Coupling Symbolic and Numeric Programs in Large-Scale Software Organizations

by

S. Talukdar, E. Cardozo, and G. Podnar

EDRC-05-14-87

Coupling Symbolic and Numeric Programs in Large-Scale Software Organizations

Sarosh Talukdar, Elcri Cardozo, Gregg Podnar
Engineering Design Research Center
Carnegie Mellon University
Pittsburgh, PA 15213

July 1987

Abstract

This paper describes an environment for building organizations of distributed, collaborating programs. Interaction of difgimiiflr problem-solving agents (symbolic as well as numeric programs) have been provided for with DPSK, a distributed problem solving kernel. DPSK has evolved from a traditional blackboard architecture to incorporate a number of collaborative mechanisms, called lateral relaricms, adapted from hianan organizational theory. This paper traces the evolution of DPSK and describes its principal features.

1 INTRODUCTION

Many large-scale software organizations will come under increasing pressure to add expert systems and other nonalgorithmic programs to augment their of algorithmic, number-crunching programs. An example area is Energy Mangement Systems (EMS). Two sources of this pressure - fault diagnosis, and real time security assessment problems — serve to show the difficulties inherent in the integration of dissimilar problem-solving agents. These are briefly described below. The remainder of the paper is devoted to the organizational aspects of combining independent and disparate programs.

1.1 Fault Diagnosis

The fault diagnosis problem is concerned with interpreting the causes of alarms that are reported to the real time operators of power systems. A large storm can cause hundreds of these alarms to appear in a matter of minutes. Some are triggered by faults. Others result from misoperations of equipment and still others from reporting errors. There is now considerable evidence that automatic processes for generating hypotheses to explain these alarms must combine expert systems, number-crunching programs and distributed processing [3], [14]. For instance, the approach used in [3] assembles hypotheses by using two crews of programs (Fig. 1). The first crew selects events (disturbances, equipment malfunctions and other errors) with which to expand incomplete hypotheses. The second crew evaluates the selected events and rejects any that make little or no progress towards explaining the given set of alarms. Some of the members of these crews are expert systems written in OPS5, others are algorithmic programs written in C.

12 Security Assessment

The real time security assessment process involves three steps:

1. select a set of contingencies
2. simulate or otherwise predict the response of the system to these contingencies
3. interpret the responses.

- select several programs
- run the selected programs concurrently.

In some cases it might be desirable to ensure that these steps are strictly separated in time, while in others, it might be desirable to allow them to overlap and proceed in parallel.

Having repeatedly been reminded of the importance of distributed approaches to engineering tasks (see [5], [12], [14], for instance), we set out some yean ago to produce an environment for implementing such approaches. The first result was a set of tools called COPS (Concurrent Production System) [8]. COPS is written in OPSS and provided facilities for creating multiple blackboards distributed over a network of computers. Programs communicate with remote blackboards via "ambassadors" (Fig. 2). Each ambassador is a set of rules that represents the interests of its parent program. The computational cycle for each processor remains the same as in the uni-processor case except that the first step may result in the selection of a program that is an ambassador. When this happens, the second step results in an exchange of data between processors.

In working with COPS, certain differences in the control issues for uni- and distributed processing have become clear to us. In the uni-processor case, the paramount control issue is deciding which piogiam to ran. In the distributed case, this issue becomes progressively less important with increase in the relative number of processors, and disappears entirely when each program has its own processor. Instead, the paramount control issue becomes the selection of mechanisms for collaborations among programs. What is the range of alternatives for these mechanisms? As previously mentioned, we will use human organizations as our models in identifying alternatives. The next section will list these alternatives. Subsequent sections will discuss how to implement them in software.

2 DESIGN ALTERNATIVES

2.1 Structural Representations

The structures of both human and software organizations can be represented by directed graphs with two types of nodes and three types of arcs (Fig. 3). The nodes represent agents and databases; the arcs represent channels for commands, signals and data flows.

The command-arcs establish lines of authority and usually flow from the top down. They provide routes for messages like: "do this subtask," "send me a progress report," and "stop."[1]* If only the command arcs and agent-nodes are preserved, the graph degenerates to a traditional organization chart The signal arcs usually flow from the bottom up. They provide routes for feedback, particularly, to report unexpected happenings like commands that cannot be executed.

The data flow arcs represent the channels provided for the movement of information other than commands and signals.

22 Operations

By operations we mean the activities of agents over time. Consider agents A, B and C from Fig. 3. In general, they can work concurrently, as in Fig. 4. Since they share a database, they can exchange information. These exchanges can occur at preplanned points in time, as happens between A and B, or spontaneously, as happens a little later among A, B and C.

Much of the cooperative activity in human organizations relys on spontaneous (asynchronous, opportunistic)

communications. Software organizations can also benefit from such communications. By way of a simple example, consider the task of solving a sec of nonlinear algebraic equations. Many numerical methods are available for this task, but no single method can be relied upon to always work well. One way to deal with this situation is to arrange for several methods to search for solutions in parallel, exchanging clues and other useful bits of information as they find them (Le^ spontaneously). As a result, solutions are found Caster than if only preplanned communications are allowed, and also, solutions are found in cases where the methods working independently would fail hopelessly [12].

2 J Contingency Theory and Lateral Relations

Contingency theory has been derived mainly from empirical studies of large human organizations and consists of recommendations for structures that either prevent the occurrence of contingencies or facilitate their handling [7]. The recommendations can be divided into two categories: adding resources, and improving communications. The latter category can be further divided into: strengthening the vertical information system, and creating lateral relations. To explain these terms, consider the essential mode of operation of an organization which is to recursively apply a cycle with three steps: decompose a task into subtasks, perform the subtasks, and finally, integrate the results. The natural organizational structures for performing these cycles are hierarchical with charts that take the forms of trees. TTie natural lines of information flow in these trees are vertical. Some improvement in performance of an organization can usually be obtained by improving these vertical channels. However, by fir the biggest improvements in performance, especially in the handling of contingencies, is obtained by establishing lateral relations - mechanisms that support horizontal exchanges of information. Five that seem particularly applicable to software are listed below and illustrated in Fig. 5.

1. Direct contact: a horizontal dataflow or signal arc between two agents at the same level Without a horizontal arc, information to be exchanged between these agents would first have to flow up to a common manager and then back down. Besides taking longer, the information could become distorted along this vertical path.

2. Groups: sets of agents or independent organizations that share data. Markets are a special case of groups. In a market, the shared data include offers to buy and sell services.

3. Representatives: to make known and protect the interests of remote agents.

4. Task forces: when several departments (sets of agents) have overlapping concerns, the pair-wise exchange of representatives can be less convenient and effective than the formation of a task force with members from each department As an example, consider the process of simultaneous engineering for automobile parts. Decisions made during the design stage of these parts can, of course, have profound effects on downstream stages like manufacturing and testing. For instance, a designer may incorporate a feature that is difficult or impossible for the available machinery to manufacture. To prevent such contingencies, a task force is formed with representatives from tooling, manufacturing, testing and other departments. The task force oversees the designers* efforts and intervenes when the interests of its parent departments are threatened.

5. Matrix management in which two or more command arcs terminate in a single node. This arrangement allows A and B to share the services of C (Fig. *56).* Among the benefits are increased reliability (C can be reached through B when A fails) and quick response (B can intervene even when C is working for A). Among the costs is the possibility for C to become confused.

3 DPSK (DISTRIBUTED PROBLEM SOLVING KERNEL)

3.1 Overview

DPSK provides the software builder with a small set of primitives. These primitives have been designed to be inserted in the instructions of an expandable set of languages. At present, this set is: C, Fortran-77, OPS5 and Lisp (Franz and Common). With the primitives, software builders can readily synthesize all the alternatives from the

preceding section and ihereby, assemble arbitrary organizations, made up of agents written in a variety of dissimilar languages, and distributed over a network of computers. In theory, the numbers of programs and computers can be arbitrarily large.

DPSK itself, is written in C for networks of computers running Unix™ 4.2 bsd. Internally, DPSK employs a shared memory that is distributed over the participating computers.

We elected to build DPSK around a shared memory for two reasons. First, blackboards have demonstrated that shared memory is very useful in assembling communities of collaborating programs in uni-processors. (In fact, we feel that shared memory is by far the best feature of the blackboard idea). Clearly, the characteristics that make shared memory attractive in uni-processors can only become more attractive in distributed processor environments. Second, the representations that we prefer in thinking about organizations rely heavily on shared memory (cf. Figs. 3 and 4). It is easier to build a system that closely parallels one's favorite representations. However, before finalizing the choice of shared memory we also considered message based systems and remote procedure calls. They were rejected because we felt they would be far less powerful [2].

32 Primitives

DPSK contains 12 primitives that can be divided into four categories — commands, synchronizers, signals, and transactions. The primitives themselves are listed in the appendix. Brief descriptions of their categories are given below.

The command primitives are used to activate and control programs. An agent can "run," suspend," "resume," or "kill" other agents in any of the processors in the network. This also allows for any number of program clones to be created and run in parallel.

The synchronization primitives are used to create and check for the occurrence of "events." The events enable concurrent processes to be coordinated. For instance, to ensure that some activity in Agent A finishes before Agent B is allowed to begin, one would insert primitives into A at the appropriate point to assert an event X, and in the beginning of B to wait for the assertion of X.

The signal primitives are used to signal the occurrence of a contingency or to interrupt the execution of preselected groups of processes and cause them to execute portions of their code designated to handle such exceptions.

Transaction primitives are used to structure and access the shared memory. (A transaction is a time stamped operation designed to maintain consistency and correctness in distributed databases [4,10].) The data to be shared is stored in Objects, each of which consists of a Class designation followed by Slots for attribute-value pairs. The values can be character strings, integers or floating point numbers. For instance:

```
{line
 [name HB]
 [sb21
 [eb8]
 [resistance 0.09854]
 [reactance 1.232]}
```

is an object of class "line" with five attributes. Objects are accessed through pattern matching. For instance, the pattern: { line [eb 8]} would access the above object and all the others in shared memory that belong to class "line* and have "eb"=8.

3.3 Usage

The synchronization, and transaction primitives are used to synthesize operating alternatives and those structural alternatives that require shared databases. The command and signal primitives are used to synthesize the remaining structural alternatives. This covers all possibilities except "dynamic rewiring." The present version of DPSK provides no special facilities for the dynamic reconfiguration of an organization.

4 EXAMPLES

4.1 A Simple Distributed Team

Consider the general problem of searching a tree for a solution, given a number of computers and a program called S. Suppose that S, can identify the children of a given node and determine if one of them is the desired solution. One way to tackle this search problem is by representing nodes by objects of the form:

{ Node [Number 12] [Parent 5] [Children (16 17 18)] - - - }

Using the command primitives, a small program sets copies of S to run in each of the available computers. These parallel agents are controlled by this program whose essential functions are:

1. Identify all the unexpanded nodes by retrieving objects that match the pattern: **{ Node [Children nil] }**
2. Assign a searcher (copy of S) to each unexpanded node by adding a slot to the node-object with the searcher's name in it.

Each searcher retrieves nodes to which it has been assigned, expands them and adds the new nodes so obtained to the shared memory. When the desired solution is encountered, a signal may be used to identify this situation, and the searchers terminated.

In all, about 30 lines of new code have to be written, and this number is independent of the number of computers used [2]. We believe that a comparable system written without DPSK in Lisp or C would require at least ten times as much code.

4.2 A Distributed Diagnostician

An example of a specific problem is the synthesis approach to the fault diagnosis problem (c.f. section 1.1). This has been implemented using DPSK on four networked microvaxen. The results are shown in Fig. 6. With three or more microvaxen, the calculation times are small enough to be useful in real time operations.

5 CONCLUSIONS

There are two distinct types of benefits that can be gained from distributed processing. The first is widely recognized – modular, expandable computer networks that allow the amount of computing power that is made available to be easily increased. The second is not well known in software engineering but is taken for granted in building human organizations – namely that many difficult tasks have parallel decompositions that yield easier subtasks than their serial decompositions. In particular, decompositions that promote opportunistic collaborations among parallel subtasks seem to provide easier and better ways to solve problems than serial decompositions.

When a single processor is used to house a number of programs, the principal control issue is deciding which of the programs to run. With distributed processors, however, some or all of the programs can run simultaneously and the principal control issue becomes how to arrange collaborations among them.

Case studies of human organizations indicate that different tasks benefit from different collaborative mechanisms. This paper lists several mechanisms, adapted from human organizational theory, that seem especially suitable for software **organizations. These** mechanisms, along with a variety of other design alternatives, have been made **available to the software builder** through a tool kit calkd DPSK.

We feel that the best way to develop large-scale software organizations which integrate numeric and symbolic problem solving agents, and to expand the capabilities of existing systems to include nonalgorithmic programs (problems such as the EMS fault diagnosis and security assessment), will be to make the software couplings through an optimized version of DPSK, and networked workstations to provide for effective concurrent operation.

6 REFERENCES

1. Buchanan, B.G. and Shortliffe, E.H.
Rule-Based Expert Systems.
Addison-Wesley Publishing Co., New York, 1984

2. Cardozo,E.
DPSK: A Distributed Problem Solving Kernel.
PhD thesis, Depc. of Electrical and Computer Engineering, Carnegie Mellon University, January, 1987.

3. Cardozo, E., and Talukdar, S.N.
A Distributed Expert System for Fault Diagnosis.
in *Proceedings cfthe IEEE Power Industry Computer Application Conference.* Montreal, Canada, May, 1987.

4. Date,CJ.
An Introduction to Database Systems.
Addison-Wesley Publishing Co., Reading, Massachusetts, 1983.

5. Elfes,A.
A Distributed Control Architecture for an Autonomous Mobile Robot.
International Journal for Artificial Intelligence in Engineering 1(2), October, 1986.

6. Fox, M.S.
An Organizational View of Distributed Systems.
IEEE Transaction on Systems, Man, and Cybernetics SMC-11(1), January, 1981.

7. Galbraith,J.
Designing Complex Organizations
Addison-Wesley Publishing Co., Reading, Mass, 197S.

8. LeaaL. and Talukdar, S.N.
An Environment for Rule-Based Blackboards and Distributed Problem Solving.
Artificial Intelligence, 1(2), 1986.

9. Simon, HA.
The Design of Large Computing Systems as an Organizational Problem.
Organisatiewetenschap en Praktijk.
RE. Stenfert Kroese B.V., Leiden, 1976.

10. Smith, R.G., and Davis, R.
Frameworks for Cooperation in Distributed Problem Solving.
IEEE Transactions on Systems, Man, and Cybernetics SMC-11(1), January, 1981.

11. Specter, A X, Daniels, D., Duchamp, D., Eppinger, J l, and Pausch, R.
Distributed Transactions for Reliable Systems.
Technical Report CMU-CS-85 -117, Dept of Computer Science, Carnegie Mellon University, 1985.

12. Talukdar, S.N., Elfes, A., and Pyo, S.
Distributed Processing for CAD - Some Algorithmic Issues,
Research Report DRC-18-63-83, Design Research Center,
Carnegie Mellon University, 1983.

13. Proceedings of the Boeing Workshop on Blackboard Systems,
Seattle, Washington, July 1987.

14. Talukdar, S.N., Cardozo, E., and Perry, T.
The Operator's Assistant - An Intelligent, Expandable Program for Power System Trouble Analysis, IEEE PAS Transactions, Vol PWRS-1, No *3*,
August 1985.

15. Christie, R., Talukdar, S.N.
Expert Systems for On Line Security Assessment
In *Proceedings of the IEEE Power Industry Computer Application Conference.*
Montreal, Canada, May, 1987.

16. Talukdar, S.N., Christie, R.
On Line Security Assessment, EDRC Report, May 1987.

7 APPENDIX

DPSK Primitives

A problem-solving Agent has a set of twelve primitives for all interaction with the Distributed Problem Solving Kernel (DPSK). These primitives are callable from C, OPS5, Common Lisp, and Franz Lisp. A subset is available to FORTRAN77 programs.

Communication Primitives:

Any Agent may access the shared database.

- **Begin-Transaction (class, mode)**
  Initiates access to a portion of the shared database designated by class. The mode can be READ or WRITE. A nisnber of Agents can simultaneously have READ-access to a class in the database, but only one Agent may hold WRTTE-access at a time. This call returns a Transaction-ID which is used by other primitives to designate this database access session.
- **Op-Transaction (Transaction-ID, type, pattern)**
  Facilitates all operations on the shared database. Objects can be CREATEd, READ, UPDATEd, and DELETEd, depending on the type of access specified Access is made to all Objects in the class which match a pattern of <ATTRIBUTE - VALUE> pairs.
- **Abort-Transaction (Transaction-ID)**
  Aborts a transaction currently in progress (not commonly used).

- **End-Transaction (Transaction-ID)**
  Terminates this database access session.

Process Control Primitive:

An Agent may startup and control other Agents.

- **Proc-Control (agent, action, processor)**
  Facilitates run control of agents in any processor. Agents may be RUN, susPENDed, RESUMEd, and fOLLed as indicated by action.

Synchronization Primitives (events):

An Agent may name many different events for synchronization purposes.

- **Afflrm-Event (event)**
  Affirms (or "raises") an event.
- **Check-Event (event)**
  Checks to see if the event is affirmed.
- **Wait-Event (event, sec, ustc)**
  Waits for an event to be affirmed. To designate the length of time to wait, sec, and usec, indicate seconds and microseconds.
- **Negate-Event (event)**
  Negates (or "lowers") an event

Exception Handling (signals) Primitives:

Any number of groups may be named by any Agent. Signals can be any integer number.

- **Set-Group (group)**
  Sets the calling Agent into the indicated group.
- **Set-Handler (handler)**
  Designates the routine within this Agent which will be asynchronously called when this Agent is signaled.
- **Sig-Group (signal, group)**
  Sends this signal to all Agents in the indicated group.

This work **was** supported by the National Science Foundation under its Engineering Research Centers program.

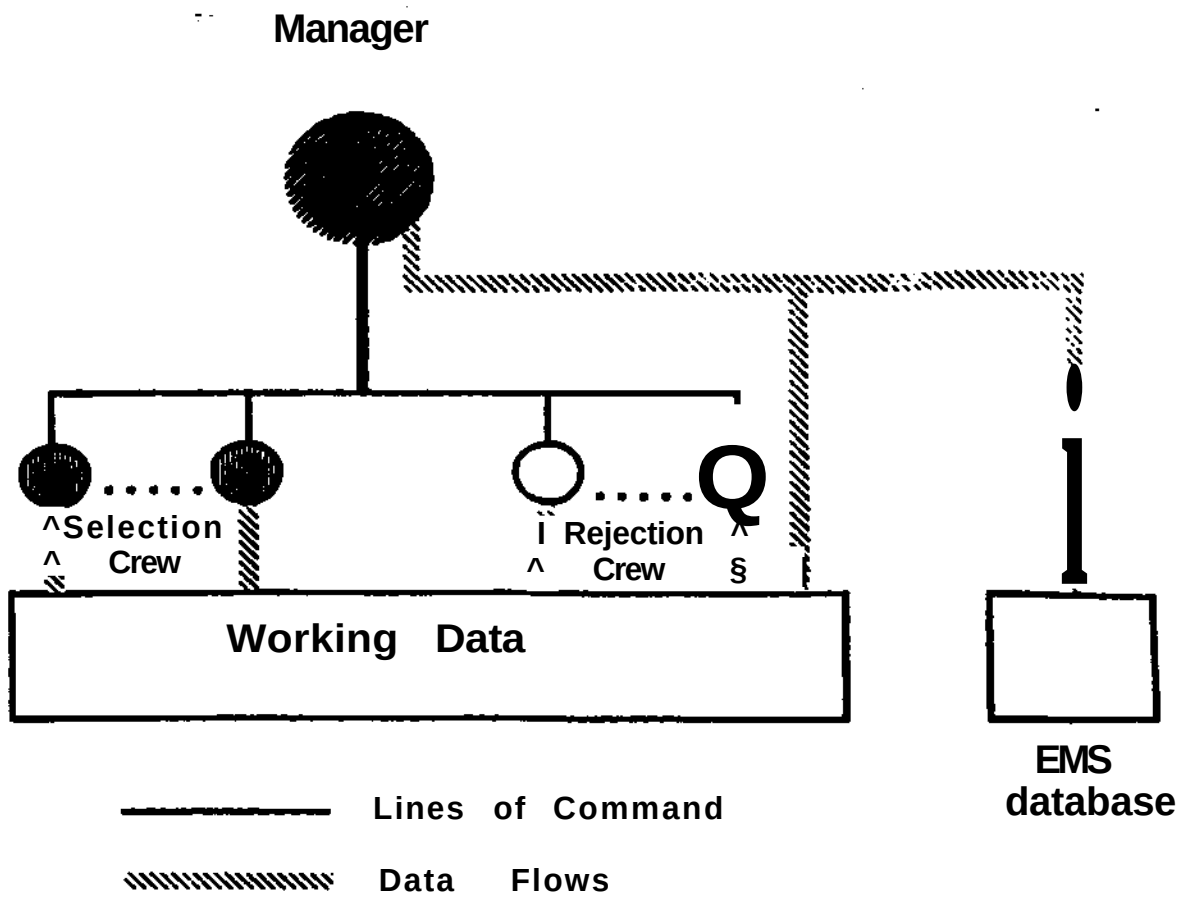


Fig. 1. Organization for patchwork synthesis.

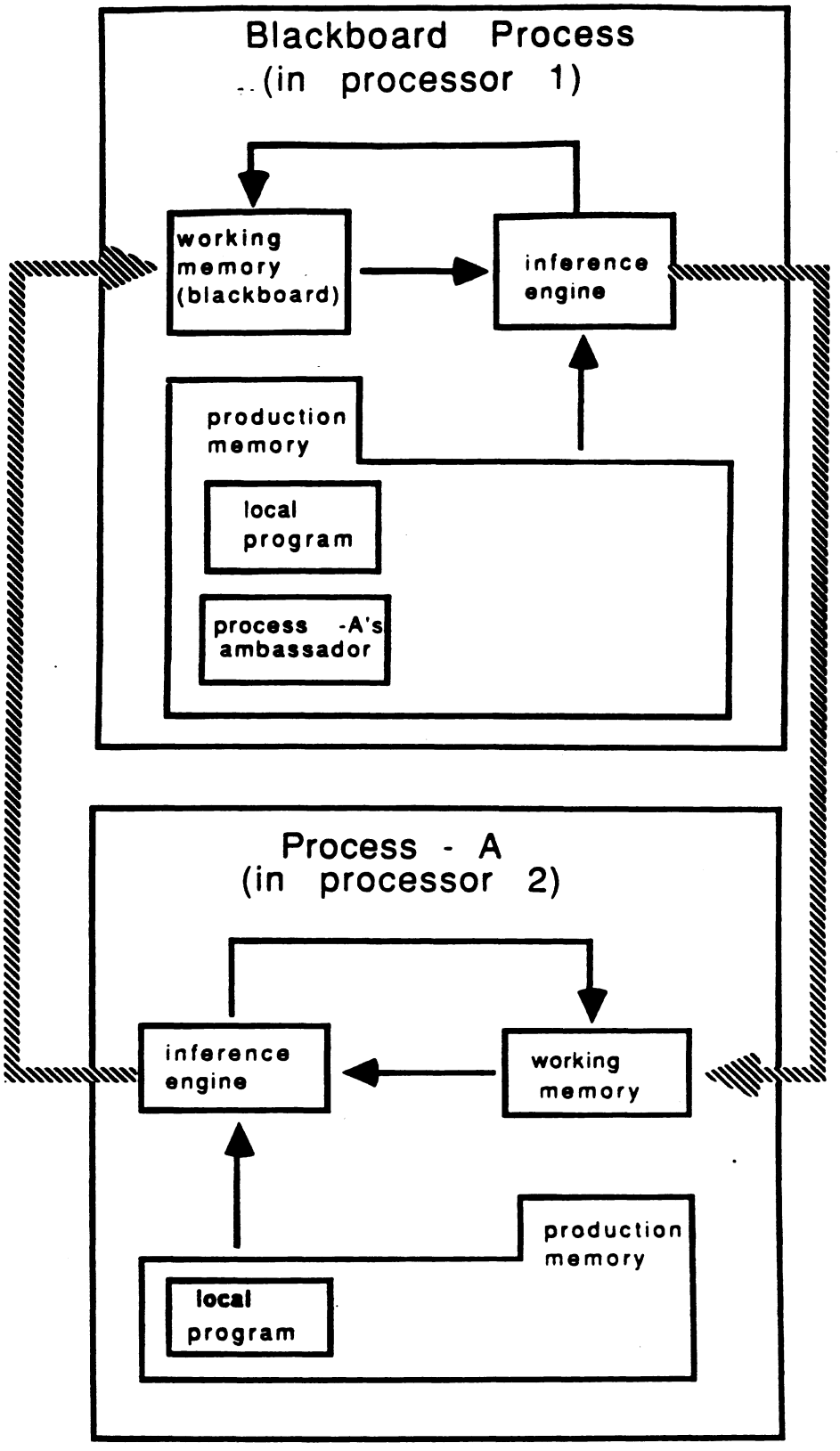


Fig. 2. Ambassadors allow a rule-based process to work as a blackboard. COPS provides the facilities for processes in remote processors to establish and use ambassadors.

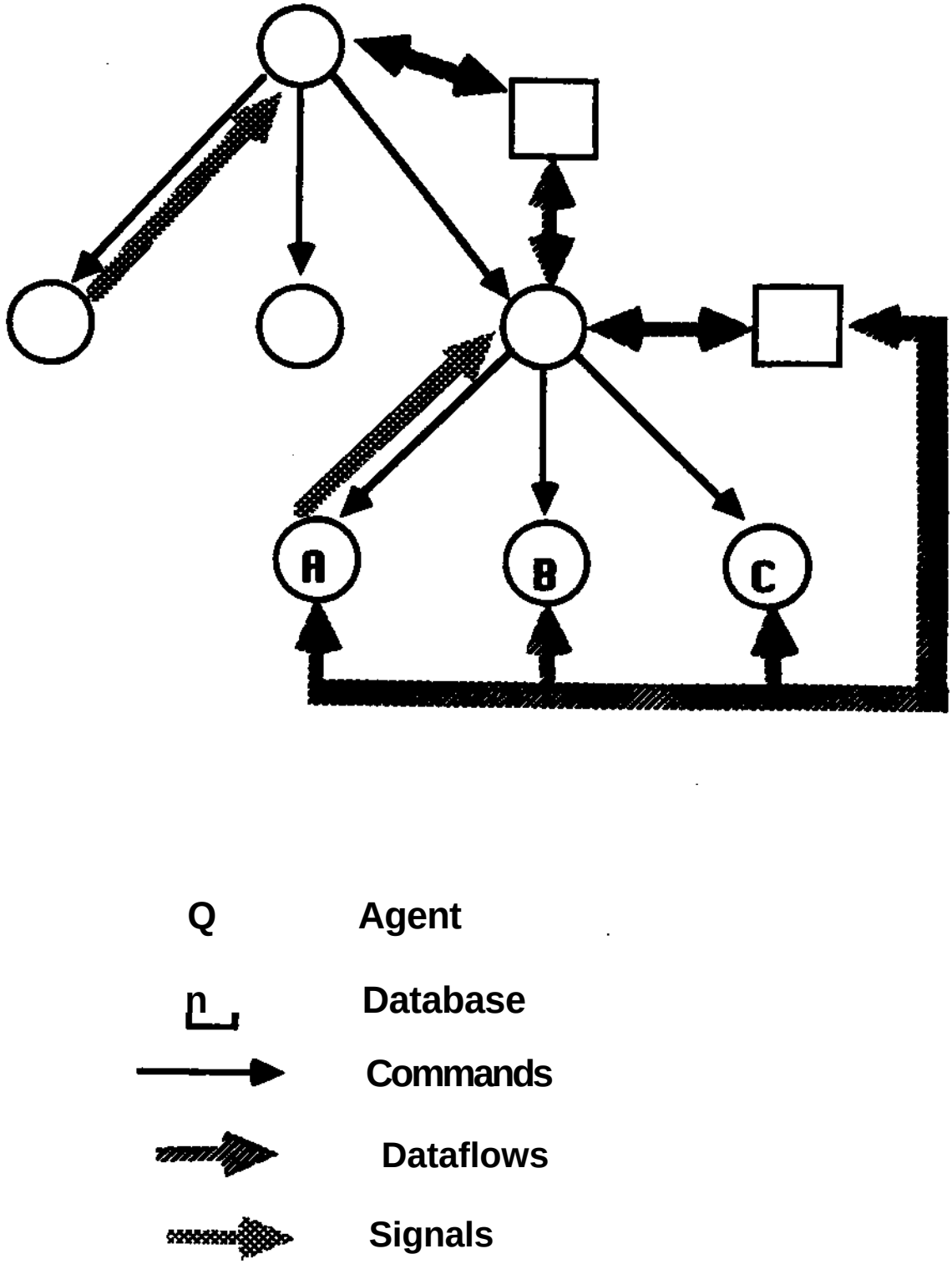


Fig. 3. An organization graph.

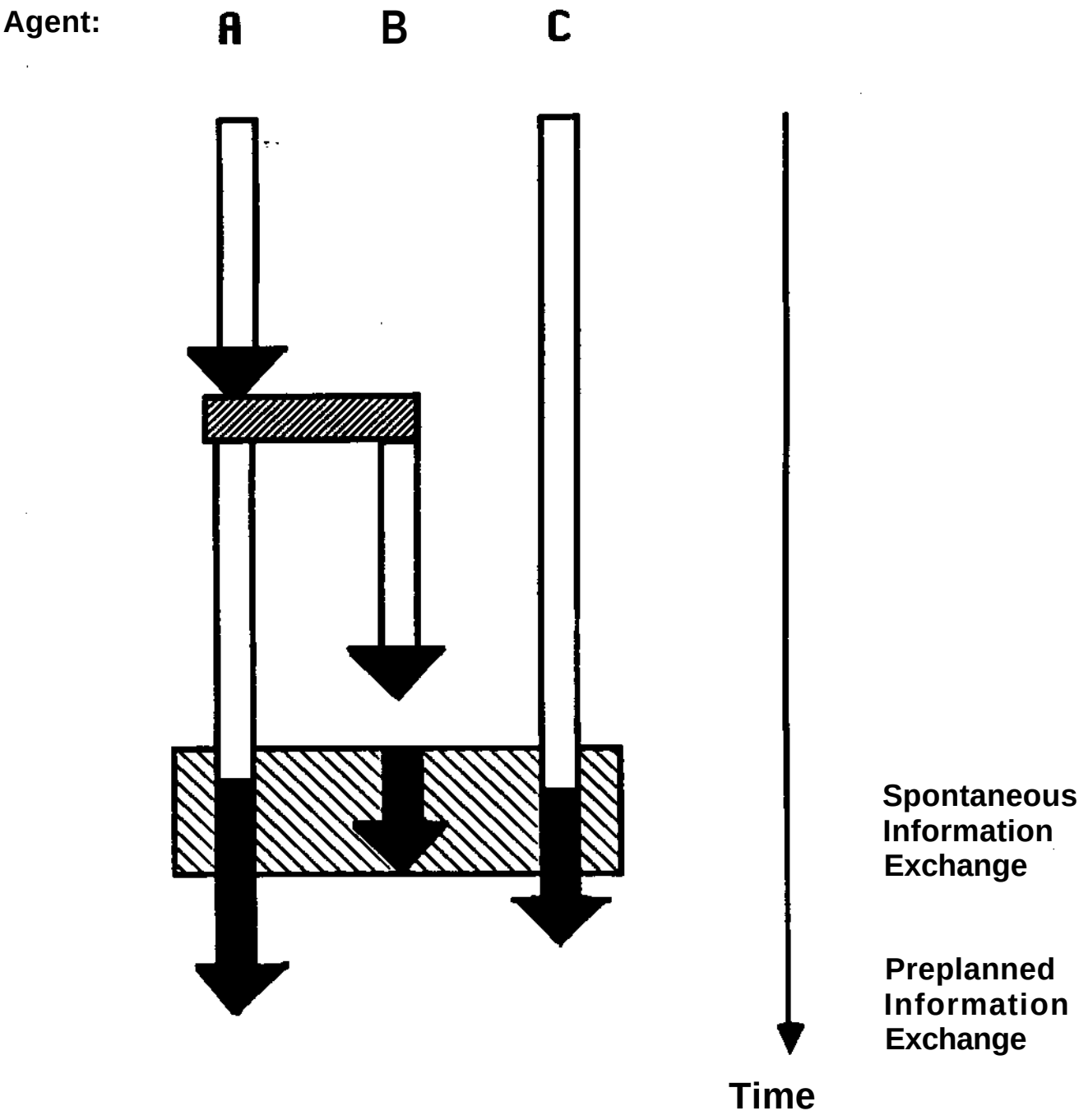
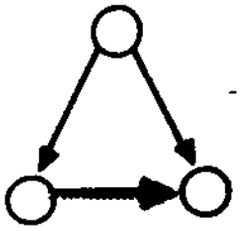
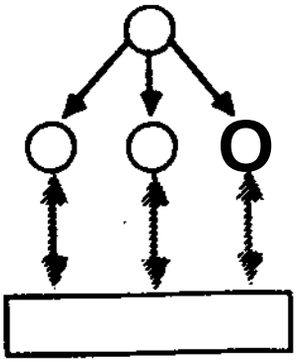


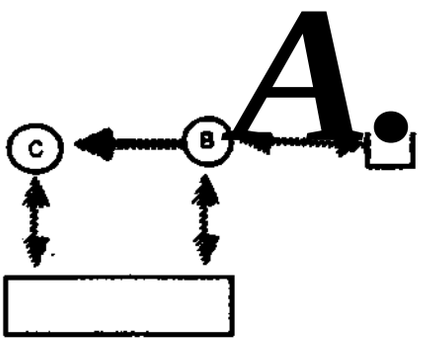
Fig. 4. Two types of collaboration-preplanned and spontaneous. In the latter case the tasks change as a result of the collaboration. (Tasks are denoted by thick arrows.)



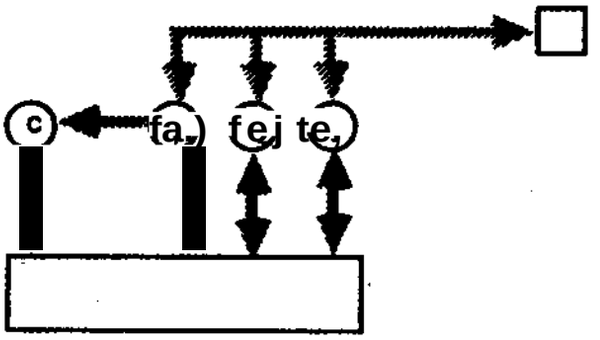
(a) Direct Contact



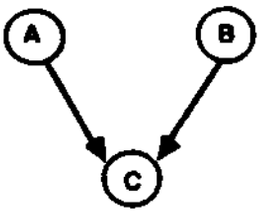
(b) Group



(c) **B serves as A's representative to C. Usually, A and B are In different processors.**



(d) B,. ^ and B_s constitute a task force to oversee the activities of C. B_1 Is the task force leader.



(e) Matrix Management (Dual Authority)

Fig. 5. Some types of lateral relations.

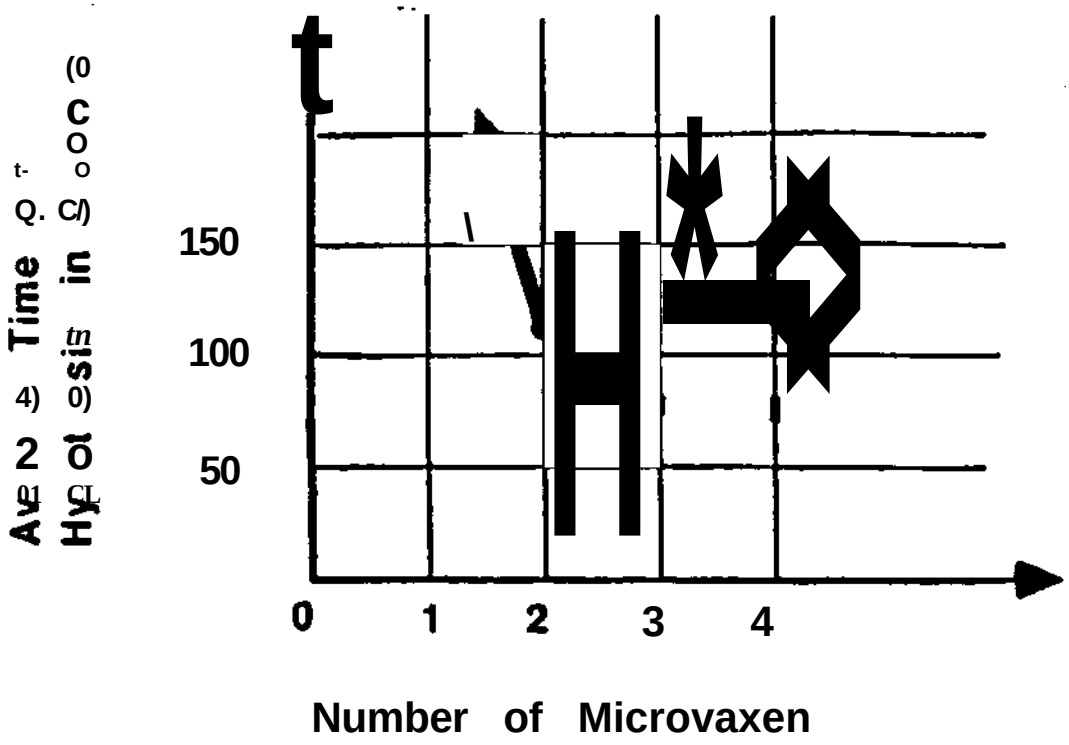


Fig. 6. Time taken to find hypotheses.