Building Large-Scale Software Organizations

by

Sarosh Talukdar, Eleri Cardozo

EDRC 05-21-88

BUILDING LARGE-SCALE SOFTWARE ORGANIZATIONS

Sarosh Talukdar, Eleri Cardozo
Engineering Design Research Center
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper describes DPSK, an environment for building organizations of distributed, collaborating programs. DPSK has evolved from a traditional blackboard architecture to incorporate a number of collaborative mechanisms, called lateral relations, borrowed from human organizational theory. This paper traces the evolution of DPSK, describes its principal features and illustrates its use with some simple examples.

1 INTRODUCTION

Many scientific and engineering communities are desperate for ways to seamlessly integrate large numbers of people and computer tools into systems. Engineering design is just one such area. The numbers of tools available for computer aided design (CAD) is growing rapidly is well into the hundreds in some disciplines. The traditional way of using CAD tools is for humans to serve as go betweens and supervisors. Otherwise, it would be far mores efficient if other tools could serve in those roles, allowing large software organizations to be assembled and combined with human organizations. In the remainder of the paper we will discuss some of the issues involved in putting together such organizations.

1.1 Terminology

- Organization: an information processing system for performing intellectual tasks like designing cars.
- Complex task: a task that decomposes into difficult subtasks that require different problem solving skills. For instance, the task of designing a car which decomposes into designing its outer shape, engine, door systems, manufacturing processes, and so on.
- Agents: the active components of organizations. An agent may be a human or a computer program.
- Uncertainty: the deficit in "up-front knowledge" needed to preplan the operations (activities) of an organization. Some common components of this deficit are: incomplete knowledge of the state of the world, inaccurate predictions of the future, imperfect agents, and unfamiliar tasks. Uncertainty is dependent on both the task and on the structure of the organization used to tackle the task.
- Contingencies: the consequences of uncertainty, namely, the obstacles that arise to block the successful completion of subtasks. For instance, an important problem may turn out to be unsolvable, an algorithm may fail to converge or a vital piece of data may prove to be unobtainable.
- Large-Scale Organization: an organization with many independent agents that can work in parallel, can collaborate and usually have high computation to communication ratios. Later, we will argue that complex tasks and uncertainty call for large-scale organizations.
- Collaboration: the exchange of raw and processed data.

1.2 Human Organizations

In seeking ways to build large-scale software organizations it is well to look to human organizations for guidance. The reasons are two-fold. First, human and software organizations are similar in principle [6], [9]. Second, human organizational theory is much more mature. We would rather borrow techniques from it than reinvent them.

Human organizations are able to complete very complex tasks in highly uncertain circumstances by:

- using large numbers of agent providing a variety powerful mechanisms for agents to collaborate
- employing parallel (concurrent) approaches.

Human organizations routinely assemble very large teams of intelligent agents-hundreds, and sometimes even thousands, of engineers, scientists and managers with widely varying knowledge and skills. A number of mechanisms have evolved to promote collaborations among these agents, making it possible for them to focus their skills on big tasks. These mechanisms rely on lateral channels for information flow that run across the lines of authority. More will be said about these channels later.

Another important aspect of human organizations is the parallel approaches they take to problem solving. The advantages are more profound than mere increases in speed. Some examples will be used to explain. First, consider a task that decomposes into a set of invariant, partially ordered steps. (Meaning that some of the steps can proceed in parallel without changing their outcomes. This is the sort of task that is usually thought of for parallel processing in computers.) Since the steps are invariant, the only gain from parallel processing is a saving in time. Now consider a quite different sort of task-that faced by one team in a football game. The team's members must work in parallel. The overall task would be impossible if only one team member were allowed to be active at a time. The reason, of course, is that the overall task decomposes into a much easier set of parallel subtasks than sequential subtasks. There are many analogous situations in engineering. Designing the different aspects of a product is a good example. If the aspects are tackled in parallel, there is the opportunity for negotiations, compromise and coordination. However, if they are tackled in series, the upstream stages invariably make choices that impose difficult or impossible constraints on the downstream stages.

1.3 Software Organizations

Despite the theoretical similarities between human and software organizations, there are profound practical differences between them, especially along the following dimensions:

- expandability. (While human organizations readily grow to encompass many agents with diverse skills, large-scale software organizations suffer from acute growing pains and are relatively rare.)
- distributed problem solving. (Concurrent, distributed activities with high computation to communication ratios and dynamically varying subtasks seem to be the basic mode of operation of human organizations. In contrast, computer systems that use concurrent computations usually concentrate on the finer grains of parallelism and tasks that decompose into invariant subtasks.)
- collaborative mechanisms. (Human organizations use a much richer set of mechanisms than software organizations).

These differences exist because of a lack of good tools with which to build large-scale software

organizations. Traditionally, there is an overwhelming amount of software effort required to integrate a number of dissimilar software packages. To further illustrate building a problem-solving organization with agents written in programming languages of significantly different orientations; consider the idea of building a human organization with individuals, each of whom is from a very different cultural and educational background, and who speak and write different languages. Coordination is nearly impossible. With the varying styles of problem solving to be expected, there are further difficulties which prevent understanding even when interpreters are employed. (Interpreters are also cumbersome and expensive.) Add to this the problems of one individual trying to understand the information files of another. The metaphor can be carried quite far.

In the last few years, blackboards have emerged as both the principal tool and conceptual form for building large-scale software organizations [13]. In essence, a blackboard is database with a built in set of support facilities that allow it to be shared by an expandable community of programs. The idea is to make important raw and processed data visible to the community. The computational cycle is modeled after that of a production system and has two steps:

- select a program (this step is done by an embedded control system).
- run the program (and as a result, change the contents of the blackboard).

Clearly, this computational cycle is designed for a single processor but has an obvious extension for distributed processing, namely:

- 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 years 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 OPS5 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 program to run. 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? We will use human organizations as our models in identifying alternatives. The reasons are three-fold, first, human and software organizations are close enough in structure to share alternatives. Second, human organizational theory is much more mature; considerably greater amounts of thought, effort and experience have gone into its development. And third, we do not wish to reinvent techniques that can be transferred from other disciplines.

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. 2). 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." 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 oth*r than commands and signals.

2.2 Operations

By operations we mean the activities of agents over time. Consider agents A, B, and C from Fig. 2. In general, they can work concurrently, as in Fig. 3. 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 set 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 (i.e., spontaneously). As a result, solutions are found faster than if only preplanned communications are allowed, and also, solutions are found in cases where the methods working independently would fail hopelessly [12].

2.3 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. The 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 far 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. 4.

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 information 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 Hke 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. 4e). 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 thereby, assemble arbitrary organizations 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. Internally, DPSK works with the aid of 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. 2 and 3). 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].

3.2 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,[1]* "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]
[sb2]
[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 transaction and synchronization 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". Aside from the creation of "children" by cloning programs, 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 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)]---}

Copies of S are placed in the available computers and set to working in parallel by a small program whose essential functions are: (1) Identify the unexpanded nodes by retrieving objects that match the pattern: **{Node[Children nil]}**; and (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.

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

Disturbances occur continually in electric power systems and their effects are reported by streams of alarms. A large storm can cause hundreds of alarms to appear in a matter of minutes. A process called "patchwork synthesis" for generating hypotheses to explain the alarms has been described in [3]. Each hypothesis consists of a set of events (disturbances, equipment malfunctions and other errors). Patchwork synthesis uses two crews of programs and a manager to coordinate their efforts (Fig. 5). The first crew selects candidate events with which to expand incomplete hypotheses. The second crew evaluates the candidates and rejects any that make little or no progress towards explaining the given alarms. When implemented in DPSK using three Microvaxen, this system produced diagnoses fast enough to be useful for real time applications in power systems.

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 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 called 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 nonalgoiithmic programs will be to make the software couplings through an optimized version of DPSK, and optimized version of DPSK, and networked workstations to provide for effective concurrent operation.

6 REFERENCES

I. 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, Dept. 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 of the IEEE Power Industry Computer Application Conference. Montreal, Canada, May, 1987.

4. Date, C. J. *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-1*1(1), January, 1981.

7. Galbraith, J. *Designing Complex Organizations* Addison-Wesley Publishing Co., Reading, Mass, 1975.

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

9. Simon, H. A. The Design of Large Computing systems as an Organizational problem. *Organisatiewetenschap en PrakWjk.* H. E. 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. Spector, A. Z., Daniels, D., Duchamp, D., Eppinger, J. L, and Pausch, R. *Distributed Transactions tor 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 tor CAD-Some Algorithmic Issues,* Research Report DRC-18-63-83, Design Research Center, Carnegie Melton 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.

Appendix: DPSK Primitives

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

Transaction Primitives

Any Agent may access the shared database.

- **Begln-Transactlon (class, mode)** Initiates access to a portion of the shared database designated by **class.** The mode can be READ or WRITE. A number of Agents can simultaneously have READ-access to a class in the database, but only one Agent may hold WRiTE-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-Transactk>n(Transaction-ID) Aborts a transaction currently in progress (not commonly used).
- End-Transaction (Transaction-ID) Terminates this database access session.

Command 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 KiLLed as indicated by action.

Synchronization Primitives (events)

An Agent may name many different **events** for synchronization purposes.

- AffIrm-Event **(event)** Affirms (or "raises") an event.
- Check-Event (event) Checks to see if the event is affirmed.
- Walt-Event (event, sec, usec) 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.**

Primitives for Interruptions, exception handling and sending signals

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-Handier (handler)** Designates the routine within this Agent which will be asynchronously called when this Agent is signaled.
- Slg-Group (signal, group) Sends this signal to all Agents in the indicated group.

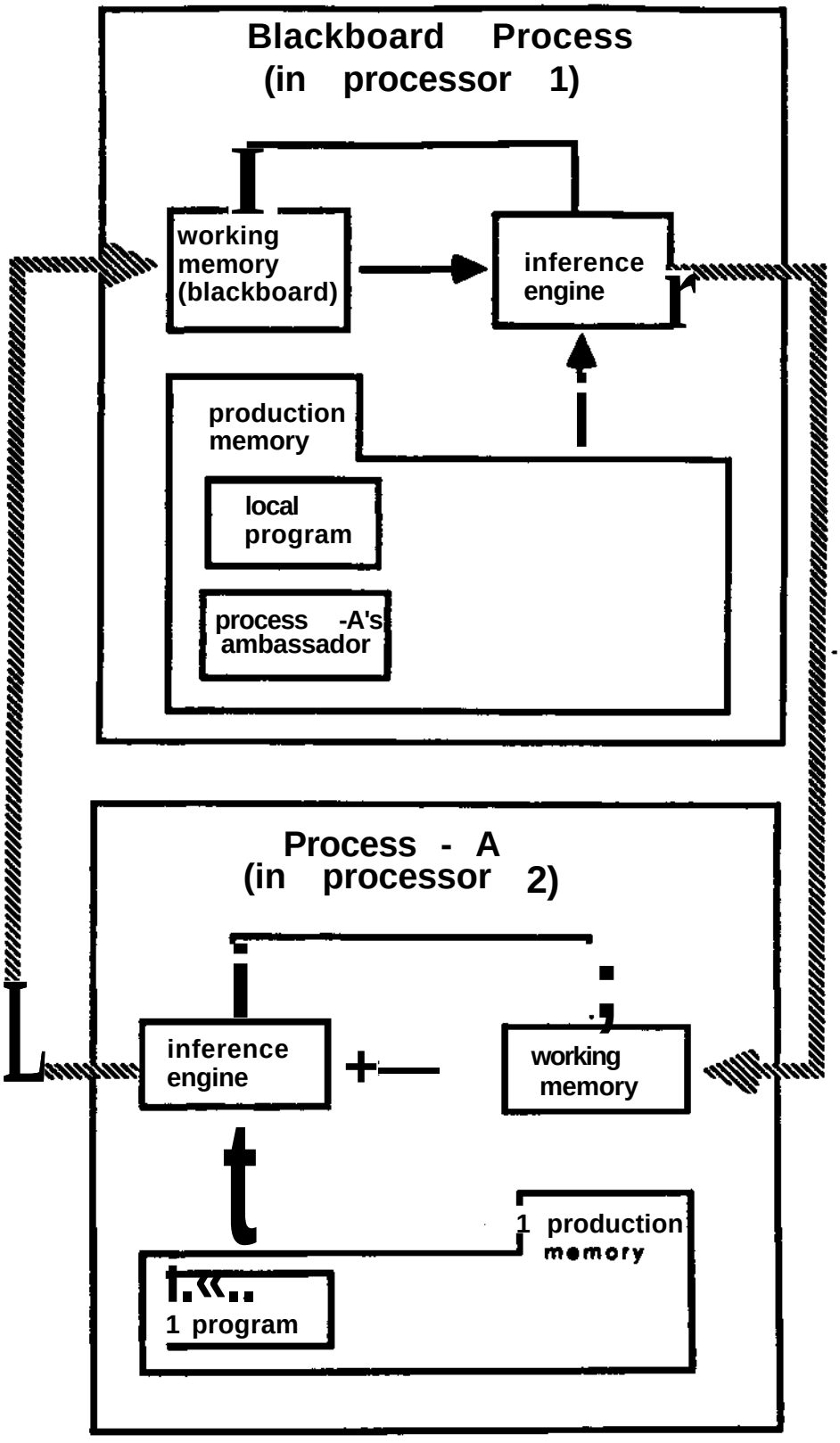
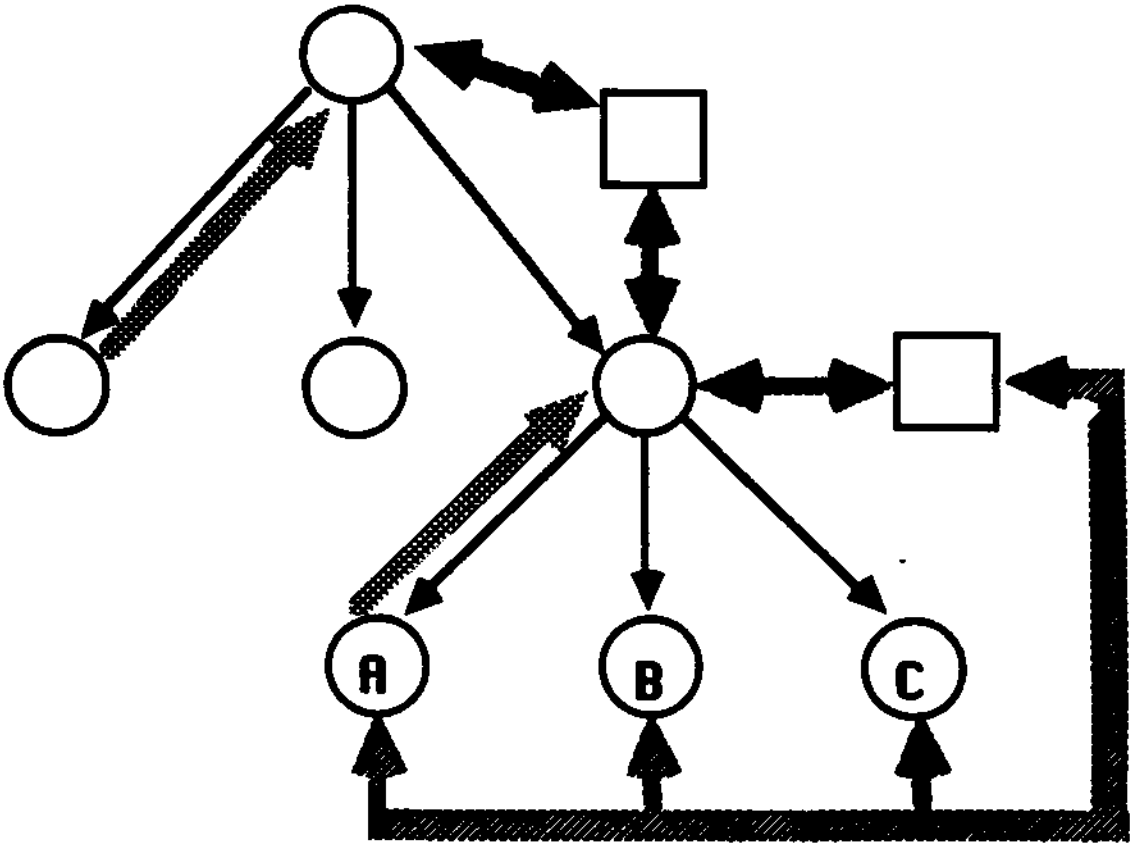


Fig. 1 Ambassadors allow a rule-based process to work as a blackboard. COPS provides the facilities lor processes In remote processors to establish and use ambassadors.



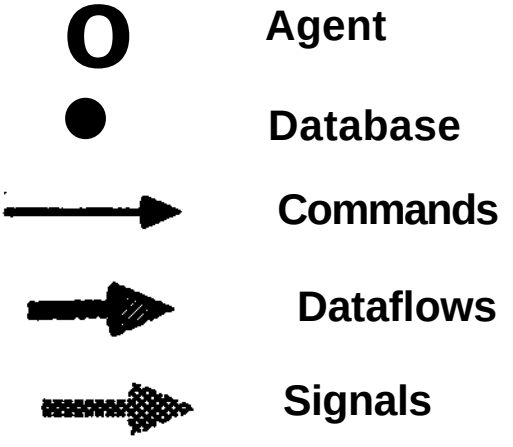


Fig. 2 An organization graph.

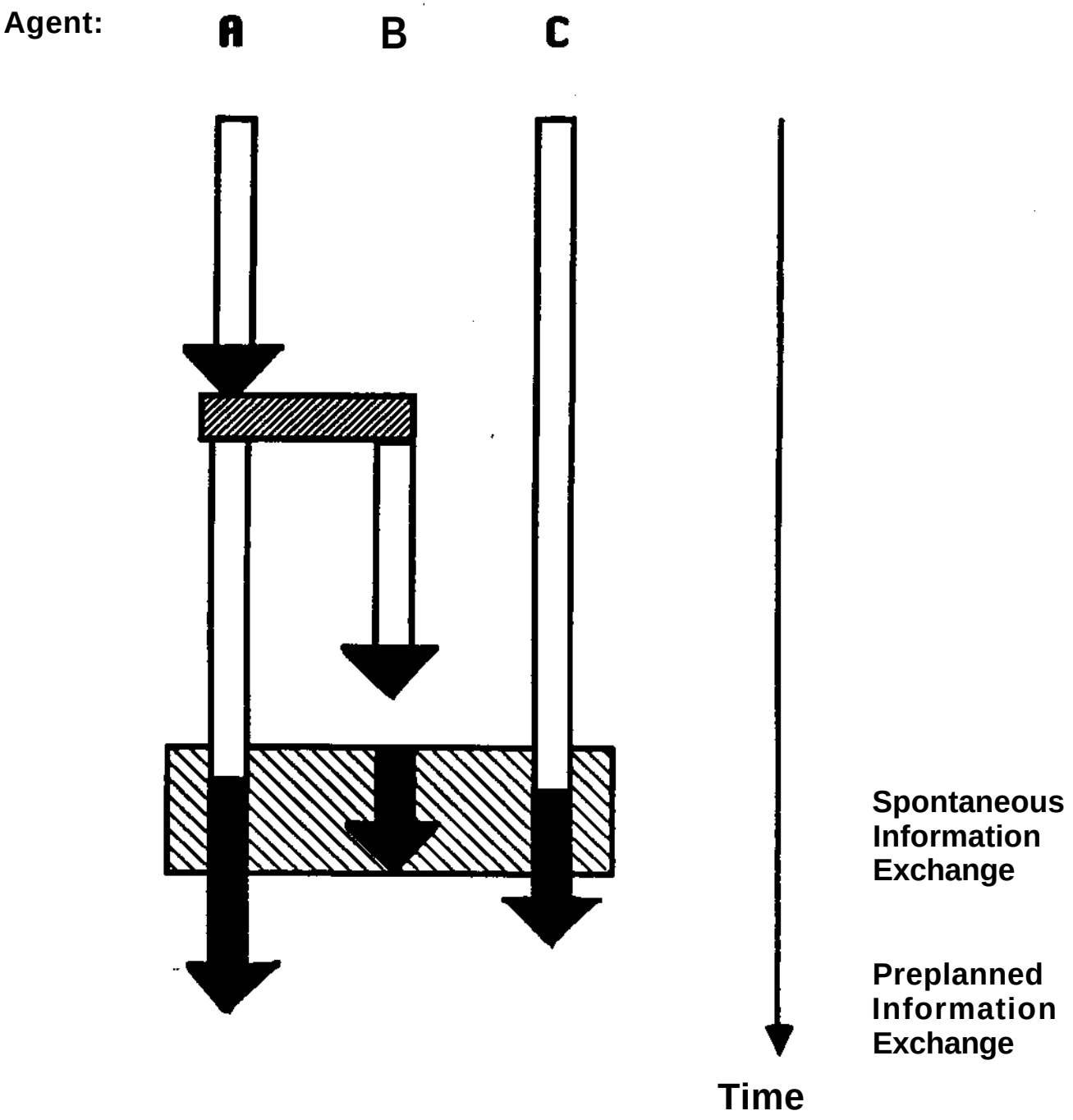
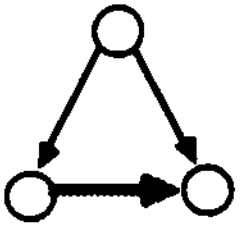
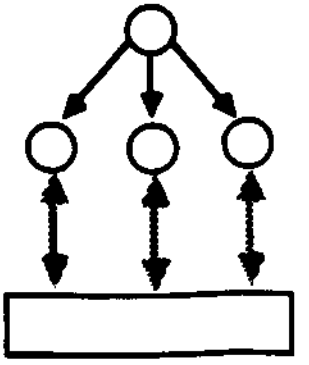


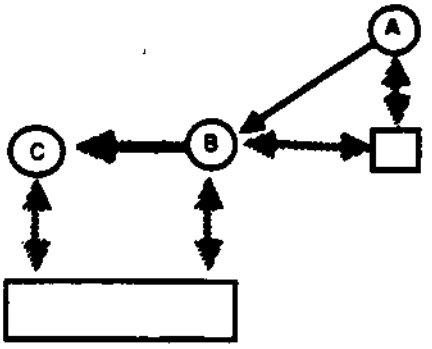
Fig. 3 Two types of collaboration-preplanned and spontaneous. In the latter case, the tasks change as a result of the collaboration (Tasks are denoted by thick arrows.)



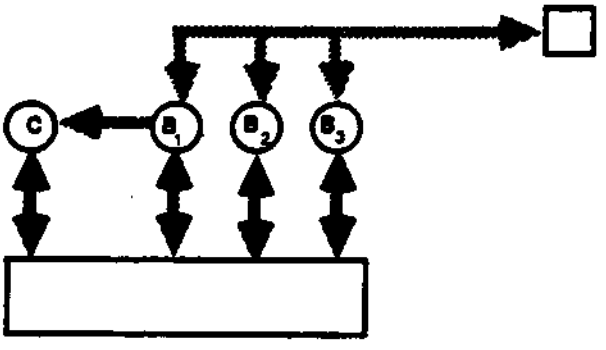
(a) Diract Contact



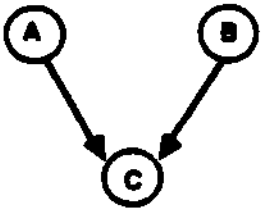
(b) Group



(c) B aarvaa aa A'a rapraaantatlva to C.
Usually, A and B ara in dlfferant proeeaaora.



(d) B_f, B_2 and B_3 conatltuta a task forca to ovaraaa the actlvltlaa off C. B, la tha task forca leadar.



(e) Matrix Management (Dual Authority)

Fig. 4 Some types of lateral relations.

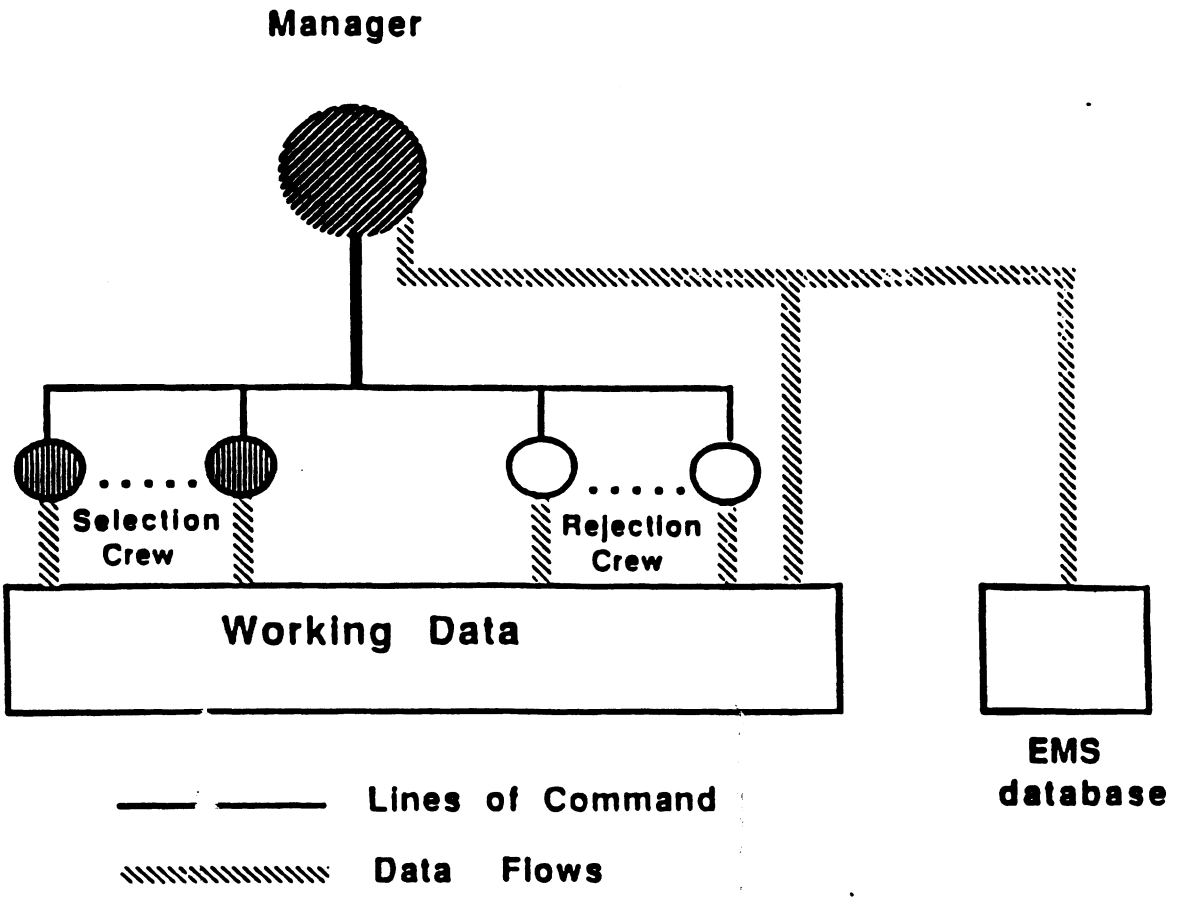


Fig. 5 Organization for patchwork synthesis.