

A Monitoring Facility for an Object-Oriented Distributed Problem Solving Kernel

Jaime Simão SICHMAN*

Escola Politécnica da Universidade de São Paulo (EPUSP)
Depto. de Engenharia de Computação e Sistemas Digitais (PCS)
Av. Prof. Luciano Gualberto, 158, trav. 3
05508 - São Paulo - SP - Brazil
jssichma@brusp.bitnet

Eleri CARDOZO

Instituto Tecnológico da Aeronáutica (CTA-ITA)
Depto. de Engenharia Eletrônica (IEE)
12225 - Sao José dos Campos - SP - Brazil

Abstract

This paper presents a monitoring facility for an object-oriented distributed problem solving kernel named DPSK+P [?] [?]. This kernel allows both data and method sharing between agents, which may have been developed in different object-oriented programming languages, so that they can cooperate in a problem solving activity. The monitoring facility is based on control and communication events, and its exhibition of results uses process animation techniques. It also presents an efficient framework for the exhibition of the communication flow in shared memory based systems, which benefits from the existing compactness in object-oriented class hierarchies.

Keywords: Distributed Problem Solving, Object-Oriented Development Environments, Debugging and Monitoring Distributed Systems.

* Jaime Simão Sichman is currently at LIFIA-Institut IMAG, 46, avenue Felix Viallet F-38031 Grenoble, France, jaime@lifia.imag.fr, phone: (33) 76574807, fax: (33) 76574602

1 Introduction

The applications being developed in the last years are typically large and complex. These applications have as design requisites the support for systems modification, extension and reconfiguration. The execution model is highly paralleled, both in respect to the computer environment and the nature of the application itself. On the other hand, systems built for engineering purposes tend to be heterogeneous in nature, meaning that the application is better designed using different programming languages and running in a distributed computational environment, which may consist of different hosts connected by a local-area network. Unfortunately, the integration of modules developed in different programming languages is very complicated and not efficient nowadays. Moreover, this kind of activity often requires a helpful environment to support the different phases of the software design process, especially debugging and monitoring facilities. In order to overcome these difficulties, one can use *distributed problem solving* techniques at the conceptual level and *object-oriented techniques* at the implementation level. Recently, some attempts are being made in order to design large-scale software organizations using these two approaches, that is, by incorporating object-oriented techniques in the development of distributed problem solving environments [?]. The DPSK+P system [?], which is described in this paper, is an example of such an approach.

1.1 Distributed Problem Solving

Distributed problem solving is a methodology arising from distributed artificial intelligence research and that is used in the design of large-scale software organizations. Given an original problem, usually very large and complex, it is successively decomposed in various subproblems, which are simpler and smaller, so that one can develop computational modules to solve them. The solution of the original problem is obtained by integrating the results of each of these subproblems. The computational modules are called agents and tend to have the following properties [?]:

- *high complexity*, meaning that the modules perform complex tasks;
- *high granularity*, meaning that the modules expend more time processing than communicating;
- *high heterogeneity*, meaning that the tasks performed by the modules are very diversified;
- *high hardware generality*, meaning that the modules execute on general purpose processors, without any customization regarding to the hardware system.

This methodology is also known as *cooperative processing*, since the agents cooperate in the original problem solving process. Cooperation is achieved by **control** and **communication actions**. Communication actions refer to information exchange between the agents, while control actions refer to the coordination activities, like activating, finishing, synchronizing and interrupting the agents' processing. When used in a proper way, these control actions can guarantee that a particular agent will start his activities at the right time and that the necessary resources will be allocated to it. The Blackboard model [?] is a classical example of cooperative processing. Other references to distributed artificial intelligence research can be found in [?] [?]. Particularly, an interesting result is shown in [?], where some techniques from organization theory are used in the design of large-scale software organizations. These attempts are based on the construction of computational models which simulate human organizations, like groups, hierarchies and markets .

A distributed problem solving kernel must therefore provide some mechanisms for the over-the-network communication and process control. Communication can be achieved, for instance,

through message passing, sharing of data, or remote procedure call. On the other hand, control can be achieved through semaphores, events, or barriers, added to the proper operating system directives for execution control.

1.2 Object-Oriented Programming

Object-oriented programming has arisen as an attempt to solve some problems detected by Software Engineering, like difficulties in extending and evolving systems [?] [?]. Therefore, an object-oriented programming style allows software extensibility in a very simple way. One can reuse a class hierarchy already developed for another application, and even specialize it, creating new classes or modifying some of its existing methods. This certainly improves the efficiency of the software design cycle.

On the other hand, a distributed problem solving kernel designed to integrate agents which are object-oriented programmed must provide control and communication primitives using the same constructions found in the object-oriented languages employed by the agents.

The DPSK+P system, an object-oriented distributed problem solving kernel is described in the next section. Section 3 presents the current main techniques used in monitoring of distributed systems. We have chosen some of these techniques in order to design a monitoring facility for the DPSK+P system [?], described in section 4. Section 5 presents the first author's current research, regarding the use of the DPSK+P system in the Distributed Artificial Intelligence domain. Finally, section 6 presents our conclusions about this work.

2 The DPSK+P System

Object-oriented programming presents inherently a distributed approach. Therefore, efforts to extend its facilities to a distributed environment are desirable and even natural. Using this kind of extension, agents developed in an object-oriented style could share objects, even if these were defined by other agents, running in the same or in another host.

The main goal of the DPSK+P system [?] is to incorporate some object-oriented practices in a distributed problem solving kernel. These practices refer to defining class hierarchies, methods and instantiation mechanisms to shared objects.

The basic idea is to allow an agent to make some of its methods and data sharables, whenever it desires that these methods and data could be accessed by other agents. In order to achieve this goal, object contents are classified in three different visibility levels: a **private level**, whose access is allowed only to the object itself, a **public level**, whose access is allowed to the other objects kept by the same agent and a **shared level**, which extends this right to all the objects in the system, kept by any agent (not necessarily running in the same host). The first two levels are usually present in object-oriented languages. The contribution of the DPSK+P system lies on the third level, which incorporates the distributed aspect. In this third level, the system must also provide a remote method activation facility, so that objects kept by one agent can activate methods defined in objects kept by other agents and, therefore, manipulate shared data.

In order to clarify these aspects, let's analyze the following classes hierarchy presented in figure ???. The private and protected level of the *Student* class define the methods and attributes visibility for objects kept by one agent or program. The DPSK+P system extends this right to other agents. It is possible, for instance, that attributes of the *Student* class kept by agent 1 be accessed by objects of *Student* class kept by agent 2, and vice-versa. This sharing of data will constitute the so called shared level. In a certain way, the goal is to extend the public level

to other agents, located at the same host or not, and developed in the same object-oriented programming language or not.

figure=shared_{dpskpp}.eps, height = 10cm

Figure 1: Shared Classes in the DPSK+P System

Some built-in special classes were created for control purposes. These classes are responsible for activating, finishing, interrupting, blocking and resuming agents, as well as for maintaining the consistency of the shared data. A more detailed description of the kernel is out of scope of this paper, but it can be found in [?] [?].

3 Debugging Parallel Systems

The classical debugging and monitoring techniques usually can't be directly applied to parallel systems. The reason is that parallel systems present additional difficulties when compared to their sequential counterparts. The main difficulties are a great number of foci of control, the great quantity of data to be analyzed, the non existence of a global state, nondeterministic processing and the intrusive nature of a monitoring or debugging facility. There are three basic approaches to overcome these difficulties [?] [?]:

- *traditional parallel debuggers*, where a sequential debugger is associated to each process being executed. Normally, there is an extra effort of coordination, and sometimes a higher level tool may be created in order to control the data flow between the sequential debuggers;
- *static analysis techniques*, based on data flow analysis of parallel programs, which detect possible errors during compilation time;
- *event based debuggers*, which define some high level constructs to represent communication and control actions between processes.

Events can be defined as atomic actions that are visible above the scope of one agent. In message based systems, sending or receiving a message constitutes an event, while an access to a shared memory position constitutes an event in shared memory based systems. More flexible systems allow the user to limit the quantity of events he wants to analyze at a given moment (ignoring those which aren't useful), as well as to create high level events by combining primitive ones [?].

On the other hand, regarding the man-machine interface, there are basically four different techniques for displaying information in such debugging and monitoring tools [?]:

- *textual presentation*, which is used when the output display is a non-graphical one. In this case, there is no explicit representation of time;
- *time-process diagrams*, where time is explicitly represented on one axis and the occurrence of events on the other. They are extremely useful when one desires to prioritize the evolution of events in time;
- *animation of processes*, where both dimensions are spatial and each process has associated to it a certain portion of the display. The processes are represented by graphical figures or icons, and the occurrence of events is represented by lines that appear and disappear between processes. These lines may change their color or width in order to represent some particular event. Therefore, the display corresponds to one instant in time, which gives a good picture of the system global state at that particular moment;

- *multiple windows systems*, which could offer both the facilities found on the two techniques described above. For instance, one can exhibit the control flow in two windows: one prioritizing the temporal evolution aspects and other prioritizing the global state representation approach.

These techniques can be used in any level of parallelism, from simple multitask systems to massively parallel multiprocessors. Finally, there is some current research in hardware based debuggers and real time debuggers [?].

4 A Monitoring Facility for the DPSK+P System

In order to construct a monitoring facility for the DPSK+P system, we have chosen the event based technique described in the previous section. *Elementary events* were defined, which correspond to **control** and **communication actions** between agents [?]. Moreover, information about shared classes and instances are also provided for the user.

The *control events* defined by the monitoring facility correspond to activating, suspending, resuming, interrupting, finishing and aborting an agent processing. Other events, corresponding to a remote method activation by an agent, are also provided.

All the communication flow between agents in the DPSK+P system is made by reading and writing shared objects. The consistency maintenance of these shared objects can be controlled by the kernel [?], which uses a transaction mechanism, if the user wishes to. The *communication events* defined by the monitoring facility refer to starting, interrupting and finishing a transaction and to read and write operations in the shared memory.

The shared memory contents (the shared classes and their instances) are also treated by the monitoring facility. A browser was constructed which enables the user to visualize the hierarchy contents and instances defined for each shared class.

Regarding the man-machine interface, the animation of processes technique described in the previous section was elected for the exhibition of elementary control and communication events referred above. In order to minimize the lack of information about the temporal evolution of the events, the original technique was slightly modified, in such a way that the last control or communication event related to each agent and shared class is maintained. This is equivalent to maintain a one event temporal evolution memory for each entity being monitored.

4.1 Exhibition of Control Events

In the control events exhibition module, each agent is represented by a circle, located in a certain display position, as shown in figure ??.

figure=mtctrl.eps,height=9.5cm

Figure 2: Exhibition of Control Events

The circles are filled with different colors, representing each one a different control event occurrence. Besides that, lines and arrows link each two circles, logically representing the link between the active and passive agents involved in the control action being represented. In the case of remote method activation and interruptive control actions, extra information is provided in textual form, concerning the name of the method being called or processed and the group

and signal number of the interruption, respectively. Whenever a new agent is activated, it will be exhibited by the monitoring facility, if the limit of simultaneous visible agents is not exceeded. The user can, however, control the agents' visibility by specifying those which he wants to observe at a particular moment. This effect, which corresponds to a filtering technique, is extremely useful in distributed problem solving, when the user is interested in observing some subproblem solving activity which requires only a certain number of agents.

4.2 Exhibition of Communication Events

In the communication exhibition module, each agent is represented by an horizontal line and each shared class by a rectangle. The read and write operations in shared objects are represented by lines and arrows that link agents to classes, as presented in figure ???. The tags associated with the lines refer to the instances identification numbers. The transaction occurrences, by their way, are visualized by colors that fill the rectangles corresponding to the classes defined in the transaction. The last operation related to each shared class is drawn in a different color, so that the last actions corresponding to each shared class are visible until a next action occurs.

figure=mtcomm.eps,height=8.0cm

Figure 3: Exhibition of Communication Events

This exhibition method introduces a powerful framework to represent effective communication actions between agents in shared memory based systems. Attempts to represent communication actions in message based models are frequently found in the literature, but this doesn't occur for shared memory based systems. This fact can be explained because each shared memory position is a potential communication channel in such a model, which makes it infeasible to construct an efficient and concise representation, notably for practical systems. In the DPSK+P system, *this is possible by the use of object-oriented techniques in the shared memory contents*. The specification of the instances accessed by the different agents is extremely compact, being represented only those linked with the last communication events that have occurred in each shared class. Although this technique can't provide a direct inspection of the instances contents, it should be viewed as an abstraction mechanism for high-level monitoring purposes. This abstraction mechanism is able only to exhibit the communication events in a compact way by limiting the quantity of logical channels simultaneously presented. Moreover, the shared class browser, which is further described, allows the user to inspect the instances contents whenever a processing fault that changes the system normal operation occurs.

4.3 Exhibition of Shared Classes

The monitoring facility allows the user to visualize the shared classes and their instances in three different abstraction levels.

The first level presents the shared class hierarchy. If the user needs to examine a particular shared class content in detail, he can activate a second level, as presented in figure ???. Finally, a third level provides the user with information about the instances created from each shared class. By the use of a mouse, the user can navigate through the shared class hierarchy, changing the root class being momentarily exhibited. By pressing the mouse buttons, the user can also modify the abstraction level, as described above.

figure=mthier.eps,height=9.6cm

Figure 4: Exhibition of Shared Class Contents

4.4 The Monitoring Facility Structure

The monitoring facility for the DPSK+P system was designed in a modular way, and it is composed of three modules: an *exhibition module for control events*, an *exhibition module for communication events* and a *shared class browser*. It was designed to be used in a multiple window system, in such a way that there is one window associated to each module, which makes it possible for the user to visualize concurrently the three modules, if he wishes to. Moreover, a modular design facilitates the incorporation of additional facilities in the future, like a time-process diagram or a high-level events editor.

From the point of view of its dynamic behavior, the monitoring facility is composed of three processes running concurrently, which correspond to each module described above. This approach permits the concurrent updating of the three windows, even if some of them are partially obscured by some of the others. On the other hand, this distributed approach allows the user to activate simultaneously more than one instance of each module, in the same host or not, if he wishes to analyze simultaneously more than one subproblem solving activity.

From the DPSK+P kernel point of view, the monitoring facility behaves as a very special group of agents. When notified that these agents are running, the kernel starts to send to them all the control and communication events that are presently occurring in the problem solving activity.

4.5 Implementation

The monitoring facility was developed in UNIX-like workstations using originally the C programming language. A second version for the C++ programming language has just been provided. The XWINDOWS v.11 system was used both as a window manager and as a generator of graphical figures.

5 Further Research

Distributed Artificial Intelligence is a subfield of Artificial Intelligence concerned with concurrency and distribution in AI computations, at many levels. Historically, interests in this field may be divided in two primary arenas: *Distributed Problem Solving*, which was already described in section 1 of this paper, and *Multiagent Systems*. Research in Multiagent Systems is concerned with coordinating intelligent behaviour among a collection of possibly pre-existing autonomous agents, i. e., how they can coordinate their knowledge, goals, skills and plans jointly to take action or to solve problems [?] [?].

Although designed in a distributed problem solving context, the DPSK+P system, and its monitoring facility, may be used in a multiagent context. Current research in multiagent systems at the LIFIA laboratory is to design different internal models of agents, which will be integrated by using a high-level interagent communication protocol [?]. The DPSK+P system is currently being used as a basis for the development of this protocol. Its monitoring facility is being of great help during the implementation phase of this protocol, and will be used in the future in order to analyse the social behaviour of the community of agents.

6 Conclusions

Distributed problem solving and object-oriented techniques can be used in the future to help the design of large-scale software organizations. These organizations will have as design requisites the support for reusing and evolving software. During the software design cycle of these organizations, tools like debuggers and monitors will be extensively used.

The monitoring facility designed for the DPSK+P system [?] consists of a set of three agents, responsible for the exhibition of control events, communication events and shared classes contents. The classical technique of animation of processes was slightly modified, in order to minimize the lack of information about the temporal evolution of the events. From the implementation point of view, it was designed in both a modular and distributed way, so that possible extensions will be easily designed.

Finally, this work also introduces a powerful framework to represent communication events in shared memory based systems, by using an object-oriented abstraction. This framework permits that the treatment of events of this kind be made in a concise and efficient way.

7 Acknowledgments

This work was supported by EPUSP-PCS and ITA-IEE, Brazil. Our development environment was partly supported by CEPTEL (Centro de Pesquisas em Engenharia Elétrica), Brazil and CMU, USA. Finally, we would like to thank Yves Demazeau, Olivier Boissier and Matthew Turk (LIFIA-IMAG) for their useful comments during the revision of this paper.

The first author's current research is being supported by EPUSP and FAPESP - Fundação de Amparo à Pesquisa do Estado de São Paulo, Grant Number 91/1943-5.

References

- [BATE90] BATES, P. Debugging heterogeneous distributed systems using event-based models of behavior. **SIGPLAN Notices**, v.24, n.1, p.11-22, Jan. 1989.
- [BERT92] BERTHET, S.; DEMAZEAU, Y.; BOISSIER, O. Knowing each other better. In: **INTERNATIONAL WORKSHOP ON DISTRIBUTED ARTIFICIAL INTELLIGENCE**, 11, Glenn Arbor, Michigan, 1992. **Proceedings**.
- [BOND88] BOND, A. H.; GASSER, L., ed. **Readings in distributed artificial intelligence**. San Mateo, Morgan Kaufmann Publishers Inc., 1988.
- [CARD92] CARDOZO, E. et al. **DPSK+P User's Manual: C++ Interface**. ITA, São José dos Campos, 1992.
- [CARD91] CARDOZO, E. et al. **DPSK+P: a distributed problem solving kernel for object-oriented applications**. Internal report. ITA, São José dos Campos, 1991.
- [CARD87] CARDOZO, E. **DPSK: a kernel for distributed problem solving**. Pittsburgh, 1987. 144 p. PHD Thesis - CAED, Carnegie Mellon University.
- [DEMA91] DEMAZEAU, Y.; MULLER, J. P., ed. **Decentralized A. I. 2**. Amsterdam, Elsevier Science Publishers B. V., 1991. / Proceedings of the 2nd European Workshop on Modelling an Autonomous Agent in a Multi-Agent World, St. Quentin en Yvelines, France, August 90 /

- [DOWE89] MCDOWELL, C. E.; HELMBOLD, D. P. Debugging concurrent programs. **Computing Surveys**, v.21, n.4, p.593-622, Dec. 1989.
- [ENGE88] ENGELMORE, R.; MORGAN, T., ed. **Blackboard systems**. Reading, Addison-Wesley, 1988.
- [FOX81] FOX, M. S. An organizational view of distributed systems. **IEEE Transactions on Systems, Man, and Cybernetics**, v.11, n.1, p.70-80, Jan. 1981.
- [JOYC87] JOYCE, J. et al. Monitoring distributed systems. **ACM Transactions on Computer Systems**, v.5, n.2, p.121-50, May 1987.
- [LARN90] LARNER, D. L. Factories, objects & blackboards. **AI Expert**, v.5, n.4, p.38-45, Apr. 1990.
- [SICH91] SICHMAN, J. S. **Uma ferramenta de monitoração para um núcleo de resolução distribuída de problemas orientado a objetos**. São Paulo, 1991. 166 p. MS Thesis - Escola Politécnica, Universidade de São Paulo. /In portuguese/
- [STEF86] STEFIK, M.; BOBROW, D. G. Object-oriented programming: themes and variations. **The AI Magazine**, v.6, n.4, p.40-62, Jan. 1986.
- [TAKA90] TAKAHASHI, T.; LIESENBERG, H. K. E. **Programação orientada a objetos**. São Paulo, IME-USP, 1990. /Textbook presented at 7a. Escola de Computação, São Paulo, 1990/ /In portuguese/