiprocessing system. In such cases, the (computational) capability of the hardware

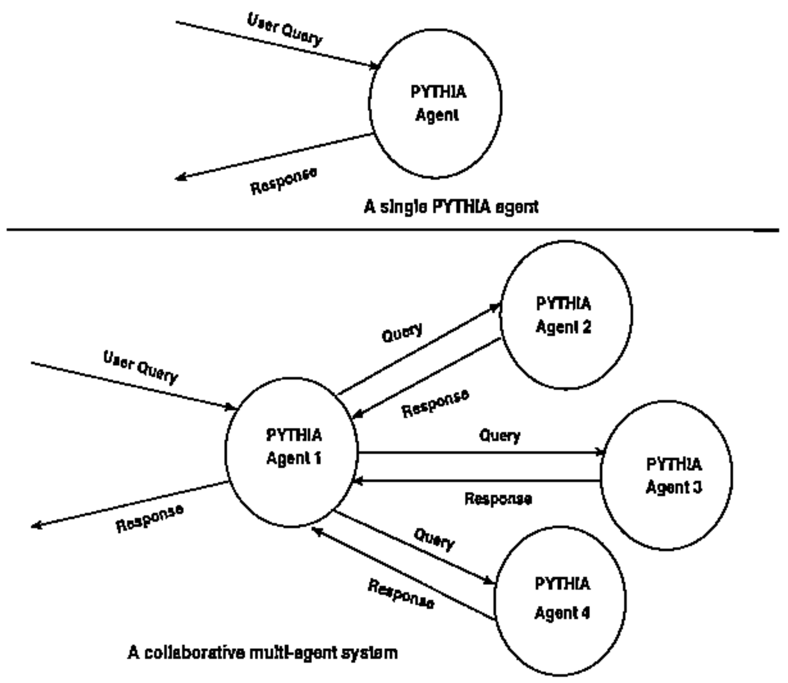

Figure 1: The architecture of multiagent PYTHIA

agents will dynamically vary. Each agent will be aware of its load at any given time, and can query other agents for their loads as well. Our multiagent system will be responsive to this dynamic behaviour, and will adapt by moving computations around. The overhead associated with migrating processes can be substantial. However, there has been recent work [23] in this area which shows promise for providing an efficient migration paradigm by using threads of control. It has been shown to be extremely effective in simulations [30].

4 Architecture and Implementation of C-PYTHIA

We have developed a preliminary implementation of the ideas outlined in the preceding sections. The overall sytem architecture is illustrated in Figure 1. We modified the original PYTHIA agent to have a "memory" of agents it knows about, the problems each agent has seen, and statistical information about the performance of each agent in regard to each type of problem the agent has been queried about. An agent can be in either a "learning mode" (LM) or a "stable mode" (SM). During the LM, an I-PYTHIA agent asks all other known agents for solutions about a particular type of problem. It collects all the answers, including its own, and then chooses the best result as the solution. In effect, each agent is using what has been described in [17] as "desperation based" communication. While in this mode, each agent is also "learning" the mapping from problem class to the agent which gave the best solution. The best solution in LM is computed by the epistemic utility formulation described earlier. After the "learning period", each agent has learned a mapping describing which agent is best for a particular type of problem. From this point on, PYTHIA is in the SM. It now switches to what we label as stable communication. In other words, it will only ask itself and the best agent to answer a particular question. It evaluates the answers received

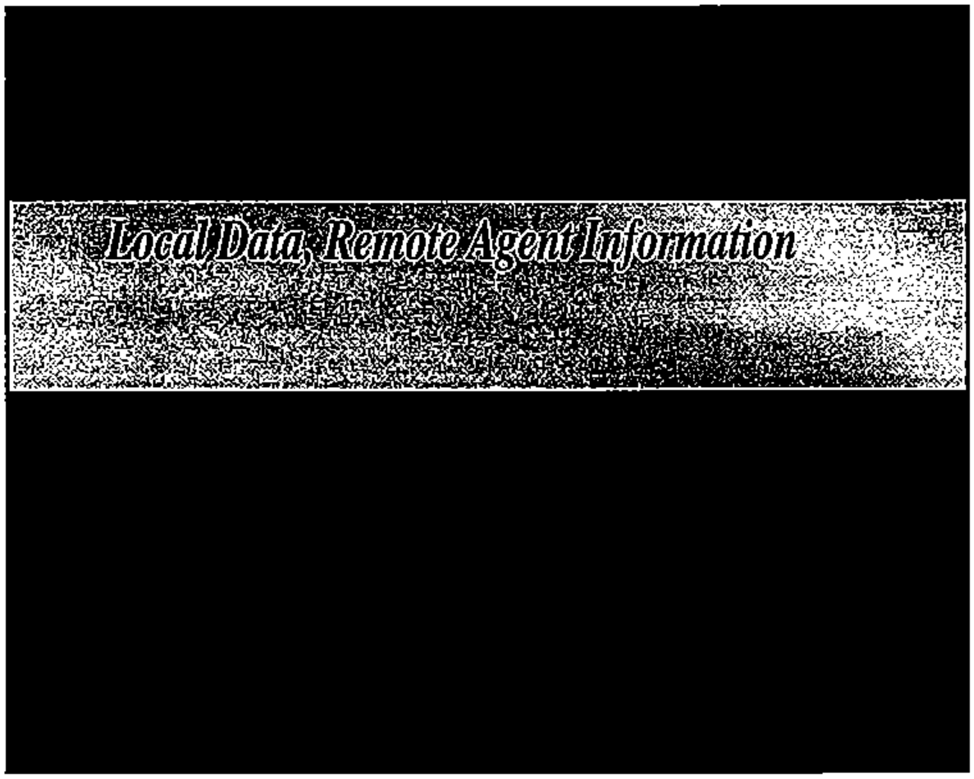

Figure 2: System Layers of each PYTHIA agent

correctly classified problems(of 23)	Mean Error	Median Error
22	0.061488	0.00001

Table 1: Results from learning the problem type to agent mapping

according to the reasonableness criterion defined earlier. If none is "reasonable" then we switch to an "exploratory" communication mode [17]. Figure 2 illustrates the layered systems of each I-PYTHIA agent that are involved in this process. If PYTHIA does not believe an agent has given a plausible solution, it will ask the next best agent, until all agents are exhausted. This is facilitated by our fuzzy learning algorithm. By varying an acceptance threshold in the defuzzification step, we can get an enumeration of "not so good" agents for a problem type. If PYTHIA determines no plausible solution exists among its agents or itself, then PYTHIA will give the answer that "was best". When giving such an answer, the user will be notified of PYTHIA's lack of confidence.

In Table 1, we illustrate the results obtained by our neuro-fuzzy learning scheme[29]. As mentioned earlier, the task was to learn a mapping from a problem to the agent best able to solve it. To this end, we chose some elliptic PDE problems from [32]. These were distributed amongst several I-PYTHIA agents, so that each agent mostly had problems of the same type. This data was then used as input to the learning scheme. As table 1 shows, we obtained extremely high classification rates, and correspondingly low values for mean and median error.

An interesting question in thie scenario described above is one of switching modes. When should an agent switch from LM to SM? Currently, we do this after an *a priori* fixed number of problems have been learned. The timing of the reverse switch is a more tricky matter which we do not address in the current implementation. Clearly, as other agents are adding to their knowledge base of "previously seen" problems, their ability to answer questions about a particular type of problem changes. In essence, by learning a mapping, we are taking an instantaneous "snapshot" of the abilities of the agents, and then using it as a "cache". We therefore need to evolve the equivalent of "cache invalidation" strategies for our learning function. There are several potential candidates which we describe next.

- **Time based:** The simplest approach is to use a time based invalidation scheme, where the agent reverts to LM after a fixed time period in SM.
- **Reactive:** Another possibility is for each agent to send out a message whenever its confidence for some class of problems has changed significantly. An agent can then chose to revert to LM when it next receives a query about that type of problem.
- **Time based Reactive:** A combination of the two approaches outlined above would send out a "has anyones abilities changed significantly" message at fixed time intervals, and switch to LM if it received a positive response.
- **Proactive:** In this approach, the agent would use the fact that switching into the exploratory mode meant that the agent it thought would best solve a problem could not. It would then switch into a learning mode by itself, without waiting for any other agent to explicitly indicate a change in its capabilities.

The backbone of any agent based system is the ability to communicate effectively among agents. In recent years the Knowledge Query and Manipulation Language, (KQML) [10] has been proposed as a medium of interagent communication. KQML was developed by the DARPA Knowledge Sharing Initiative External Interfaces Working Group especially for agent based communication. KQML based communications is based on a protocol defining performatives. Performatives are universal, in the sense that they are understood by all KQML compliant agents. The content of the performative can be in a special language that only some of the agents know. For the collaborative PYTHIA case, we use a private language (*PYTHIA-talk*) that all PYTHIA agents understand.

5 Design and Architecture of SciAgents

In our discussion so far, we have confined ourselves to homogeneous agents. However, 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* – an agent-based programming environment for scientific computing. It illustrates the application of the ideas introduced above to solving complex and heterogeneous mathematical models. 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 multiagent system, these agents

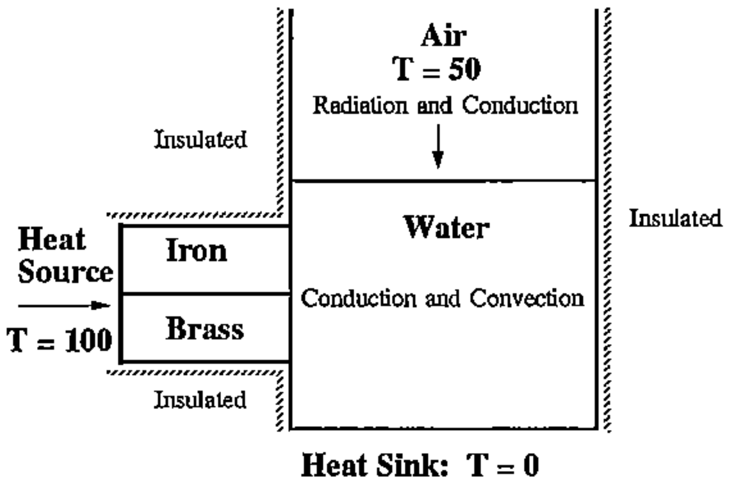

Figure 3: A simple heat flow problem.

adapt to their changing environment and each others activities by exchanging messages (speech acts). 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. 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.

We first introduce the user's view of *SciAgents* and then concentrate on the communication and the coordination between the agents in order to demonstrate their adaptive behavior.

5.1 Software Architecture and User's View of *SciAgents*

SciAgents is suitable for solving complex mathematical models that can be broken down into simple submodels with mathematically modeled interactions between them. Multiple-domain models with the following properties fall into this category.

- 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.
- 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 3. 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 [4, 5, 6] we argue that "simple" model solvers (for a single PDE defined on a single domain, for example) like //ELLPACK [14] can be used to solve the submodels. Their expected behaviour, computing locally and interacting with the neighboring solvers, effectively translates into a behavior of *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 can resolve the problems during the solution process without user intervention, and this allows seamless computation of the global solution. The overall behaviour of the agents is based on the interface relaxation technique. For PDE based models it is described in detail in [6, 5, 24]. 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 [6, 24]. 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.

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. It also allows the reuse of existing, trusted software for solving single-domain problems.

We now describe how the users build such a software system using SciAgents and their view of the software architecture of the programming environment. Consider a problem like the one shown on Figure 4. It contains several subdomains with a single submodel to solve in each of them. The user needs to break down the geometry of the composite domain 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 primarily the responsibility of the user interfaces of the individual solvers 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*. 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 4 is given in Figure 5. 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. It allows one to use *SciAgents* to solve complex problems without being proficient in computer science (numerical methods, parallel computing, etc.).

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. These templates

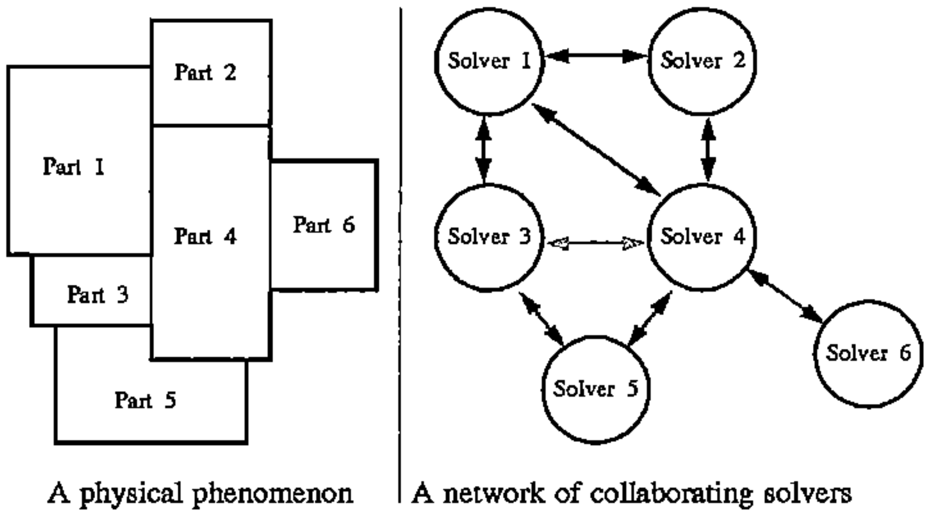

Figure 4: A schematic of the geometry of a physical phenomenon with six parts is shown on the left. The mathematical model of this physical phenomenon can be represented by a network (right) of six solvers and eight interface conditions represented by arrows.

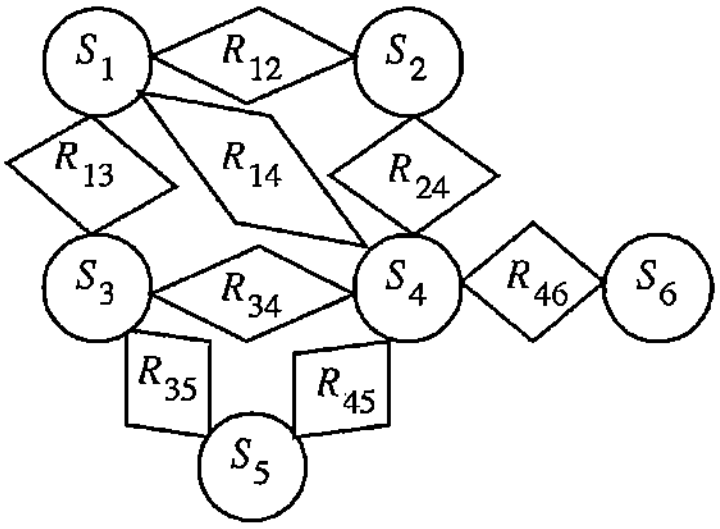

Figure 5: The user constructs a network of cooperating computing agents: solvers and relaxers. The network for the problem of Figure 2 is shown with six solvers, S_i, and eight relaxers, R_{ij}.

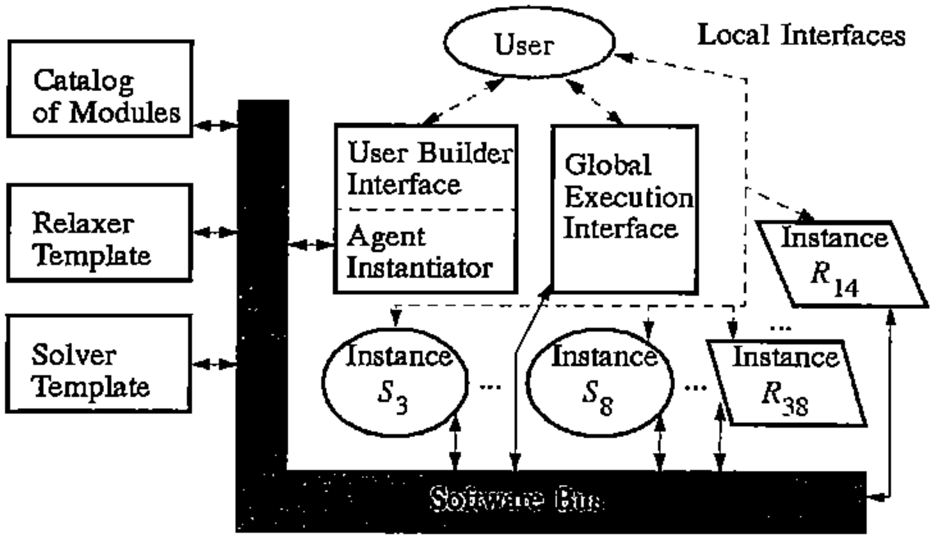

Figure 6: Software architecture of *SciAgents*: user's view. The user initially interacts with the User Interface Builder to define the global PDE problem. Later the interaction is with the Global Execution Interface to monitor and control the solution of the problem. Direct interaction with individual solvers and relaxers is also possible. The agents communicate with each other using the *software bus*.

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. 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 user involvement. 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 will solve the problem, by *cooperation* between individual agents.

The user's high-level view of the *SciAgents*' architecture is shown in Figure 6. There is a global communication medium which is used by all entities called a *software bus*[41]. 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 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 is, we believe, an important advantage of *SciAgents.* It results form each agent adapting to its environment. 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 global model.* We can easily imagine a situation when the global problem is very large. Different specialists may model different. 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.

5.2 Inter-agent Cooperation and Coordination of the Solution Process

We next describe in greater detail certain aspects of the communication and the coordination between agents during the solution process.

5.2.1 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 available hardware.

The last bullet needs additional explanation. 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 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 [30] 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 *C-PYTHIA* for this purpose. For example, they ask an available *C-PYTHIA* agent for a recommendation for each of the required parameters given the equation, the domain, and the desired accuracy. *C-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.

5.2.2 Interagent Communication and Agent Architecture

In *SciAgents* at the highest level communication is done using the Knowledge Query and Manipulation Language (KQML [7, 8]) 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 [41]. 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 – 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 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 7. 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

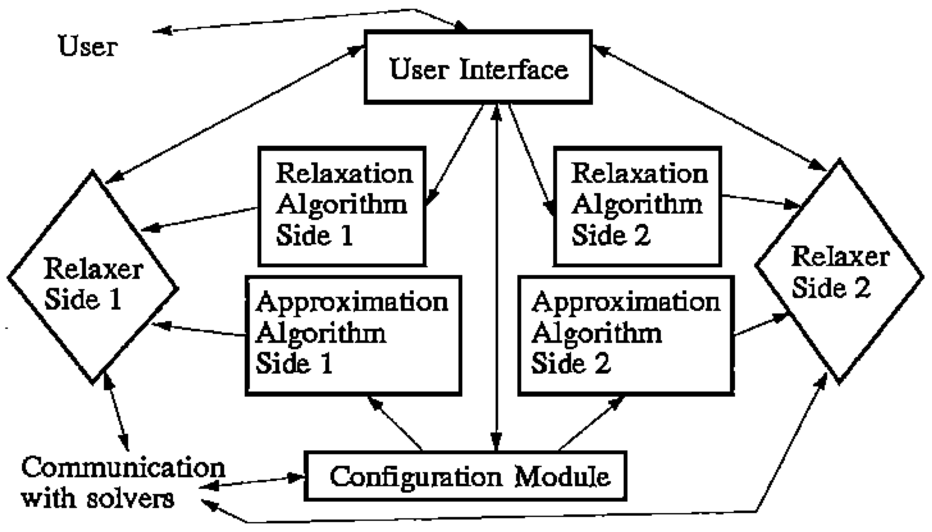
User
User Interface
Relaxation Algorithm Side 1
Relaxation Algorithm Side 2
Relaxer Side 1
Relaxer Side 2
Approximation Algorithm Side 1
Approximation Algorithm Side 2
Communication with solvers
Configuration Module

Figure 7: 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).

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) 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 8 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.

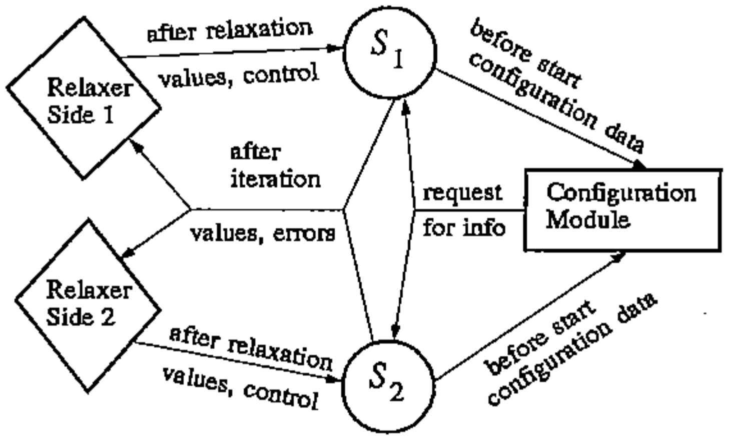

Figure 8: Relaxer agent's communication with the solvers. After the initial exchange between the solver agents and the configuration module, the information flow is very simple. In the direction from the relaxer to the solvers it can be entirely split between the two subrelaxers and their solvers. In the opposite direction the data has to be delivered to both subrelaxers.

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 9 where each rectangle represents a computing unit. All computing units use the software bus as the communication medium. Each computing 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 existance of a new agent.

5.2.3 Coordination of the Solution Process

The format and the semantics of all interagent and some intraagent messages in *SciAgents* are given in Appendix A. In this section we first 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

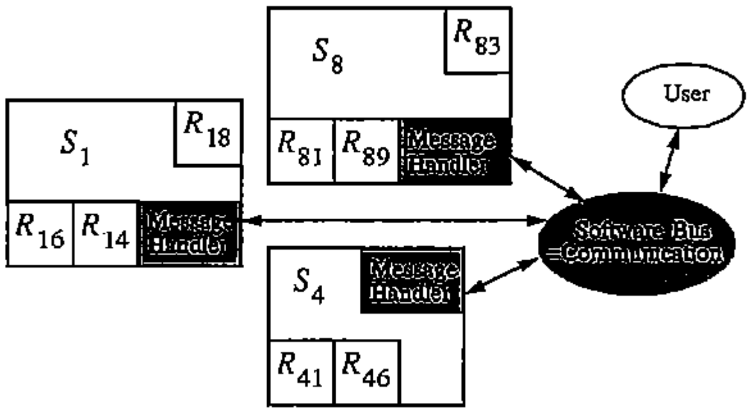

Figure 9: Software architecture of *SciAgents*: designer's view. Each rectangle represents a computing unit. There is a single subdomain solver running on one computing unit and it has all relevant parts of the relaxers for its interfaces "attached" to it. The two subrelaxers can be split between the 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.

if local convergence is achieved.

The local computations are governed by the relaxers (the solvers simply solve the mathematical models). The relaxer agents collect the errors after each iteration and, when the desired accuracy is obtained, *locally* suspend the computations and report the fact to the global 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 5. 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). 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 preceeding example, we have skipped some message exchanges designed to better tune the computations in the interestes of readability.

6 Conclusion

In this paper, we have presented a multiagent system operating in a complex, scientific computing environment. We have shown how such systems are natural for the creation of systems that would model the complexities of the physical world. That such multiagent systems have to be adaptable and involve learning is, in our opinion, evident. We have argued that learning is not the panacea that will make the difficulties of coordination in multiagent systems disappear. Specifically, we have shown scenarios from our research where learning is used, and where the system adapts based on *a priori* known models of other agents. We have also shown how *epistemic utility* theory can be used to facilitate learning in an unsupervised manner. Of course, where supervised learning is available, it can be (and is) trivially incorporated into the system. We have also outlined a framework, called *SciAgents*, that is used for creation of such multiagent systems. We have briefly outlined the kind of interagent coordination needed, and outlined a language (S-KIF) which will be used for such coordination.

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 abreviated KQML [7, 8] 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* [12] – 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 agent that services them (receives 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 [41] 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.

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.

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.

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.

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.

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.

A.4 Messages Serviced by the Subrelaxer Processes

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.

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] G. Carpenter, S. Grossberg, and S. Rosen, *Fuzzy ART: Fast stable learning and categorization of analog patterns by an adaptive resonance system*, Neural Networks 4 (1991), 759–771.

[3] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, *LogP: Towards a Realistic Model of Parallel Computations*, Proceedings of the fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (New York), ACM, 1993, pp. 1–12.

[4] T. Drashansky, A. Joshi, and J.R. Rice, *SciAgents - An Agent Based Environment for Distributed, Cooperative Scientific Computing*, Tech. Report TR-95-029, Dept. Comp. Sci., Purdue University, 1995, (submitted to Tools with AI '95).

[5] 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).

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

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

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

[9] R. Fisher, *The use of multiple measurements in taxonomic problems*, Annals of Eugenics 7 (1936), no. 2, 179–188.

[10] R. Fritzson et. al., *KQML- A Language and Protocol for Knowledge and Information Exchange*, Proc. 13th Intl. Distributed Artificial Intelligence Workshop, July 1994.

[11] 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), no. 2, 11–23.

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

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

[14] 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.

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

[16] 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.

[17] Y. Lashkari, M. Metral, and P. Maes, *Collaborative Interface Agents*, Proceedings AAAI '94, AAAI, 1994.

[18] K. Lehrer, *Theory of Knowledge*, Westview Press, Boulder, CO, USA, 1990.

[19] 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.

[20] I. Levi, *The Enterprise of Knowledge*, The MIT Press, CAmbridge, MA, USA, 1980.

[21] ———, *Decisions and Revisions*, Cambridge University Press, Cambridge, U.K., 1984.

[22] D. Marinescu and J.R. Rice, *On the scalability of Asynchronous Parallel Computations*, J. Parallel and Distributed Computing **22** (1994).

[23] E. Mascarenhas and V. Rego, *Ariadne: Architecture of a Portable Threads System Supporting Mobile Processes*, Tech. Report CSD-TR-95-017, Dept. Comp. Sci., Purdue University, 1995.

[24] 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.

[25] S. Newton and S. Mitra, *Self organizing leader clustering in a neural network using a fuzzy learning rule*, SPIE Proc. 1565: adaptive signal processing, SPIE, 1991.

[26] J. Neyman, *Basic Ideas and Some Recent Results of the Thoery of Testing Statistical Hypotheses*, J. Royal Stat. Soc. **105** (1942), 292–327.

[27] J. Neyman and E.S. Pearson, *The Testing of Statistical Hypotheses in Relations to Probabilities a priori*, Proc. Cambridge Phil. Soc. **29** (1932), 492–510.

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

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

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

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

[32] John R. Rice, Elias N. Houstis, and Wayne R. Dyksen, *A population of linear, second order, elliptic partial differential equations on rectangular domains, part I*, Mathematics of Computation **36** (1981), 475–484.

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

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

[35] P.K. Simpson, *Fuzzy min-max neural networks-part I: Classification*, IEEE Trans. Neural Networks **3** (1992), 776–786.

[36] ———, *Fuzzy min-max neural networks-part II: Clustering*, IEEE Trans. Fuzzy Systems **1** (1993), no. 1, 32–45.

[37] 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.

[38] W.C. Stirling, *Coordinated Intelligent Control via Epistemic Utiliyu Theory*, IEEE Control Systems (1993), 21–29.

[39] W.C. Stirling and D.R. Morrell, *Convex Bayes Decision Theory*, IEEE Trans. System, Man and Cybernetics **21** (1991), 173–183.

[40] 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.

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

[42] 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.

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