1996

A MultiAgent Environment for MPSEs

Tzvetan T. Drashansky

Anupam Joshi

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

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

Sanjiva Weerawarana

Report Number:
96-013

A MULTIAGENT ENVIRONMENT FOR MPSEs

Tzvetan T. Drashansky
Anupam Joshi
John R. Rice
Elias N. Houstis
Sanjiva Weerawana

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR-96-013
February 1996

A MultiAgent Environment for MPSEs*

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

Abstract

In this paper we present a paradigm for simulating complex phenomena which may involve multiple physical phenomena and complicated geometry. The computational structure is of cooperating agents and much of the proposed methodology is widely applicable, but the focus in this paper is on phenomena modeled by partial differential equations (PDE). The computational process is to subdivide the physical object into components of simple geometric shapes modeled by a single problem solving environment (PSE). PSEs are viewed as agents which solve the PDE on each component independently. The interfaces between the components must have physical interface conditions satisfied; mediator agents use relaxation techniques for this. An agent-based architecture of an environment for building systems to implement this paradigm is described, using PSEs which are encapsulated into solver agents. This approach is naturally parallel and highly scalable; it is suitable for a wide variety of parallel and distributed high performance computing (HPC) architectures; it allows for the reuse and evolution of existing HPC software, and for a convenient abstraction of the solution process for non-expert users. An implementation of the architecture, named *SciAgents*, is presented and used to solve an example problem which illustrates this multidisciplinary problem solving environments (MPSE) framework.

1 Introduction

The predicted growth of computational power and network bandwidth suggests that computational modeling and experimentation will continue to grow in importance as a tool for big and small science. In this scenario, computational modeling will shift from the current single physical component design to the design of a whole physical system with a large number of components that have different shapes, obey different physical laws and manufacturing constraints, and interact with each other through geometric and physical interfaces. For example, the analysis of an engine involves the domains of thermodynamics (gives the behavior of the gases in the piston-cylinder assemblies), mechanics (gives the kinematic and dynamic behaviors of pistons, links, cranks, etc.), structures (gives the stresses and strains on the parts) and geometry (gives the shape of the components and the structural constraints). The design of the engine requires that these different domain-specific analyses interact in order to find the final solution. The different domains share common parameters and interfaces but each has its own parameters and constraints. We refer to these multi-component based physical systems as multidisciplinary applications (MAs).

This scenario of modeling will have a significant impact in industry, education, and training. Realizing it will require the development of new algorithmic strategies and software for managing the complexity and

*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

harvesting the power of the expected high performance computing and comminication (HPCC) resources; it requires technology to support programming-in-the-large and to reduce the overhead of HPCC computing. Our research aims to identify the framework for the numerical simulation of multidisciplinary applications and to develop the enabling theories and technologies needed to support and realize this framework in specific applications. The Multidisciplinary Problem Solving Environment (MPSE) is the software implementation of this framework. It is assumed that its elements are discipline-specific problem solving environments (PSEs)[9]. The MPSE design objective is to allow the "natural" specification of multidisciplinary applications and their simulation with interacting PSEs through mathematical and software interfaces across networks of heterogeneous computational resources.

In simple terms, an MPSE is a framework and software kernel for combining PSEs for tailored, flexible multidisciplinary applications. A physical system in the real world normally consists of a large number of components which have different shapes, obey different physical laws and manufacturing/design constraints, and interact through geometric and physical interfaces. Mathematically, the physical behavior of each component is modeled by a system with various formulations for the geometry, interface/boundary/linkage and constraint conditions in many different geometric regions. In the case of complicated artifacts such as the automobile engine, which has literally hundreds of odd shaped parts and a dozen physical phenomena, it is difficult to imagine creating a monolithic software system to model accurately such a complicated real problem. Therefore, one needs an MPSE mathematical/software framework which, first, is applicable to a wide variety of practical problems, second, allows for software reuse in order to achieve lower costs and high quality, and, finally, is suitable for some reasonably fast numerical methods. Most physical systems and manufactured artifacts can be modeled as a mathematical network whose nodes represent the physical components in a system or artifact. Each node has a mathematical model of the physics of the component it represents and a solver agent[3] for its analysis. Individual components are chosen so that each node corresponds to a simple mathematical problem defined on a regular geometry. There exist many standard, reliable solver systems that can be applied to these local node problems. In addition there are nodes that correspond to interfaces (e.g. objective functions, relations, common parameters and their constraints) that model the collaborating parts in the global model. The analysis of an artifact changes through time, thus some of the interfaces appear and disappear during the analysis session. To solve the global problem, we let these local solvers collaborate with each other to relax (i.e., resolve) the interface conditions. An interface controller or mediator agent[3] collects boundary values, dynamic/shape coordinates, and parameters/constraints from neighboring subdomains and adjusts boundary values and dynamic/shape coordinates to better satisfy the interface conditions. Therefore, the network abstraction of a physical system or artifact allows us to build a software system which is a network of collaborating well defined numerical objects through a set of interfaces. Some of the theoretical issues of this methodology have been addressed in [21, 24, 18] for the case of collaborating PDE models. The results obtained so far verify the feasibility and potential of network-based prototyping.

We see MPSEs as delivering problem solving services over the Net. This viewpoint leads naturally to collaborating, agent based methodologies. This, in turn, leads to very substantial advantages in both software development and quality of service as follows. We envision that a user of MPSEs will receive at his location only the user interface. Thus the MPSE server will export to the user's machine an agent that provides an interactive user interface built on top of the standard services of the Net. The bulk of the software and computing is done at the server's site using software tailored to a known and controlled environment. The server site can, in turn, request services from specialized resources it knows, e.g., a commercial PDE solver, a proprietary optimization package, a 1000 node supercomputer, an ad hoc collection of 122 workstations, a database of physical properties of materials. Each of these resources is contacted by an agent from the MPSE with a specific request for problem solving or information service. Again, all this collaboration is built on standard Network services. All of this can be managed without involving the user, without moving software to arbitrary platforms, and without revealing source codes. This approach also allows software reuse for easy software update and evolution, things that are extremely important in practice. The real world is so complicated and diverse that we believe it is impractical to build monolithic, universal solvers for such problems. Without software reuse, it is impractical for anyone to create on his own a large software system for a reasonably complicated application. Each new automobile normally results in a new software system. Recreating such a system could easily take several months or years. In contrast, the execution time to perform the required computation might only be a few days. Notice that such a physical change usually corresponds to replacing, adding, or deleting a few nodes in the network with a corresponding change in interface conditions. These are simple manipulations on a network which do not affect the rest of the system and can thus be easily done. In this application each physical component can be viewed both as a physical object and as a software object. In addition, this mathematical network approach is naturally suitable for parallel and distributed computing as it exploits the parallelism in physical systems. One can handle issues like data partition, assignment, and load balancing on the physics level using the structure of a given physical system. Synchronization and communication are controlled by the mathematical network specifications and are restricted to interfaces of subdomains, which results in a coarse-grained computational problem. This is especially suitable for today's most advanced parallel and distributed supercomputer architectures. The network approach also allows high scalability.

2 Background and Related Work

The initial ideas regarding PSEs can be seen in the 1963 Culler and Fried paper "An On-Line Computing Center for Scientific Problems" – a time when Fortran and Algol were still novelties. The proceedings of the 1967 ACM conference "Interactive Systems for Experimental Applied Mathematics" provides an overview of early work on PSEs, which failed primarily because of the lack of computing power.

The PCs and workstations of the 1980s finally provided the computing power to realize the hopes of the early 1960s. The mass market of computing has moved from the research laboratory to the office (spread-

sheets, word processors), the home (tax preparation, education), and service industries (airline reservations, banking). PSEs naturally thrive in these markets since the solvers are usually simpler and less compute-intensive, and the users are less able to do traditional programming. Consider, for example:

- Printing: Desktop publishing systems have replaced manual page layout and typesetting.
- Accounting: Spreadsheets have replaced desk calculators and paper and pencil.
- Statistics: Systems such as SPSS and SAS have replaced Fortran programs.
- Architecture and civil engineering: Computer-aided design systems have replaced drafting tools, handbooks, and Fortran code.

These systems are not all fully developed in the way we envisage future PSEs, but they have many PSE characteristics and have had major impacts on their fields.

In science, successful systems have been built to solve partial differential equations. Such systems often have some of the functionality envisaged in PSEs. For example, the mathematical software package RPI adaptively solves parabolic PDEs in one and two space dimensions using finite-element procedures that can automatically select and vary both the mesh and the elements. Temporal integration, within a method-of-lines framework, automatically chooses between backward-difference and Runge-Kutta methods.

Ellpack was designed to solve second-order elliptic PDEs in 2D and 3D and to evaluate software for such computations. It is a modular system with a domain-specific PDE language and a variety of elliptic PDE solvers. XEllpack and Parallel Ellpack are recent extensions. XEllpack provides graphical input for constructing grids, pop-up menus for selecting solution techniques, and color graphics output for analyzing solutions. A user can interface with XEllpack from an X Windows workstation while an XEllpack client solves an elliptic problem on any machine(s) on the network. Parallel Ellpack is an interface to various libraries of parallel elliptic-PDE solvers. It allows the user to specify nonlinear and time-dependent as well as elliptic PDE problems interactively, and it maps the underlying computation onto parallel machines automatically. This mapping can be displayed and modified interactively. All three systems can collect, visualize, and analyze performance data. Other systems that can be included in this category are Vecfem, PDE2D, Deqsol, Alpal etc. For a detailed overview of PSEs and their history we refer the readers to [9].

Many agent-based systems have been developed[36, 28, 31, 27], which demonstrate the capabilities 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. We believe that using such systems to handle complex mathematical models is natural and direct. The agent paradigm allows *distributed problem solving* [22] which is distinct from merely using distributed computing.

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.

Any multi agent system also needs to have mechanisms for coordinating the activities of various agents. Several researchers have addressed the issue of coordinating multiagent systems. For instance Smith and Davis [29] 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[35] how many intelligent sensor devices could pool their knowledge to obtain an accurate overall assessment of the situation. The specific task presented in their work involved detecting moving entities, even though each "sensor agent" saw only a part of the environment. They reported results using both an hierarchical organization, as well as an "anarchic committee" organization, and found that the latter was as good as, and sometimes better than the former. Cammarata 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.* [16] describes the FA/C (functionally accurate, cooperative) architecture in which agents exchange partial and tentative results in order to converge to a solution. Joshi [13] proposes an epistemic utility based approach which allows each agent to dynamically learn about the capabilities of other agents, and respond to the changes in these capabilities. This scheme uses a combination of learning as well as a priori rules relating to scalability of parallel computations[17].

Some attempts to combine several scientific computing applications in a (more or less) single environment have recently been reported in the literature. The project EDiCA [12] developed in INRIA, France, incorporates tools like Maple, Macaulay, Alpi, and other by using an integration component called Central Control, equipped with a GUI. The user interactions with the Central Control produce a script in its command language which may invoke the tools and other available programs to convert the output of a tool to the input to another tool. Another similar system has been developed by Girard [10] in which the user selects a set of applications (from a list of available ones), the input and the output of the computations (usually a file), and the system provides a program (based on the type of the I/O each application expects, with the user resolving the ambiguities) to invoke the applications in the right order, with some automatic conversions of the outputs to the input formats of the next application to call. While these environments allow for easy addition of new applications, their design is more suitable for solving problems where the invoked tools work sequentially, as different stages of the solution process rather than working on different subproblems and interacting with each other in the way MPSEs solving composite scientific models need to. There exist component-integration systems like the POLYLITH software bus [23] which are oriented toward

distributed systems developers and provide decoupling facilities to the programmers in order to successfully interface relatively independent software entities. Such systems may help in resolving some of the integration problems that are faced in developing an MPSE, but many of the problems we have mentioned remain and need to be addressed separately. The development of standards in communicating application-level data in scientific computing, such as OpenMath [30], can be of significant help to developers of MPSEs. OpenMath is an attempt to define a standard for communicating mathematical objects, processable by computers between applications.

3 MPSE as a User Abstraction

In this section we propose an approach for building MPSEs. The abstraction of the resulting MPSEs, presented to the user is also discussed.

3.1 Target Problems for Our MPSEs

We begin by specifying the kinds of problems our MPSEs are designed to solve. In general, these are 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, and possibly heterogeneous parts.
- Each part obeys a single physical law that can be modeled by a simple mathematical model in a simple subdomain.
- The interactions between the different parts are mediated by adjusting interface conditions along the subdomain boundaries with neighbors.

These properties are common in models of physical phenomena. An example is given in Figure 1. It models the temperature distribution in a small system of 6 different substances (with different laws for temperature distribution), a heater, and a sink.

3.2 MPSEs as Networks of Computing Agents

MPSEs contain two major types of computing agents – *solvers* and *mediators*. Each solver agent computes the local solution of a subproblem of the global problem. The "core" of the solver is actually a PSE designed for solving "simple" problems like the one in the individual subdomain (for more details on solvers' architecture see the next section). The solver is considered a "black box" by the other agents and it interacts with them only using a defined interagent language for the specific problem. This feature allows all computational decisions for solving the individual subproblem to be taken independently from the decisions in any other subproblem – a major difference from the traditional approaches to multidisciplinary simulations. Each mediator agent is responsible for adjusting an interface between two neighboring subdomains. Since the interface between any two subdomains may be complex itself, there may be more than one mediator assigned

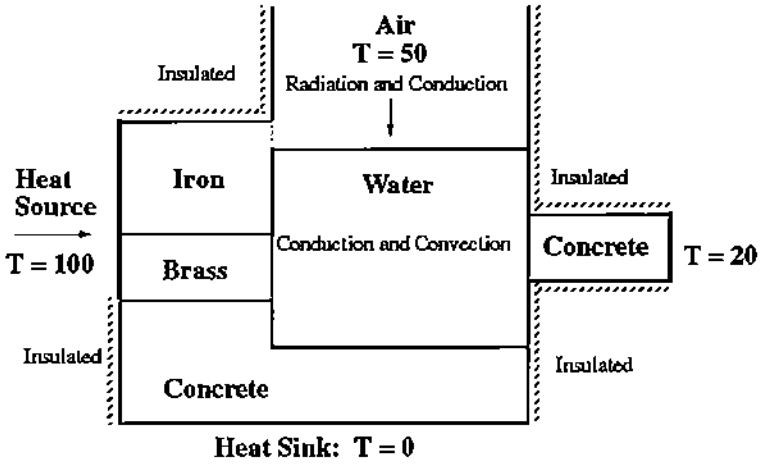

Figure 1: A simple heat flow problem with six components and three distinct physical processes of heat flow.

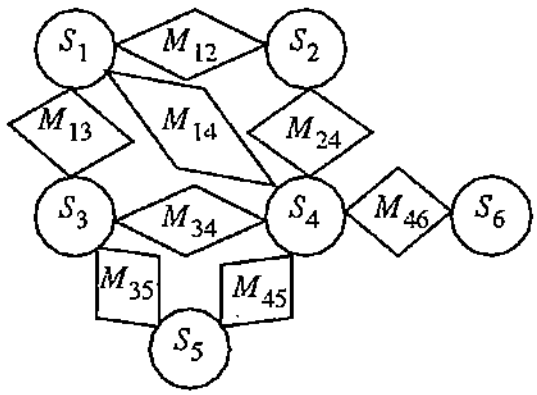

Figure 2: The user constructs a network of cooperating computing agents: solvers and mediators. The network for the problem of Figure 1 is shown with six solvers, S_i, and eight mediators, M_{ij}.

to adjust it, each of them operating on separate piece of the whole interface. Thus the mediators control the data exchange between the solvers working on neighboring subproblems by applying mediating formulas and algorithms to the data coming from and going to the solvers. Different mediators may apply different mediating formulas and algorithms depending on the physical nature of their interfaces. The mediators are also responsible for enforcing global solution strategies and for recognizing (locally) that some goal (like "end of computations") has been achieved.

The solvers and mediators form a network of agents that solves the given global problem. A network that solves the problem on Figure 1 is shown on Figure 2.

3.3 Building the Problem Solving Network — User's Abstraction

We now describe how the user builds ("programs") this network. The agent framework provides a natural abstraction to the user in the problem domain and hides the details of the actual algorithms and software involved in the problem solving.

Consider a problem like the one in Figure 1. The user needs first to break down the geometry of the composite domain into simple subdomains with simple models to define the subproblems for each subdomain. Then the physical conditions along the interfaces between the subdomains have to be identified. All that can be done in the terms of the user's problem domain. Then the user constructs the proper network of computing agents by simply *instantiating* various agents. The user is provided with an *MPSE constructor* (*agent instantiator*) — a process which displays information about the templates and creates active agents of both kinds, capable of computing. Initially, only *templates of agents* — structures that contain information about solver and mediator agents and how to *instantiate* them, are available. The user selects solvers that are capable of solving the corresponding subproblems and mediators that are capable of mediating the physical conditions along the specific interfaces, and assigns subproblems and interfaces, respectively, to each of them. The user interacts with the system using some visual programming like approach. Visual programming languages and systems have proved useful in allowing the non-experts to "program" by manipulating images and objects from their problem domain. In our case, a visual environment is useful for the MPSE constructor, or when the user wants to request some action or data.

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

The agents *actively* exchange partial solutions and data with other agents without outside control and management. In other words, each solver agent can request the necessary domain and problem related data from the user and decide what to do with it (should it, for instance, start the computations or should it wait for other agents to contact it?). After each mediator agent has been supplied with the connectivity and mediating data by the user, it contacts the corresponding solver agents and requests the information it needs. This information includes the geometry of the interface, the functional capabilities of the solvers with respect to providing the necessary data for adjusting the interface, visualization capabilities, etc. All this can be done without user involvement. By instantiating the individual agents (concentrating on the individual subdomains and interfaces) the user builds the highly interconnected and interoperable network that will solve the problem, by *cooperation* between individual agents.

The user's high-level view of the MPSE architecture is shown in Figure 3. The global communication medium used by all entities in the MPSE is called *software bus* [32]. The MPSE constructor 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 data it needs to compute a local

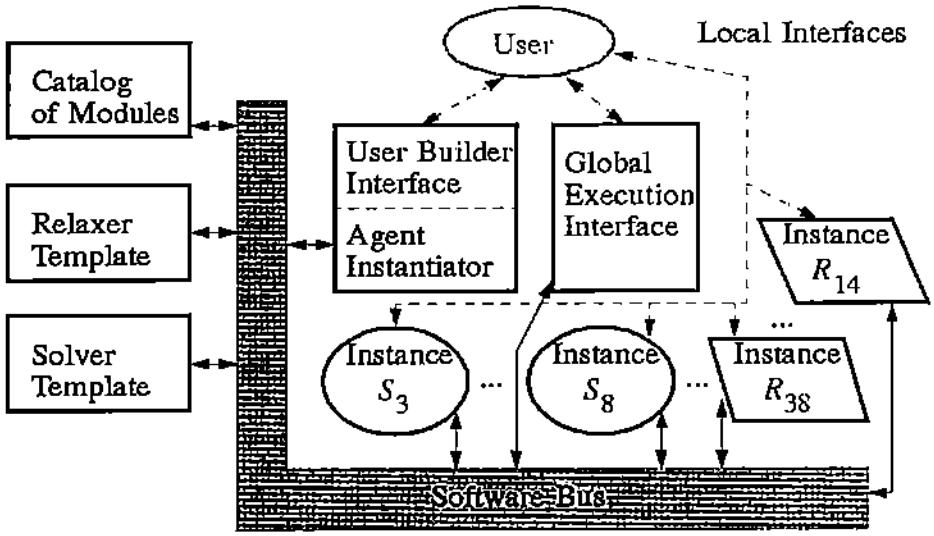

Figure 3: Software architecture of an MPSE: the user's abstraction. The user initially interacts with the User Interface Builder to define the global composite problem. Later the interaction is with the Global Execution Interface to monitor and control the solution of the problem. Direct interaction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

solution (i.e., a mediator agent is missing), then it suspends the computations and waits for some relaxer agent to contact it and to provide the missing values (this is also a way to "naturally" control the solution process). If a mediator 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 our architecture. It results from each agent adapting to its environment. We go into more detail of the inter agent communication during the solution process later.

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

The collection of agent interfaces that a user interacts with is the only software the user actually needs to run locally in order to solve her/his problem. Therefore, this architecture abstracts successfully from the user the location of the main computations (the location of the solvers and the mediators) and allows for great flexibility in this direction, including running the MPSE over the Internet and distributing the agents over a WAN.

3.4 Intelligent Completion of the Problem Specification

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

The complete functional (mathematical) description of the problem includes:

- definition of the subdomains
- definition of the models in each subdomain (from the user's point of view)
- definition of the interfaces between subdomains and physical conditions along them (the latter should be part of the mathematical model)

In addition, the user selects a visualization method (an aesthetic or pragmatic issue, not requiring any special "computing" expertise) and global solution criteria – such as solution accuracy and solution time. Note that all of the above items are entirely defined in the terms of the user's problem domain and require very little scientific computing expertise.

The agents, however, need lots of additional data, parameters, and configuration values in order to proceed with the solution process. We provide three representative examples next. First, the local solver agents need a set of computational parameters for the single-domain problem they have to solve. These may include (for instance, in the case of partial differential equations models) the discretization method for the domain and the equation, grid/mesh sizes and configurations, linear solvers, etc. One of the good features of our approach is that the solvers do not need to coordinate the values of the parameters among themselves – each solver has complete independence in its decision about the values of these parameters.

Second, the relaxer agents have to select a set of parameters related to the mediation formulas and algorithms the user has chosen and to inform the solvers what data they need to provide during the solution process.

Third, the agents have to make use of the available hardware. It is possible that multiple computing units will be available for this particular problem. The MPSE constructor will try initially to distribute the agents evenly among the computing units but it has very little information in order to make an intelligent decision – it knows only the pairs of agents that communicate with each other. The mediator agent computations usually 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 [25] 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 further 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.

In order to relieve the user from the necessity of making decisions about all these parameters, the MPSE must have some way to intelligently select them without the user's participation. There exist software systems which are capable of selecting some of the above parameters in specific problem areas. They use computational intelligence techniques and knowledge bases to deduct (close to) optimal values of various parameters. An example of such system is *PYTHIA* [15] whose objective is to advise the user of the "right", or at least "good", selections of various solvers, their parameters and the computational resources for solving a particular single-domain partial differential equations problem. Our multiagent approach to building MPSEs allows us to easily incorporate systems like that into the solution process by "converting" them into agents and using the agent communication language the solvers and the mediators communicate through. Then the computing agents send requests to the *consulting* agents for the necessary parameters and the consulting agents send back their estimate for the parameter values.

4 Software Architecture of MPSE

In this section we introduce and discuss in detail the software architecture of a programming environment for building MPSEs.

4.1 Software Components of an MPSE

It should be clear from our previous discussions that we are not trying to build a system that is capable of solving every composite heterogeneous problem. Instead we attempt to design a software system that allows users to reuse and combine legacy software to solve the specific problem at hand. In other words, the users build a different MPSE for each composite problem they solve. Figure 4 outlines the software and the services that can be incorporated into an MPSE. There is a large number of libraries designed to solve various parts and stages of mathematical models of physical systems and their use would add to the power of an MPSE-building system. Many of these libraries also contain parts of PSEs available for solving single-domain models. The PSE servers (as solver agents) and the knowledge base servers (KBSs) (as consulting agents) have to be an integral part of the MPSEs, as well as the database servers (DBSs) which manage the underlying data sets. A significant part of this software already exists. An MPSE needs a communication medium which is accessible to all agents and components. Various existing network and transport level protocols and software can be used but they need to be coupled with an agent messaging system (like a KQML [8] implementation) and an *intelligent controller* in order to successfully instantiate and run the agents, and permit inter-agent cooperation. In order to help users build the MPSE from their problem specifications various toolkits are needed such as a database of mediator and PSE (solver) templates, and builders to construct them. Finally, the functionality of the user interface requires a number of tools and editors — a "notebook"-like interface[33], tools for monitoring and browsing the computations, and tools for

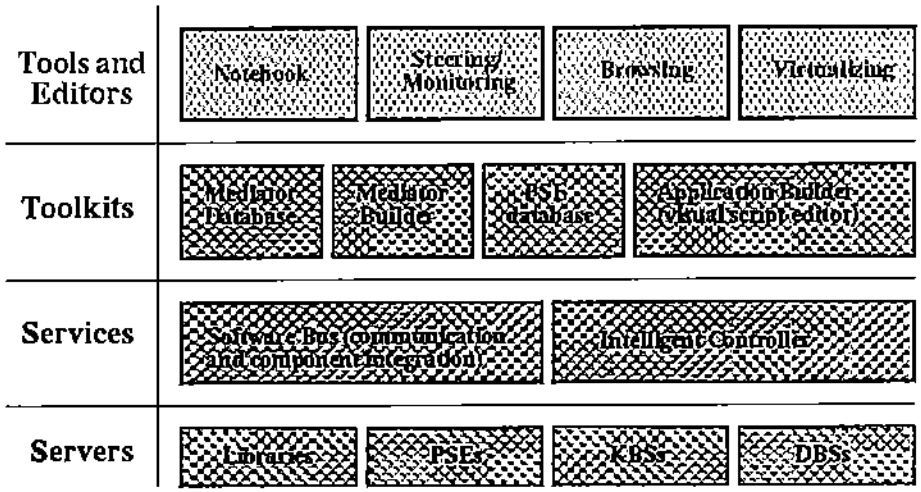

Figure 4: Software and services that can be part of an MPSE

visualizing the different parts or the global solution.

Consider now an MPSE built to solve a specific problem. We can group the software components integrated into it into five layers (see Figure 5). At the bottom (closest layer to the hardware) is the communication and the OS layer software which we call *software bus*, it provides component integration services. It has to provide a common communication medium for agents running on heterogeneous architectures and different locations. It also is responsible for facilitating agent migration and distribution (e.g., when the agent interface and its computational routines have to be run on different computers), as well as for parallel and asynchronous run of PSEs, should the appropriate hardware be available.

The integration environment layer of MPSE software consists of the *intelligent agent controller and coordinator*. It is based on the software bus, and offers higher-level communication services which can be used directly by the agents. These include capabilities for delivering KQML messages, and white page/yellow page (locator) facilities for the participating agents. Locator services can provide information such as availability, cost, etc. of given service, agent, or hardware resources. For that reason, it has to understand the ontology and some of the semantics of the language the agents use to interact. Note that the language content of the interagent communications may change from MPSE to MPSE, especially if the areas of the models they are solving are different. Identifying the common part of the language content which should be understood by the intelligent controller and coordinator is a difficult problem, which we do not address in the present research. Its solution is likely to include the use of a "standard" agent meta-language such as KQML. Another task of the intelligent controller and coordinator is to provide operational and service transparency during the migration of agents. It also initiates such migration of the computing agents when necessary. It must evaluate and enforce the global parameters, goals, and constraints given by the users. Locating or maintaining the data for composing a global view of the solution when requested by the users is also its responsibility.

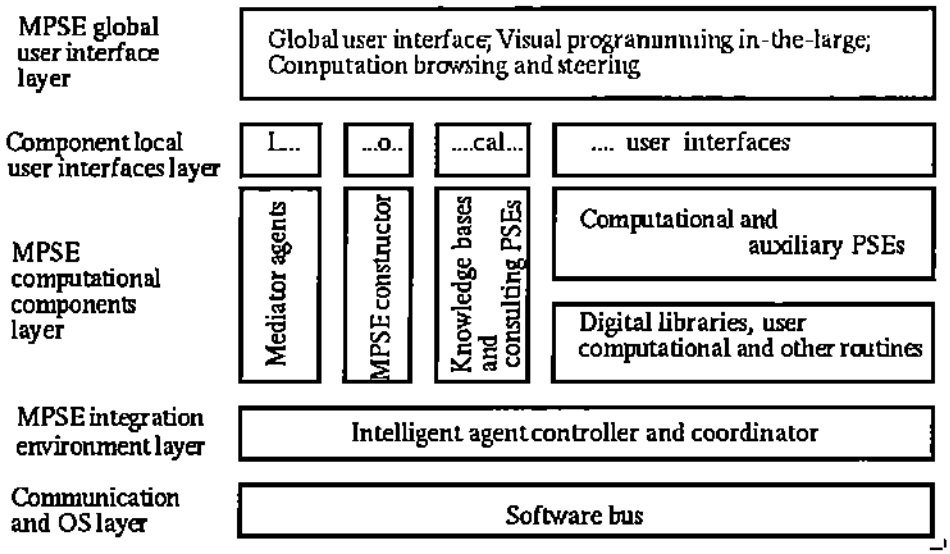

Figure 5: Software components in an MPSE system. Five layers of components are shown.

The next layer of components includes all computing and consulting agents and the MPSE constructor. We have already enumerated in previous sections the tasks of the mediator and solver agents. To recapitulate, a solver agent usually consists of a PSE that includes a number of libraries and other computational routines. The consulting agents supply the computing agents with values of various parameters of the solution process in order to automate the solution process. The *MPSE constructor* instantiates the agents when the users build the MPSE to solve a given model. It maintains databases of available or "known" solver (PSE) and mediator templates and presents them to the users. It also interacts with the instantiated agents in order to determine their capabilities and to provide them with the data about their environment and connectivity (e.g., which subproblem a solver is working on and which interface a mediator is adjusting), as well as identifies possible inconsistencies and contradictions in the global problem definition. These may arise as a result of several users defining a complex composite problem.

Each computing and consulting agent, as well as the MPSE constructor, has its own user interface. All such interfaces are grouped in the local user interfaces layer. It is the first layer to which the users have direct access. To facilitate extensive use of the *virtual computing* [34] model, and to allow collaboration between more than one users, the user interface of an agent must be separate from its functional core[2].

The top layer in our MPSE architecture consists of the *global user interface*. It has several tasks:

- Presenting a global view of the MPSE in order to allow easy building and manipulation of the agents.
- Facilitating user collaboration — the user must be able to see only part of the MPSE, to talk to another user of the same MPSE, to define only part of the MPSE and then to "connect" it to the remaining parts (which may be defined by other users).
- Obtaining and changing the global parameters for control over the global solution.

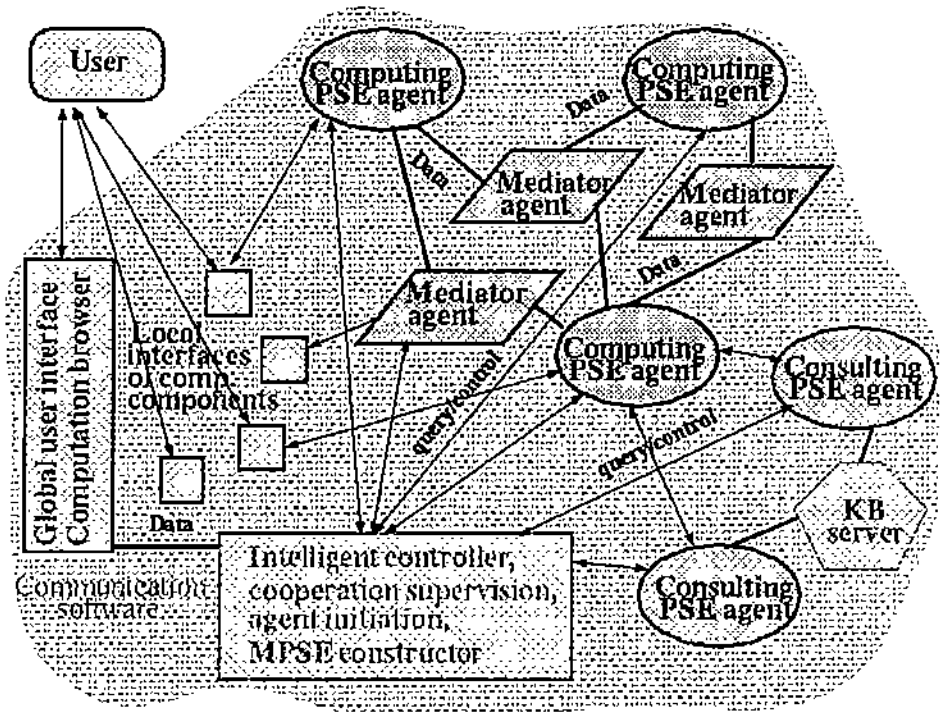

Figure 6: Functional architecture of an MPSE. The computations (and the major data exchange) are concentrated in the network of solver (PSE) and mediator agents. The computing agents communicate with the consulting ones through queries to obtain "advice" on computation parameters. The user interacts with the system through the global and local user interfaces which send queries and receive replies from the various agents. The intelligent controller and the MPSE constructor can be integrated into a single "agent" which controls the global state of the computations and instantiates, queries, and manages (if necessary) the other agents.

- Presenting independent user interfaces to different users, using the same (consistent) global data.
- Displaying the current global and partial solutions to the users.

The global user interface has access to the intelligent controller and coordinator and to the local agent interfaces (in order to activate them if requested).

4.2 Functional Architecture of an MPSE

We are ready now to present schematically the functional architecture of an MPSE (see Figure 6).

The computations (and the major data exchange) are concentrated in the network of solver (PSE) and mediator agents. This network is built by the users (by defining the composite model they wish to solve with this MPSE). All other agents are "hidden" from the users and are instantiated only if it is required by the computing agents. Only the global user interface and the intelligent controller/MPSE constructor are present before the users begin to specify the model to solve. The computing agents communicate with the consulting ones through queries to obtain "advice" on computation parameters. The user interacts with the system through the global and local user interfaces which send queries and receive replies from the various

agents. The intelligent controller and the MPSE constructor can be integrated into a single "agent" which controls the global state of the computations and instantiates, queries, and manages the other agents.

4.3 Software Reuse and Evolution

One of the major goals of the MPSE concept is to design a system that allows for low-cost and less time-consuming methods of building the software to simulate a complex mathematical model of physical processes. This goal cannot be accomplished if the existing rich variety of problem solving software for scientific computing is not used. More precisely, there are a number of well-tested, powerful, and popular PSEs for solving problems very similar or identical to the subproblems that appear when breaking the global model into "simple" subproblems defined on a single subdomain. These PSEs could easily and accurately solve such a "simple" submodel. It is, therefore, natural that we wish to use such PSEs as solver agents. However, our architecture requires the solvers to behave like agents (e.g., understand agent languages, use them to communicate data to other agents), something none of the PSEs in scientific computing are able to do to the best of our knowledge.

Our solution to this problem is to provide an *agent wrapper* for PSEs and other software modules, which takes care of the interaction with the other agents and with the other aspects of emulating agent behavior. The wrapper encapsulates the original PSE and is responsible for running it and for the necessary interpretation of parameters and results. This is not simply a "preprocessor" that prepares the PSE's input and a "postprocessor" that interprets the results, since the type of mediation between the subproblems may require communicating to the mediators some intermediate results and/or accepting some additional data from them. Designing the wrapper is sometimes complicated by the "closed" nature of extant PSEs — their original design is not flexible or "open" enough to allow access to various parts of the code and the processed data. However, it is our opinion that the PSE developers can design and build such a wrapper for a very small fraction of the time and the cost of designing and building the entire PSE. The wrapper, once written, will enable the reuse of this PSE as a solver agent in different MPSEs, thus amortizing the cost further. As part of the specifications of the wrapper the developers have to consider the mediation schemes involving submodels within the power of the PSE. An additional task is to evaluate the PSE's user interface — since the user defines the local submodel through it, it is important that the interface facilitates the problem definition in user's terms well enough.

4.4 MPSE Kernel

If different MPSEs built according to the architecture presented above are compared, one would notice that many components are very different (solvers, mediators, consulting agents, interagent language content, etc.). However, the "glue" that holds together the MPSE components, the way the user builds it (if we abstract the problem-specific terms and features), and certain parts of the global user interface have to be very similar. It is possible to design an *MPSE kernel* on the basis of the proposed MPSE architecture that contains all common software parts. These include:

- software bus
- intelligent controller and coordinator
- MPSE constructor
- global user interface and the MPSE constructor's interface

It is also possible to include in the kernel mediator agent "skeletons". As a part of our ongoing work, we are trying to abstract common features that all mediator agents will have.

5 Implementation

In this section we present *SciAgents* — an environment for building MPSEs for solving composite heterogeneous partial differential equations (PDE) models. *SciAgents* is an implementation of the MPSE approach and architecture discussed in the previous sections. The PDE models have been selected as the target class of problems of our implementation in part because they are among the more complex mathematical models that scientists and engineers use.

We first discuss briefly how solving composite PDE models fits into our MPSE building approach and the appropriate solvers and mediation schemes. Then we take a close look into the agent architecture and the interagent cooperation in *SciAgents* . Finally, we describe the coordination of the solution process.

5.1 Solving Composite PDE Models

Many physical phenomena are modeled mathematically by partial differential equations (PDEs). When the model of the phenomenon is simple enough, then the resulting PDE problem consists of a single domain with a single PDE defined on it (together with appropriate boundary conditions and initial conditions). Solving such a PDE problem numerically involves specifying the geometry of the domain, the PDE, and the boundary conditions in proper data structures, discretizing the domain according to a selected numerical method, forming a (non)linear system of equations, and solving it. The user of the solution might also like to visualize it. The rest of this section assumes some familiarity of the reader with this process. There exist general solvers (PSEs) for this class of problems like //ELLPACK [11, 26] which has tools (graphical and symbolic user interfaces) for defining the problem, a set of discretization methods for various problems, a set of linear equation solvers, and a set of routines for visualization of the solution. It also makes use of HPC hardware. There also exist consulting systems like PYTHIA [15] that are designed to relieve the user from having to make too many computational choices.

The single-domain PDE problems, however, can not adequately model many of the physical world phenomena. With the increasing use of the computers for real-world scientific simulations there is a growing need to solve multiple-domain PDE models. Most such models fall into the description of suitable problems we gave in Section 3 for MPSEs based on our approach; in fact, the example in Figure 1 is a PDE model. In addition, multiple-domain PDEs often have complicated geometry and are highly non-homogeneous. Such

problems require variable grid density and different discretization methods in different subdomains due to the different nature of the PDEs involved. The traditional domain decomposition methods consider and discretize an entire problem as a whole, before decomposing the resulting (huge) linear system for processing. This is necessary since these methods need to synchronize the grid points along the subdomain interfaces. The size of some important problems, however, is so big that considering the entire problem domain is itself an almost impossible task. For example, an engine simulation is estimated to require 100 million variables and the answer (the data set allowing the display of the accurate solution at any point) is 20 gigabytes in size. The problem contains about 10,000 subdomains with 35,000 interfaces [20].

Clearly, custom software is required for solving each multiple-domain PDE problem and it is not feasible to build it with the traditional software development technologies. On the other hand, it is easy to observe that if the composite model can be broken down into a collection of single-domain problems, we can apply the MPSE approach for which we already have efficient, existing software for the solver agents like //ELLPACK.

Then the main issue is what mediation schemes can be applied in this case — in other words, how to obtain a global solution out of the local solutions produced by the single-domain solvers. To do this, we use the interface relaxation technique [5, 4, 20, 19]. Important mathematical questions of the convergence of the method, the behavior of the solution in special cases, etc., are addressed in [20]. This technique uses physical relations among the parts of the model 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 [5, 20]. 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 mediation scheme is not hard to implement in our MPSE framework, although it requires not single but repetitive solving of the submodels by the solver agents. Its implementation does raise some technical problems which are covered in detail in [6, 4, 5].

5.2 Agent Architecture of *SciAgents*

Since *SciAgents* comply with the general MPSE architecture we have described, we concentrate here on the implementations details and on the decisions we have made in the agent architecture and the specifics of the interagent communication.

In *SciAgents* at the highest level communication is done using the Knowledge Query and Manipulation Language (KQML [8, 7]) from ARPA's knowledge sharing initiative, and particularly using the public domain implementation KAPI by EIT Corp. and Lockheed, Inc. 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 [32]. 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 and demonstrates the application of the idea of *agent wrappers*. Each solver consists of a *core* implementing the functionality of the PDE solving process and the local user interface plus a *wrapper* which gives the solver the behavior and the appearance of an agent. In one agent network the user may include solver agents obtained from different simple-problem solvers. *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. For example, the agent wrapper of //ELLPACK (currently the only available solver template, although //ELLPACK contains many actual PDE solvers when run in different modes) is less than a 1000 lines of code long, while //ELLPACK itself contains close to a million lines of code. No more than 300 lines of code have been changed in the original code of //ELLPACK in order to provide the wrapper with all necessary data. The wrapper uses the //ELLPACK user interface to obtain the subproblem definition and runs (when requested to by the mediators) the computational part. It also translates data to and from the S_KIF format and receives and sends the appropriate messages to other agents. The computational parameters not specified by the user are obtained through requests to consulting PYTHIA agents.

The architecture of the mediators 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 mediator agent is detailed in Figure 7. Each of two submediators controls and supplies data to and from one solver on one side of the interface. Each submediator uses its own relaxation and approximation algorithms and communicates relatively independently with the solver agent on its side of the interface. These submediators are the processes that do the actual computation and initiate the consecutive iterations during the problem solving process. The two submediators share the user interface and the configuration module. The user interface module presents the mediator agent as a single entity to supply and request user information. It also handles requests for dynamic changes of the parameters.

The configuration module is responsible for "orienting" the agent in its environment. After the mediator has been instantiated, the configuration module requests connectivity information (which interface am I

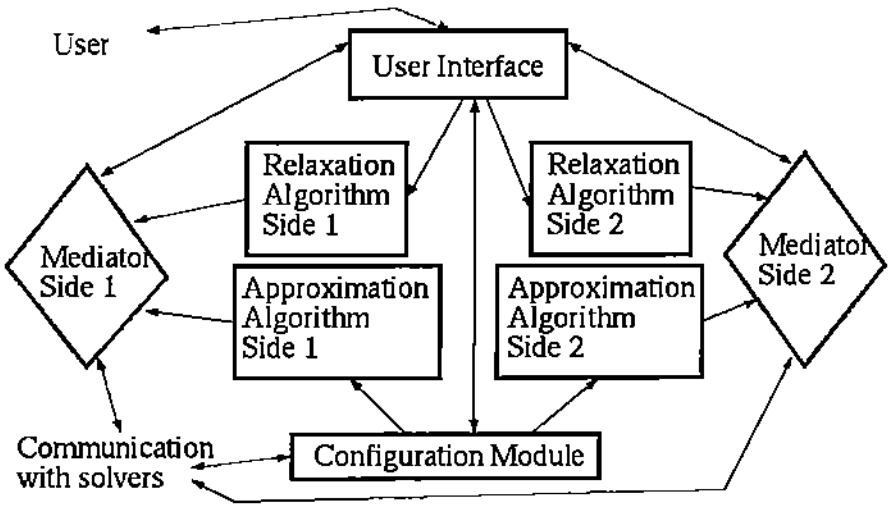

Figure 7: Software architecture of a mediator agent. The mediator agent is divided internally into two submediators — each submediator controls and supplies data to and from one solver on one side of the interface. Each submediator uses its own relaxation and approximation algorithms and it communicates 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).

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 submediators 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 submediators. Its interface with them follows the interagent communication protocol valid for the entire *SciAgents* . Effectively, the mediator 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 mediator 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 mediator to the solvers it can be entirely separated between the two submediators and their solvers. In the opposite direction the data has to be delivered to both submediators. It it important to note that the pattern of the communication between the agents is completely local — each mediator agent communicates with two solver agents and each solver agent communicates with the mediators for the interfaces of its subdomain. This locality is an advantage for *SciAgents* since it allows for good scalability.

The architecture of the mediator agents allows us to distribute N subdomain solvers and M interface mediators among N computational units (if available) in a natural and efficient way. When the mediators compute, the solvers are idle and vice versa due to the nature of the interface relaxation technique. We

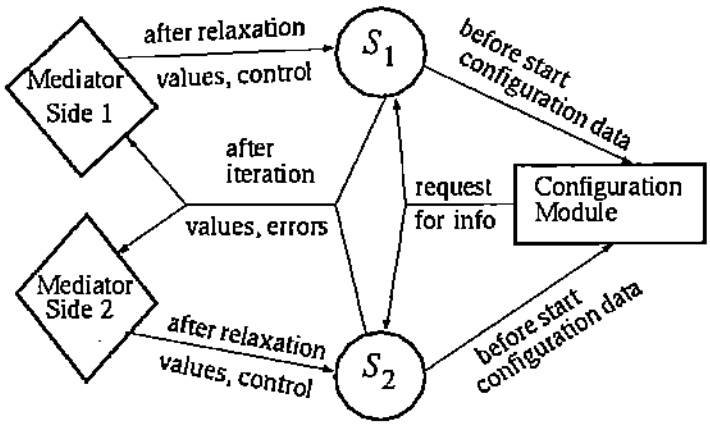

Figure 8: Mediator agent's communication with the solvers. After the initial exchange between the solver agents and the configuration module, the information flow is very simple. In the direction from the mediator to the solvers it can be entirely split between the two submediators and their solvers.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

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 (in our case the KQML message delivery system) 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 mediators for its interfaces "attached" to it.

Finally, the MPSE constructor (agent instantiator), the intelligent controller, 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.

5.3 Coordination of the Solution Process

We discuss now some important aspects of the cooperation between the agents during the solution process. There are well-defined global mathematical conditions for terminating the computations, for example, reaching a specified accuracy, or impossibility to achieve convergence. In most cases, these global conditions can be "localized" either explicitly or implicitly. For instance, the user may require different accuracy for different subdomains and the computations may be suspended locally if local convergence is achieved.

The local computations are governed by the mediators (the solvers simply solve the mathematical models). The mediator agents collect the errors after each iteration and, when the desired accuracy is obtained, *locally* suspend the computations and report the fact to the intelligent controller. 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 submediators and the submediators continue to check whether the local interface conditions are satisfied

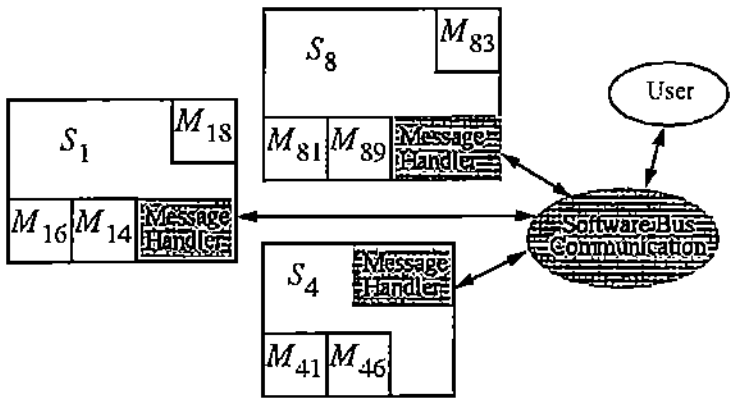

Figure 9: Software architecture of *SciAgents*: designer's view. Each rectangle represents a computing unit. There is a single subdomain solver running on one computing unit and it has all relevant parts of the mediators for its interfaces "attached" to it. The two submediators can be split between the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

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 mediator agent are no longer satisfied (even though they once were). In this case, the mediator 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 mediator reports failure to the intelligent controller and suspends the computations. The only global control exercised by the intelligent controller is to terminate all agents in case all mediators report local convergence or one of them reports a failure. The messages used in the interagent communication are given in full detail in [14], we provide a small example in the next section.

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.

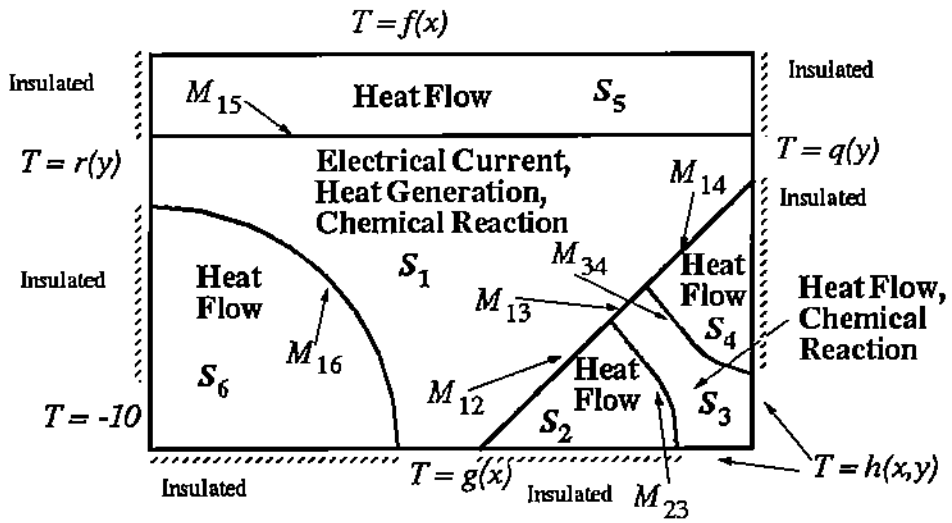

Figure 10: A multiple domain PDE model solved using *SciAgents* with 6 subdomains and 7 interfaces. The subdomains are shown, with the physical processes modeled in each of them and the solver agents S_i simulating the processes in each subdomain. The mediator agents point to the interfaces they adjust. The proper boundary conditions are also shown around the global domain.

6 Example

In this section we describe a multiple domain PDE example with 6 domains and 7 interfaces solved using *SciAgents* . It models the heat distribution caused by electrical current and chemical reactions in two substances and the surrounding subdomains. Figure 10 illustrates the domain and the model. The subdomains are shown, with the physical processes modeled in each of them and the solver agents S_i simulating the processes in each subdomain. We use the notation Ω_i for the subdomain simulated by S_i. One subdomain (Ω_1) has both an electrical current and a chemical reaction generating heat, another (Ω_3) has heat flow and a heat-producing chemical reaction, and the rest of the subdomains experience heat flow. All these physical processes are modeled by elliptic PDEs, although the underlying physics come from different disciplines (i.e., the model is *multidisciplinary*). The model satisfies the requirements outlined in Section 3 and we can use *SciAgents* to solve it. Note that solving it in any other way would not be feasible or at least would require the development of a large program designed to simulate only this specific model. The mediator agents point to the interfaces they adjust. The interface conditions are derived from their physical nature and reflect the continuity of temperature and heat conduction across the interfaces between the subdomains. Note that even though the interface between Ω_1 and Ω_2, Ω_3, and Ω_4 looks like a single interface from Ω_1, it is divided into three parts so that the mediators M_{12}, M_{13}, and M_{14} can be assigned a single piece to adjust. The proper boundary conditions are also shown schematically around the global domain.

We skip the precise problem definition (the equations, and the boundary and interface conditions) for the sake of brevity. Instead, we concentrate on how the problem is solved clarifying some details of *SciAgents* design and implementation. This model requires a two-stage solution. During the first stage, the task is to

determine a function (denoted by $v(x,y)$) which gives the amount of heat generated by the electrical current at each point of the domain Ω_1. This function participates in the heat flow/chemical reaction PDE in Ω_1 which is part of calculating the heat distribution in the global domain. The function $v(x,y)$ is a solution of a single PDE defined on Ω_1. The resulting PDE problem can be considered separately from the rest of the global domain since its solution (the function $v(x,y)$) does not depend on any of the surrounding subdomains. Hence, we do not need *SciAgents* in order to compute it — we simply have a single domain PDE with fixed boundary conditions. The function $v(x,y)$ is thus computed by an //ELLPACK session. The second stage of solution process is to form and solve the actual multiple-domain PDE problem. For that, we use *SciAgents* to build a network of 6 solvers and 7 mediators (suggested in Figure 10). Each solver agent consists of //ELLPACK and the agent wrapper we designed for it but the meshes generated for each subdomain and the other computational parameters are completely independent of each other. The wrapper invokes the //ELLPACK user interface when the agent is instantiated in order to obtain the subproblem definition from the user. After the problem is defined, an //ELLPACK program [26] is generated which is then translated into a Fortran program containing the information where (on which hardware platform) the resulting executable will be run. Then the program is compiled and linked to produce an executable, after which the //ELLPACK user interface exits and the wrapper takes over. It communicates the necessary data (including the coordinates of the boundary mesh points, the geometry of the interfaces, and that it is ready for the first iteration) to the mediators (if they are already instantiated and if they have contacted the wrapper already). For example, S_1 will respond to a query by M_{12} about the coordinates of the mesh points on its side of the M_{12}'s interface with the following message:

```
(reply  :sender S_1
        :content(get_interface_points(<reply code>,<format>,<list of points>))
        :receiver M_12
        :in-reply-to interface-points
        :language S_KIF
        :ontology PDE-relaxation)
```

The message provides information about the ontology (the context) of the message, the language of the content, and the query message this one is a reply to. If the receiver (the mediator M_{12} in this case) does not understand the ontology `PDE-relaxation` or the language `S_KIF`, it still can provide a reasonable reply message with the possible goal to negotiate a new language/ontology.

When the mediators provide all necessary boundary conditions (they are all zeros at the first iteration), the wrapper runs the executable. While running, the executable accesses the data (e.g., the boundary conditions) in files prepared by the wrapper. After the executable exits (the iteration is completed) the wrapper takes over again and extracts all required data from the computed solution and sends it to the mediators, waiting for the new boundary conditions from them. Thus, at the next iteration, no new compilation and user actions are necessary, since the same executable is run by the wrapper. This approach leads to greater

efficiency of the global solution process but does not allow dynamic migration of solver agents across machines. Consequently, no such migration is implemented in the current version of *SciAgents*. Naturally, if the users wish to change something during the computations, they may do so by invoking the //ELLPACK interface again. Then, a new executable is compiled and an attempt is made to continue the computations. In the current version, however, a change to the discretization of the subdomain results in restarting the computations locally (using as boundary conditions for the interfaces the latest solutions from the surrounding subdomains) since the mediators cannot evaluate the interface conditions if they do not have the coordinates of the interface mesh points from both sides of their interface before each iteration starts. Other user interventions (short of changing the equation, the geometry, or the proper boundary conditions) have minimal effect on the computational process outside the solver agent. Such changes may include specifying different linear solver, different visualization routine, different hardware architecture, or different machine to run. Note that these choices, including the discretization choice, should be made only by expert users — otherwise the available consulting agent advises on most of their values.

The mediators use the same relaxation formulas which differ only by the different heat conduction coefficients for each subdomain. A *PYTHIA* agent is used to "advise" the solver agents on the appropriate computation parameters. Queries to it are send only at the computation setup time and the obtained parameter values are reused at each iteration (if the users do not change them explicitly).

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] P. Dewan, *Principles of Designing Multi-User User Interface Development Environments Languages*, Proc. of the IFIP TC2/WG 2.7 Working Conference on Engineering for Human-Computer Interaction (Ellivuori, Finland), IFIP, Aug 1992.

[3] T. Drashansky, A. Joshi, and J.R. Rice, *SciAgents – An Agent Based Environment for Distributed, Cooperative Scientific Computing*, Proc. IEEE Intl. Conf. Tools with AI ICTAI'95, 1995, pp. 452–459.

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

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

[6] T. T. Drashansky and J. R. Rice, *SciAgents – Solving Complex Heterogeneous PDE Models Using Networks of Interacting Problem Solvers*, Mathematical Modeling and Scientific Computing 6 (1996), (to appear).

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

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

[9] E. Gallopoulos, E. Houstis, and J.R. Rice, *Computer as Thinker/Doer: Problem-Solving Environments for Computational Science*, IEEE Computational Science and Engineering **1** (1994), no. 2, 11–23.

[10] M. Girard, *An Easy Way to Construct Distributed Software*, Proc. of KBUP'95 - First Intl. Workshop Knowledge-Based Systems for the (Re)Use of Program Libraries, INRIA, Sophia Antipolis, 1995, pp. 65–74.

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

[12] INRIA, *Description of Project EDiCA*, http://zenon.inria.fr/safir/SAM/Edica/edica.html#Description.

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

[14] A. Joshi et al., *On Learning and Adaptation in Multiaget Systems: A Scientific Computing Perspective*, Tech. Report TR-95-040, Dept. Comp. Sci., Purdue University, 1995.

[15] ———, *Neural and Nero-Fuzzy Approaches to Support Intelligent Scientific Problem Solving*, IEEE Compputational Science and Engineering (1996), (to appear).

[16] 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.

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

[18] S. McFaddin and J. R. Rice, *Collaborating PDE Solvers*, Appl. Num. Math **10** (1992), 279–295.

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

[20] 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.

[21] ———, *Modeling with Collaborating PDE Solvers — Theory and Practice*, Computing Systems in Engineering **6** (1995), 87–95.

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

[23] J. Purtilo, *The POLYLITH Software Bus*, ACM Trans. Prog. Lang. and Systems **16** (1994), no. 1, 151–174.

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

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

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

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

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

[29] 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.

[30] R. Sutor, *The OpenMath Consortium*, http://wizkids.matematik.su.se/users/leifj/wshop/openmath.html.

[31] 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.

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

[33] S. Weerawarana et al., *Using NCSA Mosaic to build notebook interfaces for CS&E applications*, Tech. Report CSD-TR-95-006, Department of Computer Sciences, Purdue University, 1995, (submitted to IFIP WG2.7 EHCI '95).

[34] ———, *Web//ELLPACK: A Networked Computing Service on the World Wide Web*, Tech. Report TR 96-011, Dept. Comp. Sci., Purdue University, 1996.

[35] 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.

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