PCL: A PROCESS-ORIENTED JOB CONTROL LANGUAGE

Victor Lesser, Daniel Serrain, Jeff Bonar

Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

A new type of network operating system is required to efficiently implement large, complex process structures on processor networks. This new type of operating system should provide a high-level view of the process structure. It should allow the modular specification of both static and dynamic characteristics of process structures. We propose to provide these facilities by integrating a process-oriented job control language called PCL (Process Control Language) into a network operating system.

We have developed constructs in PCL which allow non-procedural specification of replication, sharing, and re-use of a process structure and its components. This specification also provides information for the scheduler about control flow and resource usage patterns in the process structure, permitting flexible and efficient binding of processes to processors.

1.0 Introduction

Recent advances in hardware technology (i.e. LSI) and communication structures (i.e. packet-switching) have made it possible to construct networks of tens or hundreds of processors. These range from loosely-coupled networks, like the ARPA-Network [Kahn and Crowther, 1971], to closely-coupled networks, such as the UC Irvine LOOP system [Farber, 1975] or the Carnegie-Mellon CM* system [Swan et al, 1977]. Large and complex process structures are required to exploit fully the parallelism and redundancy of these networks. Existing designs for network operating systems do not facilitate the efficient implementation of such process structures.

This research was begun at the Computer Science Department of Carnegie-Mellon University under the support of the Defense Advanced Research Projects Agency (contract no. F44620-73-C-0074) and a research grant from IRIA (Institue de Recherche en Informatique et Automatique) of France. The research at the University of Massachusetts was supported under National Science Foundation grant MCS78-04212.

Daniel Serrain's present address is: Texas Instruments, B.P.5-06270 Villeneve-Loubet, France.

1.1 Inadequacies of Current Network OS

Current network operating systems are designed for small, simple process structures. They do not provide the user with appropriate process specification tools, process debugging and execution monitoring capabilities, or efficent processor-independent resource scheduling for large and complex process structures.

Process specification tools are inadequate. Existing network operating systems (see Forsdick et al [1977], Wulf [1974], and Jones et al [1977] for examples) have procedural process structuring primitives, imbedded in the code of individual processes, that deal only in a limited way with groups of processes and their interconnection structure. Because of this, the process specification code is difficult to understand, hard to write, and lacking in modularity. Furthermore, the specification code is unavailable for inspection by the system. We discovered these inadequacies while implementing two speech understanding systems (Hearsay II [Fennell and Lesser, 1977] and Dragon [Baker, 1975]) on the Hydra operating system [Wulf, 1974] for the C.MMP [Wulf and Bell, 1972] multiprocessor at Carnegie-Mellon University.

Debugging facilities in existing network operating systems are inadequate because they operate with only one process at a time. It is often difficult to locate the process being debugged or to understand its current connections with other processes. Primitives which allow the user to locate processes symbolically are needed. With these primitives a user can monitor the transactions and state transitions of a single process, all processes which are executing the same code, or some arbitrary subset of the process structure.

Current schedulers for network operating systems are inadequate because they assume either an inflexible binding of processes to processors or complete independence among processes. A fixed binding significantly reduces reliability and modularity, particularly in an environment where the number of processors and the size, nature, and number of memories are continually changing. Furthermore, a fixed binding limits the use and efficiency of process structures which dynamically evolve according to interactions with their environment. Only processes which need a specialized processor (e.g., a particular I/O processor) should be tightly bound. In other cases, the system should consider locality

CH1445-6/79/0000-315$00.75 © 1979 IEEE

information and control relationships among processes before performing a (possibly dynamic) binding. The use of this information should help maximize parallelism, minimize communication costs, and minimize operating system overhead.(*)

1.2 A High-Level View of Process Structure

Current network operating systems are inadequate because they lack an appropriate high-level view of a process structure. An appropriate high-level view of the process structure should include both static and dynamic information. For example, it should include the data requirements of individual processes and the frequency of access to these data. It should also make clear how processes are connected, both in terms of direct communication through messages and indirect communication through shared data. Those processes which run in parallel, as co-routines, or sequentially should be discernable. Finally, this view should include the characteristics of the process structure's dynamic response to new data.

More generally, a high-level view should allow a user to view a system as something more complex than a network of producer and consumer processes. We would like to view systems as a "society of interacting processes" whose structure is similar to that of a complex organization. For instance, such organization structuring ideas as Simon's "Nearly-Decomposable Hierarchical System" (1962) should be reflected in high-level specification.

We propose to provide a high-level view of process structure by integrating a process-oriented job control language called PCL (Process Control Language) into a network operating system.

1.3 PCL for Process Structuring

PCL permits a user to specify his process structure, both data and control, to the operating system in a non-procedural manner. This process-based description can be used to guide the operating system in initially configuring the user's process structure, and also in organizing dynamic evolution, scheduling, and process communication. This description is also necessary for the effective development of high-level, process-oriented debugging tools.

PCL provides a mechanism to express an arbitrarily complex process instantiation (activation) pattern in terms of simple operations on a finite and static description. Processes perform simple operations within their own local view of the process structure. When the operations are interpreted by the system in the context of the static PCL description, non-local modification to the process structure may result. For example, one process requests the creation of another process. This involves not only the new process and the connection from the old process, but also the creation of other prespecified connections between the new process and other components currently instantiated within the system. The process initiating creation need not (and for modularity reasons, should not) know about the creation of these other connections. Specification of the process structure is not implicit in the code of the processes themselves, but is explicit in the PCL description of the process structure.

The PCL's facilities are largely orthogonal to the facilities found in sequential languages and operating systems. For this reason, PCL may be implemented either by extending a host language or, as we suggest, by implementing PCL as a job control language where PCL may control execution of user modules, and, in turn, user modules may use functions provided by PCL. A benefit of this approach is that a user's process structure can be implemented with modules coded in a variety of languages. (See Barnett (1975) for another approach to implementing process control in a job control language.)

In the remainder of the paper, we will first describe the components that the PCL uses to specify a process structure. Much of the PCL design makes sense only for a complex and dynamic process structure. To motivate the PCL design, we next present a problem of this type: an air traffic controller implemented on a processor network. With this problem, we will develop a model for large, complex, and dynamic process structures. After presenting an architecture for a PCL based network operating system, we describe the PCL constructs which implement this model.

2.0 The Nature of a Process Structure

Components in a PCL description of a process structure are processes, memory segments, ports, and links. These components are software analogies to the memory, processor, and bus connections which are used to describe the Processor-Memory-Switch structure (Bell and Newell, 1971) of the hardware components. Clusters are groupings of components and are themselves components. Clusters will be explained in detail later.

A process consists of a body of code, state information, and a set of ports. I/O devices are treated as processes which can be bound only to one class of hardware components.

A memory segment consists of an array of memory locations (possibly in a secondary file structure) with an access function for manipulating the information contained in the segment.

(*) For example, the overhead for process context swapping can be minimized if groups of closely interacting processes can be executed simultaneously. In particular we are concerned with the "control working set" phenomenon (Lesser, 1972). Lesser predicts that the execution of a closely coupled process structure on a multiprocessor may result in a significant amount of supervisory overhead caused by a large number of process context switches. These process context swaps occur because there are not enough processors to support the simultaneous execution of the closely coupled processes. It is analogous to the thrashing which occurs when there is not enough physical memory to hold a data working set. A system scheduler can minimize this thrashing if it is aware of clusters of closely interacting processes.

A port represents access capabilities of a process or cluster to other processes, clusters, and memory segments. A port can send and receive data and control information. Ports do not represent the connections between components, but are connected to links, which do represent these connections.

A link is a channel for transferring information from a port to other ports. For links carrying control information or accessing memory, the name of the object implicitly represents a port. Links allow the specification of the communication structure to be independent of the specification of the communicating components. They also allow the components to remain mutually anonymous.

3.0 An Example Process Structure

Consider a network system for automated air traffic control(ATC). At any given time many planes are in the air. Each plane sends information about itself (position, speed, amount of fuel, etc.) in a burst of data. These bursts of information are sent periodically or on demand from the system. The controller process receives each burst of information, updates its global information, and based on that information decides which planes to contact next and what messages to send to those planes.

We will describe the system with the process structure in figure 3.1. CONTROLLER, the master process, takes all bursts received but not processed. With these it updates a global data base, IN_AIR, and after initiating two processes for each plane in the air, goes to sleep. These processes are DISTILL(i) and NEXT_ACTION(i). DISTILL(i) preprocesses information in IN_AIR for the use of NEXT_ACTION(i). NEXT_ACTION(i) determines the message to be sent to plane i and the priority of that message. DISTILL(i) and NEXT_ACTION(i) are coroutines with NEXT_ACTION(i) initially in control. They share PLANE_LOCAL(i), a working memory segment. When NEXT_ACTION(i) is completed it places its results in POSSIBLE_ACTIONS and signals its completion to CONTROLLER. When all NEXT_ACTION(i) are completed, CONTROLLER is awakened, selects which messages in POSSIBLE_ACTIONS are to be sent to the planes, sends them, and reinitiates the cycle. The number of NEXT_ACTION and DISTILL processes vary from cycle to cycle based on how many planes are in the air.(*)

3.1 Static Characteristics of the ATC

First, let us consider the static characteristics of the ATC process structure. We must specify the type, number, and interconnections of structure components. In some cases this information is parameterized. For example, there are n copies of DISTILL, NEXT_ACTION, and PLANE_LOCAL, where n (the number of planes in the air) is determined by CONTROLLER and communicated at run-time.

(*) In order to keep the example simple, we have not specified a process structure that reuses structure based on expectation of a plane remaining in the air from cycle to cycle.

We must describe the nature and the structure of the communications among the components. For example, the data link between CONTROLLER and the memory segment IN_AIR specifies the read-write access capability of CONTROLLER to IN_AIR. The communication link between CONTROLLER and NEXT_ACTION components has a broadcast structure; the same message is sent to the n NEXT_ACTION processes in order to wake them up to work on new data. The structure of this link is different from the one going back from NEXT_ACTION(i) to CONTROLLER. This return link acts as a "concentration semaphore"; a single message is received only when all n are sent.

Finally, component sharing is to be described. For example, multiple links connected to the memory segment IN_AIR indicates that it is shared.

Figure 3.1.1 summarizes these static characteristics. The diagram lacks information about dynamic aspects of the structure, but contains all information necessary to build an arbitrary dynamic structure out of the static components described. One task of the PCL is to capture and localize the static characteristics of a process structure.

3.2 Dynamic Characteristics of the ATC

There are several important dynamic characteristics of the ATC process structure.

There are master processes. In the example, the master process CONTROLLER distributes work in parallel with many NEXT_ACTION processes, which themselves locally distribute work to the DISTILL processes.

Components or groups of components can be copied at will. For example, there are n copies of DISTILL, NEXT_ACTION, and PLANE_LOCAL where n is determined dynamically by CONTROLLER. Each copy has the same internal process structure but with different initial values. Note that even with potential sharing, the link structure must conform to the static specification. For example, L2, the link between CONTROLLER and NEXT_ACTION, should remain a single broadcast link to all NEXT_ACTION processes, no matter how many are created.

Components can be shared. IN_AIR is shared by CONTROLLER and all DISTILL processes. Again, the link structure must conform no matter how components are shared.

Components can be instantiated dynamically. When the process structure is initialized, for example, CONTROLLER and IN_AIR must be fully instantiated. The rest of the structure, however, can exist as simply a static description. (The nature of the static and dynamic descriptions will be explained later.)

PCL must allow a user to represent these dynamic aspects of a process structure, as well as the static aspects previously discussed.

Figures 3.2.1 and 3.2.2 show the PCL description and skeleton code for the ATC. The details of the PCL description will be discussed later.

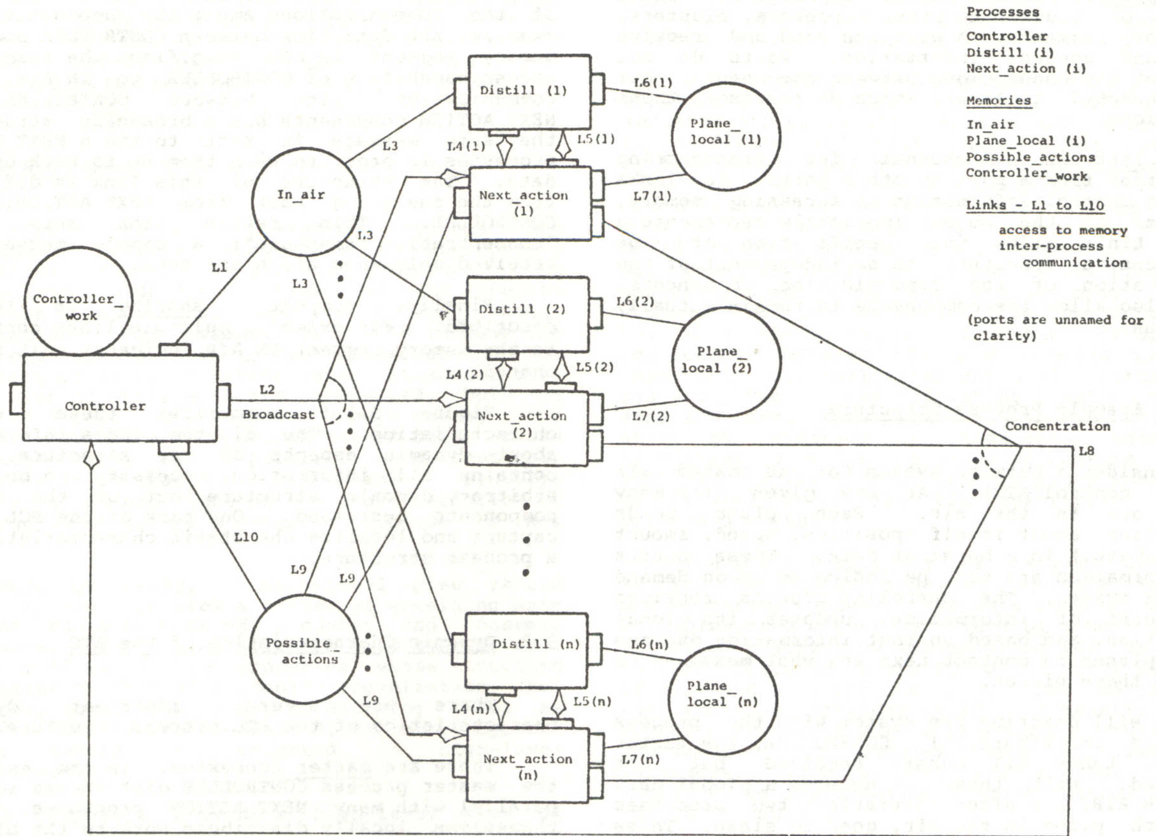


Figure 3.1 - Air Traffic Controller Process Structure

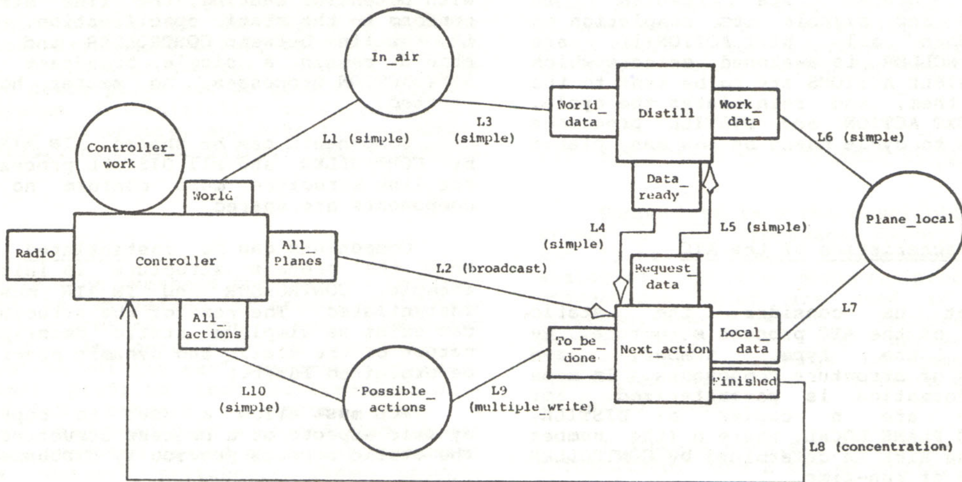


Figure 3.1.1 - Static Summary of Air Traffic Controller Process Structure

Figure 3.2.1 - AIR TRAFFIC CONTROLLER PCL

```
Cluster Air_traffic
   Internal port Radio
   Master Process Controller (copies = 1,
                              phase = execute
                              creation = local )
        external port Radio
        internal port World, All_planes, All actions
        memory Controller_work (size = 1000,
                                mode = readwrite,
                                creation = local,
                                phase = execute )
        end/* Controller */
  Memory  Possible_actions      (size = maxplane,
                                 mode = read,
                                 creation = local,
                                 phase = execute )
   Cluster Plane  (phase = bind, expand with (N),
                  creation = local, copies = N )
        internal port  Finished, To_be_done
        master process  Next_action  (copies = 1,
                                      phase = initialized
                                      creation = local )
             external port  Finished, To_be_done
             internal port  Request_data, Local_data
             end  /* Next_action */
        process  Distill  (copies = 1,
                           phase = initialized
                           creation = local )
        internal port  World_data, Work_data, Data_ready
        end  /* Distill */
    Memory  Plane_local    (copies = 1,
                            phase = bind,
                            creation = local,
                            size = 100,
                            mode = readwrite )
    Memory  In_air   (copies = 1,
                      phase = execute,
                      evolves with Controller,
                      creation = own,
                      size = (N*40),
                      mode = readwrite )
    link  (connections = Next_action. Request_data: Distill,
           type = simple,
           carry = control,
           traffic = high )
    link  (connections = Next_action. Local_data: Plane local,
           type = simple,
           carry = data )
    link  (connections = Distill. Data_ready: Next_action,
           type = simple,
           carry = control,
           traffic = high )
    link  (connections = Distill. Work_data: Plane_local,
           type = simple,
           carry = data )
    link  (connections = Distill. World_data: In_air,
           type = simple,
           carry = data,
           traffic = high )
           end  /* Plane */
    link  (connections = Controller. World: In_air,
           type = simple,
           carry = data,
           traffic = medium )
    link  (connections = Controller. All_planes: Plane,
           type = broadcast,
           carry = control )
    link  (connections = Controller. All_actions: Possible_actions,
           type = simple,
           carry = data )
    link  (connections = Next_action. Finished: Controller,
           type = concentration,
           carry = control )
    link  (connections = Next_action. To_be_done: Possible_actions,
           type = multiple write,
           carry = data )
    end  /* Air_traffic */
```

Figure 3.2.2 - SKELETON CODE FOR THE AIR TRAFFIC CONTROLLER

```
Controller:
     while true do
          begin
          while not IS_EMPTY (Radio)
               do begin
                    RECEIVE (Radio, Controller_work (1),
                             Length_radio_message);
                    Update_all_planes
                    end;
          EXECUTE (All_planes, N = Numplanes_in_air);
          SLEEP
          end

Next_action:
     begin
     Compute_own_situation;
     /* Examine all near-by planes */
     for i := 1 to n
          do begin
               Plane_local (COPY_NUMBER) := i;
               EXECUTE (Request_data);
               SLEEP;
               if Plane_local (Plane_near)
                    then Update_situation
               end;
     Compose_message;
     SEND (To_be_done, Local_data (Final_situation),
          Situation_description_length);
     WAKEUP (Finished);
     BIND
     end

Distill:
     begin
     Get_raw_data (In_air (Plane_local (COPY_NUMBER)));
     Massage_data;
     Construct_situation;
     WAKEUP (Data_ready);
     INITIALIZE
     end
```

4.0 The Nature of Complex Process Structures

From the static connectivity pattern and the dynamic control flow of the example process structure, some observations about decomposing a task into a set of cooperating processes can be made. These observations have motivated a PCL design which allows us to represent both the static and dynamic aspects of a process structure in a modular and non-procedural way.

A complex task can be decomposed into subtasks or clusters. This decomposition is somewhat arbitrary and may be based on shared data within a subtask, functionality of a subtask, interaction intensity, and component life histories.

Clusters are themselves made up of subclusters, in terms of both information and control flow. This permits a hierarchical description of a process structure. The use of hierarchy in specifying a process structure is very useful but should be formulated so that non-hierarchical communication still can be represented conveniently. This method of describing a process structure is compatible with Simon's (1962) view of an organization as a "Nearly Decomposable Hierarchical System".

An important method for obtaining parallelism in a decomposition is by replicating the structure of a subtask and partitioning the data among the instances. A wide variety of different process structures can be constructed by specifying the replication attributes of the components of the subtask. For example, within a substructure, a component may be replicated or may be shared by other components which are replicated themselves. In the ATC we cluster DISTILL, NEXT_ACTION, PLANE_LOCAL, and IN_AIR. Figure 4.1 is an n-copy replication of this structure. N copies are made of DISTILL, NEXT_ACTION, and PLANE_LOCAL. A single copy of IN_AIR, however, is shared by each copy of the other three components in the cluster.

Replication and sharing are specified by a component's <u>creation characteristics</u>. Creation characteristics, which can be parameterized, specify how the component's structure is instantiated. They can be used to specify automatic modification of the process structure upon certain predefined events.

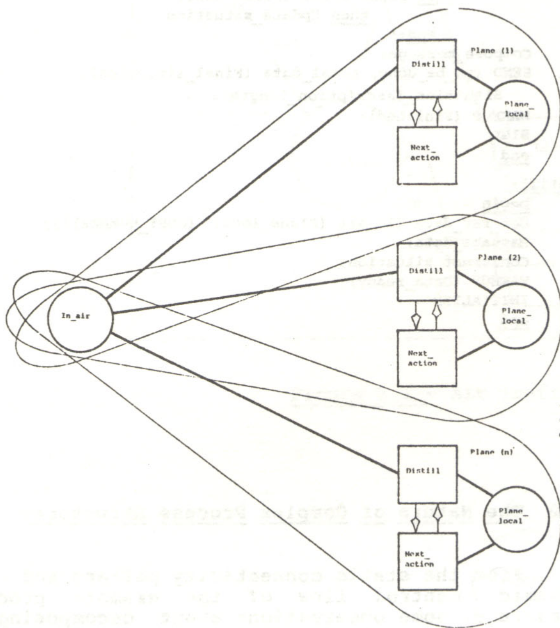


Figure 4.1 - N_copy replication of Distill, Next_Action, Plane_local, In_air cluster

Replication implies the need for synchronization and communication paradigms in order to coordinate and communicate with the multiple component copies. These paradigms include lock-step control, broadcasting of data, and others which will be detailed later. Many of these control paradigms can be implemented directly by communication protocols that deal with groups of processes. For example, the lock-step control paradigm specified in the example can be defined by combining two group communication protocols. One protocol broadcasts the same message to a group of processes, the other protocol waits until all processes in a group have replied before sending a composite message.

The communication paradigms linking components of a process structure can be made independent of the number of replications and the manner of component sharing. A given link is statically specified to connect certain components with a specified communication protocol. When this link is instantiated, it connects all copies of the components in its static specification with the specified communication protocol. For example, a simple link is replicated for each instantiated process so that messages can be directed to specific instantiations. A broadcast link sends a copy of each received message to each receiver, independent of the number of receivers.

Decomposition of a process structure must also address the structure's dynamic behavior. When the process structure associated with a subtask is constructed it can be used repeatedly to perform the same function. Each use might require only minor modifications or re-initialization of the process structure. In the ATC, the NEXT_ACTION processes never change. If this is to be implemented effectively on a computer network, it is important to specify that the NEXT_ACTION processes and their communication links do not have to be completely disassembled every time they complete an analysis.

By describing and manipulating a structure's <u>evolution state</u>, its dynamic behavior can be controlled. An evolution state describes the extent of instantiation of a component. A wide range of dynamic behaviors can be described by pre-specifying how the evolution state of a component is related to the evolution state of its enclosing structure. The description and parameterization of the subcomponent structure simplifies these minor modifications for each repetition and permits the system to automatically modify appropriate parts of the process structure upon certain predefined events.

5.0 Integrating a PCL Into an Operating System

A PCL-based network operating system should describe a process structure on three levels:

1. <u>Process Network Template</u> (PNT) - the static description of the process structure as defined by its PCL description.

2. <u>Dynamic Process Network</u> (DPN) - the current state of the executing process structure as described by the activation records of the processes that are currently instantiated.

3. <u>Process-Processor Binding</u> (PPB) - the current assignment of processes to processors and memory segments to physical memory.

The use of the PCL description is intimately intertwined with the system's execution of a user's process structure. The PNT is initialized by the PCL interpreter based on the user's statically specified process structure. As part of the initialization, the PCL interpreter instantiates parts of the user process structure in the DPN and starts the execution of some of the instantiated processes. During execution, PCL commands generated by executing processes cause

changes to occur to the DPN structure. This is supervised by the PCL interpreter which refers to the PNT and DPN when interpreting a PCL command. The PNT must be used because component activation records in the DPN may describe components not fully instantiated. PCL commands which cause these structures to be fully instantiated must refer back to the component's PNT specification in order to complete the instantiation.

No capabilities, processes, or memory segments can be added to the DPN unless their templates are specified in the PNT. The PPB is created and maintained by the scheduler, which may use both the PNT and the DPN in making its decisions.

The DPN may contain multiple or shared instances of the process structure as long as these instances are created from the elements in the PNT. For example, the interpreter may instantiate a process and memory segment as multiple copies of the process sharing a single copy of the memory segment, a single copy of the process using multiple copies of the memory segment, or even multiple copies of the process using multiple copies of memories segments according to a specified communication discipline. The allowable instantiations are specified in the PCL statically (e.g., "six copies are to be created at run-time") or through parameters (e.g., "n copies are to be created where n is determined at run-time").

6.0 The Design and Basic Concepts of PCL

The PCL allows a user to describe a large, complex process structure with minimal specification. Furthermore, it allows the specification to be non-procedural wherever possible. In this way, PCL provides not only the framework for an actual implementation of a process structure, what Riddle and Wileden (1978) refer to as an "implementation domain description", but also a more abstract "problem domain description".

The PCL allows description of a process structure in terms of static or <u>structural</u> characteristics, <u>evolutionary</u> characteristics, and <u>creation</u> characteristics. The structural characteristics describe individual components. Evolutionary characteristics describe how a component or group of components is represented in the system at any given time(e.g., described but uninstantiated, using system memory, or on system queues and potentially executing). Creation characteristics describe how components are shared and replicated. We will discuss each of these separately.(*)

6.1 Structural Characteristics

Structural characteristics are described in terms of process network components as previously mentioned. In addition to process, memory, port, and link components in the PCL, there are clusters. A cluster is a structure built from the other basic components and other clusters. Clustering of components allows a user to describe a process structure hierarchcally.

Clustering can aid in scheduling by indicating which components will interact often and by supplying specific multi-component resource requirements. The scheduler should attempt to allocate clustered components so that communication within the cluster can be as efficient as possible.

Consider the example in figure 6.1.1. There are five processes named A, B, C, D, and E. A and B communicate extensively, as do C and D. A sometimes communicates with C, D sometimes communicates with B. A communicates with E rarely. A reasonable clustering is indicated in figure 6.1.2. Cluster CLØ contains E and CL1. CL1 in turn contains CL2 and CL3. CL2 contains A and B while CL3 contains C and D. Of course, determining an appropriate clustering would be more difficult in situations with less obviously hierarchical communication relationships.

In the ATC, the cluster PLANE contains the components that communicate closely (see figure 4.1). The cluster indicates that these components should be allocated to hardware elements which can interact efficiently with each other. The cluster also indicates that the scheduler should attempt to allocate and execute them as a group.

Structural characteristics of a component remain constant independent of replication, sharing, or the evolutionary state of associated components. This means that all structural characteristics can be kept in the PNT and need not be copied.

In particular, the specification of the communication protocols and structures, represented by links, remains independent of the dynamic characteristics of the process structure. In order to accomplish this, we have defined a variety of link types which specify different communication protocols. More importantly, the different types <u>statically</u> specify how the communication structures will be modified as the number of linked components <u>dynamically</u> vary.

There are nine kinds of links. (The properties of each link type are summarized in Table 1.) <u>Simple</u> links connect two single components. <u>Broadcast</u> links connect a single sender with many receivers. Every entering message is sent to each receiver. <u>Multiple-write</u> links connect many senders to one receiver. A message from any sender goes to the receiver. <u>Multiple-read</u> links connect one sender to many receivers. A message from the sender goes only to the first receiver who reads it. <u>Concentration</u> links connect many senders to one receiver. The receiver only receives a message when all senders have written a message. The received message is a concatenation of the input messages.

The other four link types are combinations of broadcast, multiple-write, multiple-read, and concentration. <u>MW-MR</u> has the multiple-write function on input and the multiple-read on output. <u>CB</u> links concentrate inputs and broadcast outputs. Similarly for <u>MW-broadcast</u> and <u>concentration-MR</u>.

Component structural information (possibly parameterized) includes the number of copies, specific component attributes (e.g., the size of a memory segment), and initialization rules of a

---

(*) A formal definition of the PCL is contained in a longer version of this paper, published as Technical Report No. 78-12, Computer and Information Science Department, University of Massachusetts, Amherst, Mass.

TYPE	REPRESENTATION	COMMENTS
SIMPLE	A → B	A: sender node B: receiver node - One queue per couple (A, B)
BROADCAST	A → B_1 … B_n	A: sender node B_i: receiver nodes - Each message pushed into the queue is read by all the B_i
MULTIPLE-WRITE	A_1 … A_n → B	A_i: sender nodes B: receiver node - Each message pushed into the queue by any A_i process is received by process B
MULTIPLE-READ	A → B_1 … B_n	A: sender node B_j: receiver nodes - To each message pushed into the queue is associated only one receiver (the first one to read it)
CONCENTRATION	A_1 … A_n → B	A_i: sender node B: receiver node - The message received by node B is the "concentration" of the n messages pushed into the queue (if the link is of semaphore type: a signal is sent. If it is of data type, the message received is the concatenation of the n input messages.).
MW-MR	A_1 … A_n → B_1 … B_m	A_i: sender nodes B_j: receiver nodes - It is the multiple-write function in input and the multiple-read function in output
CB	A_1 … A_n → B_1 … B_m	A_i: sender nodes B_j: receiver nodes - This is the concentration function in input and the broadcast function in output.
MULTIPLE-WRITE-BROADCAST	A_1 … A_n → B_1 … B_m	A_i: sender node B_j: receiver node - This is the multiple-write function in input and the broadcast function in output
CONCENTRATION - MULTIPLE-READ	A_1 … A_n → B_1 … B_m	A_i: sender node B_j: receiver node - This is the concentration function in input and the multiple-read function in output

TABLE I.

Comments:

- the rectangles represent the queues
- the "-" represent access to the queue
- the A_i and B_j are processes

component. These are specified in a list of keyword/value pairs appearing with the component being described. These attributes may be pre-specified or generated during execution. Without execution-time specification of attributes, dynamic process structures would need to be kept in some "fully-extended" form at all times, even though much of the structure might be only rarely used.

Global structural relationships are indicated by nesting (scoping) of component descriptions. The description of a cluster's components, for example, are textually nested within that cluster's description.

The PCL also allows the specification of a component without implying where that component will be used. These are called templates and are analogous to "type" declarations in PASCAL (Jensen and Wirth, 1974). This is both notationally convenient and helpful in abstracting the specification for a given variety of component.

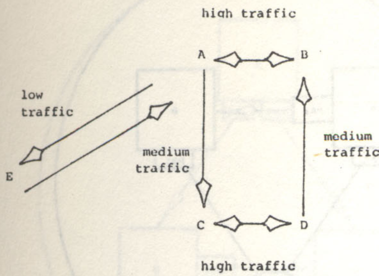


Figure 6.1.1 - Structure with processes having different communication intensity.

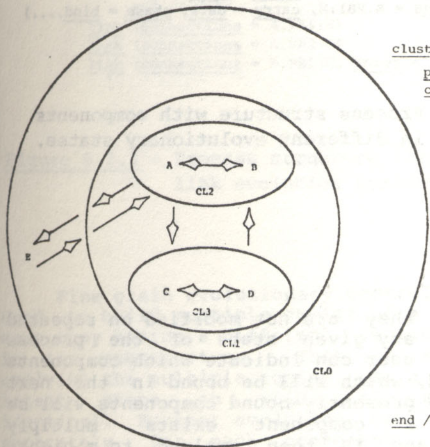


Figure 6.1.2 - Clustering of the process structure in Figure 6.1.1

6.2 Evolutionary Characteristics

Evolutionary characteristics of a process structure describe when and how components are created, executed, and destroyed (deleted from the DPN). A component can be in one of five different states: uninstantiated, bound, expanded, initialized, or executing. A component moves from the earlier states to execution. When execution ends, the component is destroyed or moved to an earlier state. (see figure 6.2.1).

1. An uninstantiated link, memory, or process is known to the system, but exists only as a static description in the PNT.

2. A bound link, memory, or process exists as an activation record in the DPN.

3. In the expanded state, any needed physical memory is allocated. This includes two different kinds of memory. A memory component has physical memory allocated when it is expanded. Similarly, the physical memory necessary to implement the queues associated with links is allocated when the links are expanded.

4. A link in an initialized state is ready to pass messages. Initialized memory might be preloaded with some specified values, but in general is ready to be accessed. An initialized process is ready to execute. Initialization of a component can be done with values from the (static) PCL description or with values contained in the control message which causes the component to move from the expanded state to the initialized state.

5. In the executing state, a process might not actually be running on a physical processor, just as in a conventional multiprogramming operating system a process may be "blocked" or "ready" instead of "running". Memory segments and links which are in the executing state are "used" and must go through the initialization state before they can re-enter the executing state. From the executing state a component may be destroyed, or go back to any of the other states.

A cluster has the same five states. The evolution of a cluster can result in the evolution of all its components up to the cluster's current state. The extent to which a component evolves with its cluster is specified by an evolution attribute of the component. Since a cluster can have other clusters as components, an arbitrarily complex process structure can be evolved whenever a cluster is evolved. This mechanism allows the user to specify non-procedurally which components will exist together at each evolutionary state.

Consider the cluster in figure 6.2.2. It consists of two processes, A and B, and a memory segment M. A is to begin executing, using M. At some point A will start B executing and stop its own execution. B will also use M. When the cluster begins executing, A and M will be executing. This is specified as the default. B, however, could be specified to be in some earlier state. It can be forced into the executing state by a control message passed by A. M remains in the executing state at all times the cluster is executing.

In the ATC, CONTROLLER is the only process in the outer cluster. When execution of the cluster begins, CONTROLLER and IN_AIR are executing and the rest of the cluster PLANE is uninstantiated. CONTROLLER determines the number of PLANE cluster

instantiations and initiates their execution.

The semantics of link evolution are slightly different than for other components. Links stay in their initially specified evolution state until they are needed. When information is sent out a port connected to a non-executing link, the link is automatically evolved up to execution. If either of the components connected by the link goes to some state other then execution, the link goes back to the initially specified evolution state.

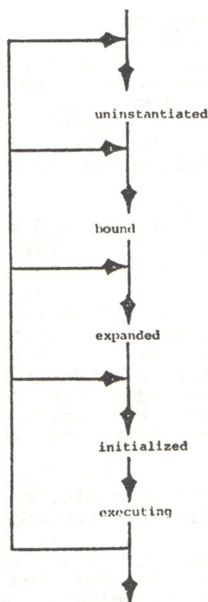
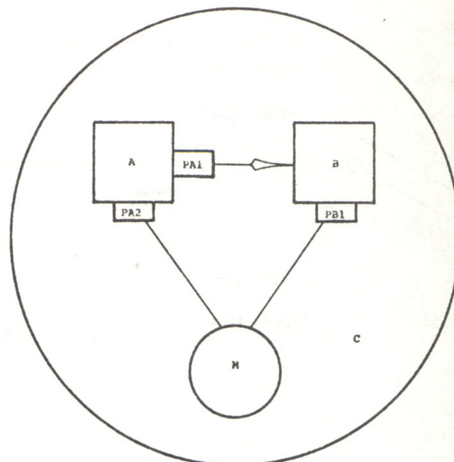


Figure 6.2.1 - Component evolutionary states and state transitions.




```
cluster C (phase = execute,...)
   master process A (phase = execute,...)
      internal port PA1, PA2
         :
      end /* A */
   process B (phase = bind,...)
      internal port PB1
         :
      end /* B */
   memory M (phase = execute,...)
   link (connections = A.PA1:B, carry = control,...)
   link (connections = A.PA2:M, carry = data,...)
   link (connections = B.PB1:M, carry = data, phase = bind,...)
   end /* C */
```

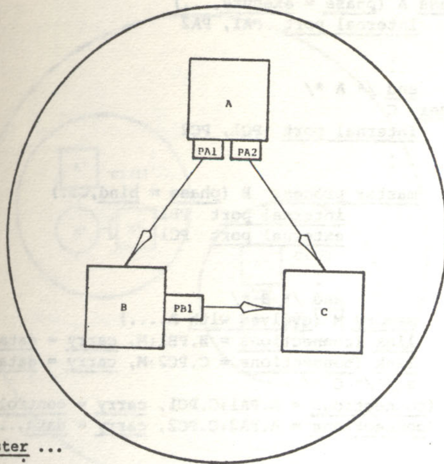
Figure 6.2.2 - Process structure with components in different evolutionary states.

Consider the example in figure 6.2.3. Process A often talks to processes B and C. B occasionally communicates with C. The link between B and C is specified to be bound. If B and C are executing and B sends a message to C, the link between them is evolved to execution. When either B or C go to some non-executing state, the link goes back to being bound. If the link had been specified as executing, it would remain in the executing state even if B and C were not executing.

With the five different states possible for each component, a user has fine grain control of the process structure evolution. This allows memory segment allocation, initialization, and scheduling to be performed at the appropriate time. In particular, the occurrence of a data-dependent or predefined event can cause a component to evolve through its different phases.

Fine grain evolutionary control allows the user to give the system information allowing a more efficient use of the multiprocessor network. Whenever possible, repeated executions can avoid the costly rebuilding of process structure components if they are not modified on repeated execution. For any given state of the process structure the user can indicate which components should be bound, which will be bound in the next state, and how presently-bound components will be reused. If a component exists multiply instantiated, and is then evolved to a higher state with a different number of instantiations, the system will add or destroy copies as needed. This occurs with, for example, the PLANE clusters in the ATC. If the user avoids specification of the fine grain evolutionary structure by putting all components in the executing state, the system will proceed using internal scheduling heuristics.(*)

(*) It is important to remember that we intend the PCL to be a working process-oriented job control language. Because of this, we have integrated specification of evolutionary characteristics and certain link attributes (e.g. "capacity" and "traffic") to minimize operating system overhead and communication costs. When used, these specifications add complexity to a PCL description. Perhaps intelligent scheduling, based on a simpler PCL description, would allow a user to avoid this complexity without sacrificing performance.




```
cluster ...
   :
   master process A (phase = execute,...)
          internal port PA1, PA2
             :
          end /* A */
   process B (phase = execute,...)
          internal port PB1
             :
          end /* B */
   process C (phase = execute,...)
             :
          end /* C */
   link (connections = A.PA1:B)
   link (connections = A.PA2:C)
   link (connections = B.PB1:C, phase = bind)
   :
```

Figure 6.2.3 - Process structure illustrating link evolution semantics.

Fine grain evolutionary control can also be used to build multiple copies of a structure with each copy having a different replication pattern in its substructure. This is accomplished by evolving the multiple copies as a unit to the latest state in which the structures are identical. Further evolution can be controlled separately for each component. Figure 6.3.5 (discussed in detail later) is an example of this.

In the example process structure of figure 6.2.2, fine grain evolutionary control could be important in two different situations. If the cluster was seldom executed and system resources were limited, resource usage could be reduced by keeping B bound (as opposed to executing) as long as possible. On the other hand, if the cluster was executed often, the overhead of continually rebuilding A and B could be eliminated by keeping them expanded or initialized whenever they were not executing.

In the ATC, we wish to avoid continually rebuilding the DISTILL and NEXT_ACTION processes since they will be repeatedly used for each segment of a given input stream. They are to remain initialized whenever the outer cluster is executing.

Sub-components will often evolve in lock-step with their enclosing component. This is the default indicated in the PCL by lexical nesting of component descriptions. Sometimes, however, it is necessary that a component evolve independently of those components with which it is structurally connected. A syntactic escape is provided in the PCL to allow the specification of usage patterns separate from evolution patterns.

As an example, consider figure 6.2.4. The process A begins executing and loads memory segment M. A then starts process B and destroys itself (returns to the uninstantiated state). B uses M heavily for a relatively long period of time. For structural reasons, B and M are in the same cluster. M, however, must evolve with A so it can be loaded initially.

In the ATC, the memory segment IN_AIR belongs structurally to the cluster PLANE because of its heavy use by the DISTILL processes. Because it must be initialized before PLANE is created, its evolution is tied to the outer cluster.

6.3 Creation Characteristics

Creation characteristics specify to what extent the copies are shared by other components in the process structure. A component is either own, meaning it (all of its copies) is shared, local, meaning each time the enclosing structure is replicated the component (all of its copies) is replicated, or dynamic, meaning that replication will be specified at run-time. Since the specification of the number of copies of a component can be done at run-time, there is a subtle interaction between the copies attribute and the creation characteristics. This interaction is best described by a series of examples.

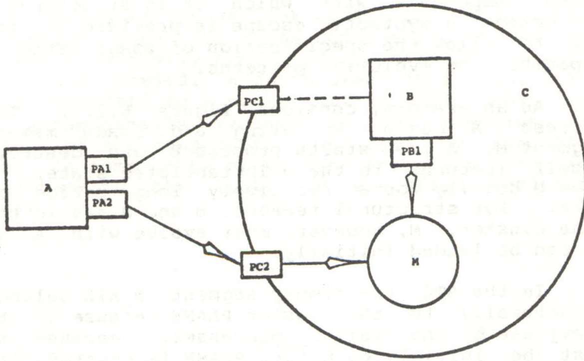
Consider the static process structure description in figure 6.3.1. There are two processes, A and B, both of which have links to memory segment M.

In figure 6.3.2, all subcomponents of C0 are local and have one copy specified. When C0 is evolved with three copies, three copies of C0 and all its subcomponents are created.

In figure 6.3.3, all subcomponents of C0(j) are again local, but this time C1 itself has two copies. When C0 is evolved with two copies, two copies are made of all subcomponents. Thus, two copies of C1 are created for each copy of C0. In all, there are four copies of C1.

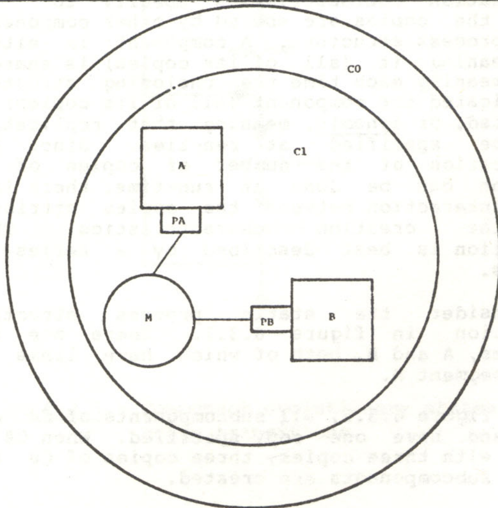
In figure 6.3.4, we have specified 2 copies of C1 and A. We have also specified B to be own. For each copy of C1 there will be two copies of A and one copy of M. Both C1's will share B, however.

Finally, figure 6.3.5 illustrates the use of the dynamic creation attribute. There are three copies of C0, C1 is dynamic, and the other components are left out of this example. Since C1 is dynamic, each of the three copies of C0 could have a different number of C1's. In this case each C1 will be separately evolved with a different number of copies. The number of copies is passed by the process which initiates the evolution.



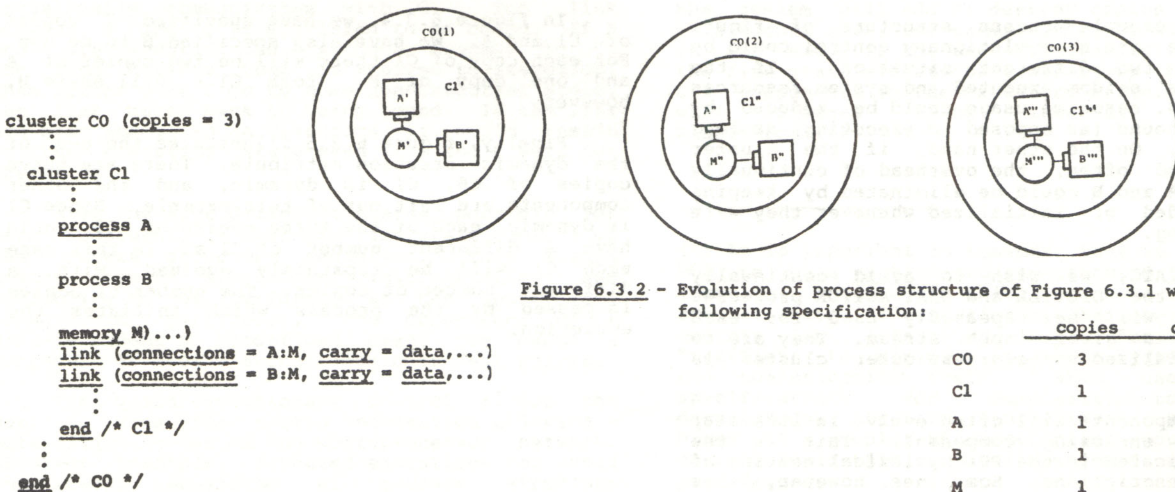

```
process A (phase = execute,...)
        internal port  PA1, PA2
            :
        end /* A */
cluster  C
        internal port  PC1, PC2
            :
        master process  B (phase = bind,...)
                internal port  PB1
                external port  PC1
                    :
                end /* B */
        memory M (evolves with A,...)
        link (connections = B.PB1:M, carry = data,...)
        link (connections = C.PC2:M, carry = data,...)
        end /* C */
link (connections = A.PA1:C.PC1, carry = control,...)
link (connections = A.PA2:C.PC2, carry = data,...)
    :
```

Figure 6.2.4 - M is structurally associated with B because it is heavily used by B.. M is evolutionarily associated with A since it is initially loaded by A .




```
cluster C0
    :
    cluster C1
        :
        process A
          internal port PA
          :
        process B
          internal port PB
          :
        memory M(...)
        link (connections = A:M, carry = data,...)
        link (connections = B:M, carry = data,...)
          :
        end /* C1 */
    :
  end / * C0 */
```

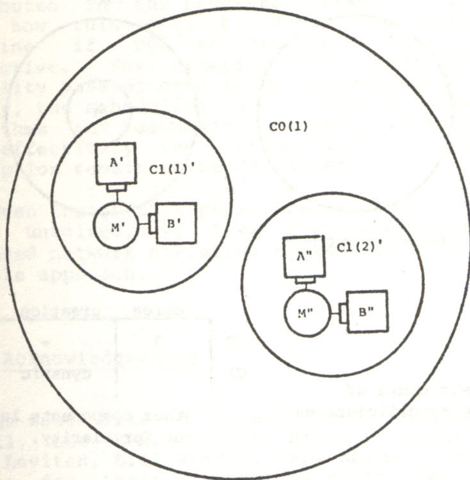
Figure 6.3.1 - Static description of a process structure which will be used to illustrate creation attributes.

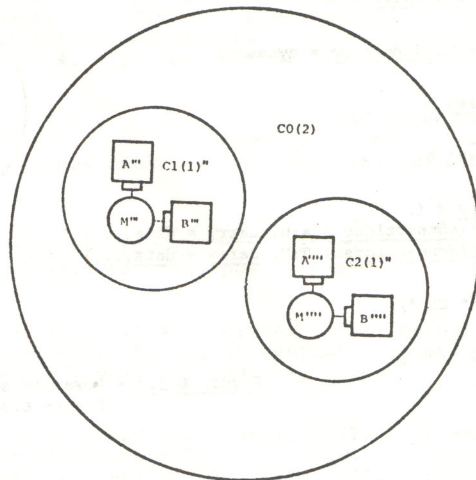



```
cluster C0 (copies = 3)
    :
  cluster C1
      :
      process A
        :
      process B
        :
      memory M)...)
      link (connections = A:M, carry = data,...)
      link (connections = B:M, carry = data,...)
        :
      end /* C1 */
    :
  end /* C0 */
```

Figure 6.3.2 - Evolution of process structure of Figure 6.3.1 with the following specification:

	copies	creation
C0	3	-
C1	1	local
A	1	local
B	1	local
M	1	local

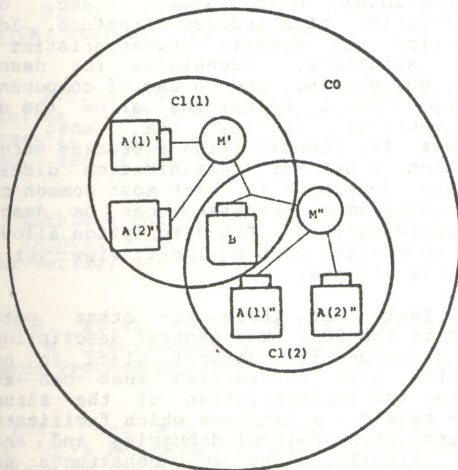





```
cluster C0 (copies = 2)
  :
  cluster C1 (copies = 2)
    :
    process A
    :
    process B
    :
    memory M (...)
    link (connections = A:M, carry = data,...)
    link (connections = B:M, carry = data,...)
    :
    end /* C1 */
  :
end /* C0 */
```

Figure 6.3.3 - Evolution of process structure of Figure 6.3.1 with the following specifications:

	copies	creation
C0	2	-
C1	2	local
A	1	local
B	1	local
M	1	local

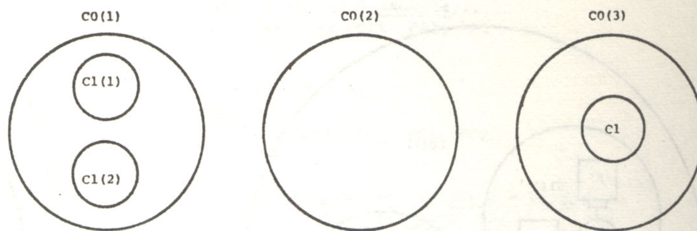



```
Cluster C0
  :
  cluster C1 (copies = 2)
    :
    process A
    :
    process B (creation = own)
    :
    memory M (...)
    link (connections = A:M, carry = data,...)
    link (connections = B:M. carry = data,...)
    :
    end /* C1 */
  :
end /* C0 */
```

Figure 6.3.4 - Evolution of process structure of Figure 6.3.1 with the following specification:

	copies	creation
C0	1	-
C1	2	local
A	2	local
B	1	own
M	1	local

```
cluster C0 (copies = 3)
   :
  cluster C1 (creation = dynamic)
     :
    process A
     :
    process B
     :
    memory M (...)
    link (connections = A:M, carry = data,...)
    link (connections = B:M, carry = data,...)
     :
    end /* C1 */
   :
end /* C0 */
```



	copies	creation
C0	3	-
C1		dynamic

Other components left out for clarity.

Figure 6.3.5 - Possible evolution of process structure of Figure 6.3.1 with the following specificiations:

7.0 Run-time Communication With the PCL

Processes can execute certain PCL commands at run-time. Specifically, a process must be able to access memory segments to which it has links, find its instantiation number when there are multiple copies, change the evolutionary states of components (including itself), send data, and receive data. Examples of the different facilities are in the ATC code presented in figure 3.2.2 .

A memory segment is accessed as if it were an array of words local to a process. If there are multiple copies of a memory segment, the first "subscript" specifies the copy and the second the word in that copy.

The system function COPY_NUMBER can be used to uniquely identify a process which has multiple copies. When created, each copy of the process is assigned a COPY_NUMBER. COPY_NUMBER's are assigned in the order of creation. This is used, for example, by NEXT_ACTION and DISTILL in the ATC.

A process can call system provided routines which generate control messages for interaction with PCL at run-time. A process can change the evolutionary state of another process using the BIND, EXPAND, INITIALIZE, EXECUTE, and DESTROY calls. The SLEEP and WAKEUP calls are used in the standard way to suspend and restart an executing process. All but the SLEEP call have as their first parameter the port through which the control message is sent. Other parameters for BIND, EXPAND, INITIALIZE, and EXECUTE specify values for instantiation parameters of the destination component.

When CONTROLLER moves the PLANE cluster to execution, for example, the number of copies of PLANE must be bound. Control messages can be sent only through ports which have associated control links. The one exception to this rule is that a process may send itself a control message.

Data is sent using the SEND call. SEND has three parameters: the port through which the data is to be sent, the address of the first word to be sent, and a count of the words to be sent. These words must be in the memory segment available to the sender. Data is received by a symmetrical RECEIVE call. Its parameters specify a port in which to receive, a memory cell in which to place the first word, and a count of words to be received.

When something is sent down a link whose queue is full, the sending process is suspended until the queue has room for the new data. Similarly, a process trying to receive from a empty queue is suspended until there are enough data to receive. The routines IS_EMPTY and IS_FULL applied to a port name allow a process to avoid the automatic suspension if that is desired.

8.0 Future Research and Conclusions

By introducing a high-level view of process structure in a network operating system, the PCL simplifies the specification of a complex process structure. The PCL provides a syntactic framework for describing both static and dynamic characteristics of a process structure. In order to describe the dynamic characteristics of a process structure, techniques for describing replication, sharing, and re-use of components are developed. These techniques allow the user to non-procedurally specify the most common techniques for constructing a process structure. In addition, a set of communication disciplines have been developed in which most common control relationships among processes can be described. Most importantly, a PCL description allows each process to operate with a local view yet still have global effects.

We feel that there are other potential benefits to the PCL. The central description of a process structure is a natural place to provide the system with information that can aid in scheduling and implementation of the structure. It also provides a database which facilitates the development of high-level debugging and analysis tools. Finally, the PCL constructs provide information which could substantially reduce dynamic capability checking because the legal communication paths among components is prespecified in a PCL specification.

There are a number of research issues which need to be dealt with if a PCL-based network operating system is to be realized. An effective implementation of PCL will require that the interpreter, PNT, DPN, and PPB be appropriately

distributed in the processor network. It is not clear how this will be done. We must also determine if PCL as described here is too restrictive. For example, does some form of capability passing need to be introduced into PCL? Finally, the scheduling and resource allocation algorithms and debugging and analysis tools that will effectively use information in a PCL description remain to be developed.

Even though this paper leaves many research issues unsolved, we feel that the concept of a PCL-based network operating system is a new and valuable approach.

9.0 Acknowledgements

We would like to thank Jeff Barnett, Dan Corkill, Lee Erman, Peter Hibbard, Anita Jones, Steve Levitan, Bill Riddle, Paul Rovner, and Jack Wileden for their thoughtful criticisms of this paper.

10.0 References

Baker, J.K., 1975. "The Dragon System - An Overview". IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. ASSP-23, pp. 24-29 (February).

Barnett, Jeffrey A., 1975. "Module Linkage and Communication in Large Systems". in Speech Recognition, ed. Raj Reddy. Academic Press.

Bell, Gorden and Allen Newell, 1971. Computer structures: Readings and Examples. McGraw-Hill Computer Science Series.

Farber, D.J., 1975. "A Ring Network". DATAMATION, February, pp. 44-46.

Fennel, R.D. and V.R. Lesser, 1977. "Parallelism in AI problem solving: a case study of Hearsay-II". IEEE Transactions on Computers, C-26, pp. 98-111 (February).

Jensen, Kathleen and Niklaus Wirth, 1974. PASCAL User's Manual and Report, second edition. Springer-Verlag, New York.

Jones, A.K. et al, 1977. "Software management of CM* - A distributed multiprocessor". AFIPS Conference Proceedings, Vol. 46, pp. 657-663.

Kahn, R.E. and W.R. Crowther, 1971. "Flow Control in a Resource-Sharing Computer Network". Proceedings of the ACM/IEEE Second Symposium on Problems in Optimization of Data Communication Systems. pp. 108-116.

Lesser, Victor R., 1972. "Dynamic Control Structures and Their Use in Emulation". Ph.d thesis Stanford University.

Riddle, William E. and Jack C. Wileden, 1978. "Languages for Representing Software Specifications and Designs". Software Engineering Notes, 3, 4. (October, 1978), pp. 7-11.

Simon, H.A., 1962. "The Architecture of Complexity". Proceedings of American Philosophical Society, 106, pp. 467-482.

Swan, R.J. et al, 1977. "CM* - A modular multi-microprocessor". AFIPS Conference Proceedings, Vol. 46, pp. 637-634.

Wulf, William, 1974. "HYDRA: The Kernal of a Multiprocessor Operating System". Communications of the ACM, Vol. 17, No. 6. June.

Wulf, William and C.G. Bell, 1972. "C.MMP - A Multi-Mini-Processor". AFIPS Conference Proceedings, Vol. 41, part II, FJCC, pp. 765-777.