The Overseer
A Powerful Communications Attribute for Debugging and Security in Thin-Wire Connected Control Structures

by
David J. Farber
John R. Pickens

Techinal Report # 75

ICCC - 76

The Overseer
A Powerful Communications Attribute for Debugging and Security in Thin-Wire Connected Control Structures

by
David J. Farber
University of California at Irvine
Information and Computer Science Department
Irvine, CA 92664
Telephone (714) 833-6891
and
John R. Pickens
University of California at Santa Barbara
Dept. of Elec. Engineering and Computer Science
Santa Barbara, CA 93106

Thin wire communications, otherwise known as serial message sending, encourages modularity in distributed program design and makes visible the interprocess communications streams to an unprecedented degree. In this paper, a powerful process monitoring capability, the overseer function, is proposed to aid the program developer in guaranteeing the dynamic correctness of his distributed process mix. The top down design process is overviewed with the emphasis on generating an analyzable model of the intra-module control structure. With appropriate augmentation of interprocess communications streams it is feasible to endow the communications with a control sequence validation capability. The need for dynamic changing process contexts is discussed, and the overseer is shown to be capable of emulating this level of process behavior. Path verification (for protection) and single channel monitoring (for dynamic probing) are two final attributes which may usefully be part of the overseer function. Overall the overseer is only a part of a systematized process for distributed system design, but promises great potential in improving the visibility of dynamic process behavior in distributed systems.

BACKGROUND

Over the past several years , two major trends have caused us to re-examine the organization of processors which exist in the computer field. These trends are:

1. Hardware cost has shown a marked decrease and, therefore, justifies the incorporation of more and more complex functions into hardware. For example, it becomes feasible to place' computation where it is needed; this means distributed computation. It is also more feasible to spend hardware to reduce the cost of performing a total function and/or to enhance the reliability, security, and fail-safety of systems.

2. Modern programming ideas have penetrated the practical programming practice. Top-down, modular, loosely coupled process oriented designs and production oriented user centered program design languages are becoming more commonplace. These design and program production systems have enabled the field to increase greatly the productivity of designers and at the same time increase the reliability and understandability of the resultant systems. In particular, they have allowed us to have a finished design for a system prior to the coding and debugging of the system. In addition they have enabled us to abstract from the design the control and data structures of the resultant system.

This paper introduces a program for the investigation of system design and examines in depth one aspect of this program -- the Overseer. We do not intend to create the impression that the ideas outlined on this paper are easy or that results will be forthcoming overnight. We do intend to illustrate that the necessary theoretical and analytic tools are either now available or can be developed with reasonable additional effort.

THEME

We are attempting to create a total environment in which we may conceptualize, design, program, debug, and then monitor the systems that are to be implemented in distributed, or thin-wire connected, environments. The flavor of this total design environment is best illustrated by outlining a scenario that a system designer would go through in the course of using the approach outlined in this proposal.

When the designer starts his task, he would be embedded in an online program design system similar to that proposed by Caine [Cai75] utilizing a modified Program Design Language (PDL). Additional constructs are necessary to allow the control mechanisms needed by the "thin-wire" interprocess communications used by distributed systems. The distributed design PDL , which we shall call DPDL, is constructed so as to enhance the intercommunication among humans who jointly develop the system design. Toward this goal, the DPDL allows a loose, ambiguous

----
This research has been supported by ARPA under a Network Security grant and by the NSF under the Distributed Computer Systems Project.

semantics but does support and require the statement of an accurate, computer analyzable, control structure. The ambiguous semantics of DPDL encourages the type of open communications between designers that has been shown to pay off in good and easily used design systems but also gives us the complete interprocess level control structure which we need for the analysis of the resulting systems.

At the completion of the design cycle, or more realistically at various critical points in the design process, the resulting design is abstracted into a set of control graphs [Pic75,Pic76], similar in concept, but not in interpretation, to the Graph Model of Computation [Cer72,Gos71] or the Petri net [Hol68,Hol69,Pat70]. We are primarily interested in properties of the control graph that relate to race conditions, deadlocks, and most importantly in the control-recoverability. The latter term is closely related to the intuitive notion of system recoverability familiar to all system designers, the primary difference being that the transitions are limited so as to yield systems in legal states, i.e. the system will not find itself trapped in any of a set of illegal states. Since we have accepted the notion of ambiguous semantics for productivity sake, we do not know whether or not the system being developed is correct in the sense that it will yield correct answers in its executing environment.

Historically in formal analyses of systems the combinatorial complexity precludes non trivial applications. We note however that in the distributed environment that we advocate, that the units of computation are intrinsically modular and hierarchical in nature. Thus with the nature of the systems that are likely to be implemented on the distributed architectures, we expect to be able to successfully analyze the resulting graphs piecemeal and recursively toward higher and higher structures. A node of some graph representing a given level of detailed design can itself be a complex graph, and if that lower level graph has no undesirable traits (such as deadlocks) and is recoverable, then the node on the higher level graph that represents the lower level unit is an acceptable node with respect to the analysis that we intend to perform on it [Pos74].

With the design graph in hand, the designer may now analyze for features that can be verified. If the design is non recoverable then the analysis will also show how to modify the design so as to make it recoverable. With the verified design in hand, the designer is able to "package" the design for efficient performance on a specified set of hardware. An analysis of the program graph allows packaging of the modules of the system to minimize certain cost functions, like communications, while preserving certain performance and reliability levels. The work by Foodym [Foo75] serves as an realistic example of the type of packaging analysis. In certain cases, the designer may even modify the system parameters to see the offect on the resulting system.

With the packaged system at our disposal we can now examine the possibilities on the debugging and run time monitoring of the designed system.

PROPOSED APPROACH

Given the above stated observations on cost/function and program design methodologies, we find it increasingly feasible to propose a machine architecture which executes a modular, distributed message oriented environment. An overview of this processor/programming environment is as follows:

1. We begin with a design for an application system that is to run on our message oriented processor system. A Distributed Program Design Language (DPDL) is used to express all intermodule control and data dependencies, as well as the internal module functions.

2. We next derive from the formal control structure embedded in the DPDL a program graph [Pic76] which describes the possible control flow and synchronization requirements for the application system (APS). The graph nodes represent blocks of code within modules as derived from our top down design, and the arcs represent the paths on which valid inter-module messages may flow. This structure assumes that we have constrained the design in such a manner that all communication between modules on the APS program net is via the explicit flow of messages - i.e. thin-wire communications.

3. From this Program Graph (PG), we next derive a dual graph, the Message Flow Graph (MFG) [Pic76], in which message flow is represented by vertices rather than arcs. The message flow graph (MFG) is more convenient and concise than the Program Graph for use in monitoring control flow.

Given this flow plus the observation that each module in a given program graph may in and of itself be a program graph (properly imbedded and subject to the condition that it is activated by the arrival of messages in the higher level program graph and terminates by sending messages out in that higher level graph) then it is possible to define the complete allowable flow of the application system (APS) as it runs. One of the goals of the computer organization that we propose and the purpose of this paper, is to use this program graph by the communications system to oversee the actual control flow, or equivalently the message flow, of the operating APS.

Given this overseer function, the claim we make is that it is possible to insure a set of desirable attributes about the APS performance, viz:

1. Messages not originated by legal nodes of the APS program net can not enter the program net. This is a statement about the security of the architecture vis-a-vis intrusion from non authorized users (or programs).

2. If the APS arrives at a "hung-up" state, the overseer will take notice and "abort" the APS.

3. If the nodes of the APS program net take too long to function, the overseer will take note and initiate proper error recovery procedures.

4. Inadvertent attempts to move message to nodes that have no explicit flow path are detected and forbidden. This occurrence indicates system malfunction, or probably the implementers implemented a system with a structure different from the one which the design (in DPDL) claimed was to be implemented.

5. Invalid sequential/synchronous behavior will be detected.

THE OVERSEER AS A COMMUNICATIONS ATTRIBUTE

1 Introduction

We now discuss the part of the interprocess communications system which we call the Overseer. It contains the following capabilities with respect to distributed control structures:

o Control-Sequence Verification (CSV),

o Path Verification (PV),

o Single Channel Monitoring (SCM).

2 Overseer Operating Environment

The Overseer exists in a variety of operating environments. Process management, naming conventions, interprocess communications conventions, and communications system structures vary widely from system to system and even sometimes within a particular mix of distributed programs. Nevertheless, in order to discuss overseer issues intelligently in a way relevant to actual implementations, we define the terminology and operating environments around which we frame our remarks.

An important design goal for the Overseer is that it be able to _operate correctly in an environment of vulnerable machines_. The control sequence and path verification techniques should be designed with this environmental constraint in mind. Steps should be taken to make the Overseer invulnerable and failsafe. Process initiation and machine initiation procedures should operate correctly under the assumption that host machines may misconstrue or even falsify requirements and capabilities. In short, "the only reliable component of a distributed programming environment need be the overseer itself [Bai75a,Bai75b].

2.1 Review of Our Program Design Problem

Although the design and development process has been discussed previously we rephrase it here to give a better perspective to the operation of the overseer. The problem is as follows:

Given: A designer has available one or more machines, with zero, one or more processes per machine and zero, one or more pseudo-processes per process (Section 2.2). A standardized communications system, embellished with the overseer function, is to be used for all interprocess communications (thin-wire communications).

_Do_: Design and implement a distributed program composed of one or more processes residing on one or more machines, interconnected via thin-wire communications, and exhibiting an arbitrary degree of concurrency and interprocess synchronization.

In solving this problem it is necessary to design , implement , debug , and validate the control structure .

2.2 The Process Concept

The notion of process evokes several different images and interpretations. For example, in normal usage in describing operating systems, "process" often refers to a body of code, such as the I/O Handler "process". In discussing the operation of the task scheduler, however, "process" may connote a unique context or state space. We favor the latter interpretation.

Multiprogramming refers to the capability of processes to execute in parallel or to be scheduled independently of each other. Transfer of control between processes is normally handled by the operating system and may be synchronous - i.e. processes run until they explicitely give up control- or asynchronous - i.e. processes lose control through rescheduling done in conjunction with interrupts.

In many environments, either the high cost of process contexts [Lau75] or the logical structure of interprocess communications [DCOS74] dictates the creation of pseudo-processes. In such situations we often find one or more modules which appear as normal processes to their operating system(s), but, internally, each module multiplexes one or more pseudo-process contexts by varying pointers to pseudo-context data blocks. Each module handles the scheduling, creation and deletion of pseudo-processes within its domain. For our purposes, we may include pseudo-processes in the notion of process, realizing that there is a one-to-one correspondence between processes and overseer control graphs.

2.3 Program Modules

A distributed program is composed logically of modules. A module has the following attributes:

o Modules correspond to a fixed body of code.

o Modules contain one or more entry points or functions.

o Associated with each module are one or more process or pseudo-process context instantiations (section 2.2).

o Associated with each context instantiation is a unique Program Graph.

2.4 Communications Conventions

As stated earlier, we assume a thin-wire communications framework. In addition, we assume that the communications system may be isolated functionally from the cooperating processes. Two representative communications technologies which we use to illustrate our overseer organization are ARPANET [ARPA] - line switched - and DCS [DCOS74] - message switched. DCS has the additional property that addressing is directly in terms of process name and, since only one path per process pair is allowed, a given pair of communicating pseudo-process conversations must be multiplexed onto a single path.

The overseer's operation is closely tied to that of the communications system. The overseer must have control over message flow in and out of processes. The overseer's implementation can be as distributed as the communications system which it oversees. A centralized overseer may be acceptable in a star- or ring-network, but may overload communications in a distributed network, such as ARPANET.

The overseer may impose other requirements on the communications system, such as maintaining a distributed data base for large control structures. In the interest of brevity, we assume that these and other problems are solvable and, therefore, we concentrate on the logical requirements necessary for the tractability of the overseer functions.

3 Control Sequence Verification

3.1 Definition of Problem

In control sequence verification the overseer validates the control flow and synchronization requirements for each defined process. The verification is limited, by definition of the Program Graph, to that level of control expressible by the message flow. We are not interested in the detailed internal control flow within each process.

Messages which represent invalid control sequences are prevented from delivery to the destination process. Such messages represent control-faults and are handled either by an error code in the delivery status, or by discardment and causing a timeout fault.

3.2 Control Graph Partitioning

Each process in a distributed environment must have a unique and logically separate control graph within the Overseer - note that this does not state how the overseer implements unique instances of graphs, but rather it states what logically must exist. Even in a tightly coupled distributed processes, where each asynchronous process maintains the same view of the total operating control structure, separate control graphs are maintained by the overseer, one per each actual context. The overseer guarantees valid control behavior of each local control graph, thereby guaranteeing valid control behavior of the overall distributed process.

The overseer partitions the total control graph such that the message flow in and out of each logical process is represented by a partial control graph local to that process. If, for example, there are N operating contexts in a single distributed process, then the overall distributed control graph is partitioned into N partial control graphs which have a one-to-one correspondence with the N operating contexts. A point of simplification is that each partial control graph need maintain only as much structure as is necessary to represent the local message flow.

In numerous cases a process, or process mix, is designed independently of the external control environment of which it is to be a part. This is not an unfamiliar phenomenon, as we often encounter such organizations as subroutines or operating system calls in monoprocessor systems, and as server processes [ARPA,Cro72] in network systems. Such organizations are to be expected in modular design methodologies.

Control structures enforced by modular, general, service processes are usually more general than actually allowed by particular control environments. For example, a file handler may allow reads and writes to occur in any order on a given file, but a given caller on the file handler may require reads and writes to alternate.

Now that we have identified both the total control graph and the process of modular design, we may discuss two approaches toward constructing partial control graphs. In the first, the entire control structure of a given distributed program is delineated. In this environment the individual process contexts are tightly bound and, most likely, are designed with detailed knowledge of each others' operations. We have, in this instance, the canonical form of the control graph as it is to be seen by the Overseer. The task of partitioning such a graph is simply to isolate the subsets of arcs and vertices local to each real process context. Graphs partitioned in this manner lend themselves to a structural based addressing scheme (section 3.4). However, this method suffers in environments which change dynamically or which consist of general service modules which may be integrated into many particular control structures.

In the second approach the partial control graphs of general modules are designed independently. The overall control graph for a particular program is partitioned into independently designed partial control graphs. This approach supports modular design techniques, but allows non-canonical forms of program graphs, which in turn disallows structural based addressing schemes (section 3.4).

of every process or pseudo-process within its domain. Given this fact, and the above delineation of possible control graph operations, we are faced with the following questions:

- How visible is the overseer operation to communicating processes?
- How is each graph structure made known to the overseer?
- How is each unique control graph context created and destroyed?
- How is the correspondence made between messages and graph transitions?

Some aspects of these problems depend upon the particular operating system and communications system organization. Specific data structures and the details of module initiation/termination, for example, are outside the scope of this research. Nevertheless, we make several comments on tradeoffs that exist in most environments.

The Overseer appears to incur the most processing overhead if it is entirely invisible to communicating processes. In this kind of implementation the Overseer must not only detect token flow within individual control graphs, but must also create and delete emulated contexts based upon the message flow. Automatic graph restructuring may be required, as when certain control branches are disallowed (see discussion of graph restructuring in section 3.7).

At the other extreme, minimum Overseer processing overhead is incurred when context and structure changes are communicated explicitely to the Overseer. In this implementation the Overseer maintains message filters, which are directly related to partial control graph paths, but are more concisely represented. However, although overseer processing is reduced, CPU processing is increased. Out-of-band communications between overseer and process is now required to update the message filter data structure. The impact of this approach on the programmer may be minimized by incorporating control structure updating into the system programming language, thus making such overhead invisible to the programmer, but a part of the processing nevertheless.

3.4 Token and Arc Identification

Given a Program graph for some distributed process structure, and its transformed Message Flow Graph, our problem is to define the mechanism whereby the overseeer identifies token flow. Here we explore out-of-band message headers - Message Arc ID, or MAIDs - as a possible solution to the problem. As suggested previously, our comments are framed in the ARPA and DCS environments, but may be easily extended into other environments.

There are several design goals for the ideal overseer. First, the overseer should be able to uniquely and unambiguously identify each and every arc over which tokens flow. Second, the overhead incurred in marking messages for recognition should be minimal. Third, the graph addressing scheme should be amenable to dynamic binding, such as is required for a general service module.

Not all of these goals are compatible. There is a tradeoff, for example, between the level of detail visible to the overseer and the resultant extra message overhead.

Four schemes for out-of-band message arc ID fields, MAIDs, are presented below, with comments on their strengths and weaknesses. Section 3.5 discusses the ambiguities which must exist in any communications system based overseer.

Recall that modularly designed Program Graphs may be partitioned according to their implemented process structure. Each partition is composed of an arbitrary control graph structure and has an arbitrary number of message arcs crossing its boundary. The derived MFG has similar partitioning except, that instead of message arcs crossing partition boundaries, message vertices are replicated on partitions with common boundaries (section 3.2).

Given that there is a one-to-one correspondence between process context and graph context, and that message arcs on the PG connect pairs of processes, we conclude that process name is an important component of arc identification. Each arc is identified, partially, by the pair of processes to which it is connected. If we consider the direction of message flow and the participating process names we have, in fact, a very coarse arc identification scheme. This scheme is precise, however, whenever there is only one message and/or response between any pair of processes. Our first attempt is:

$$\text{MAID(1)} ::= < P(i) \dashrightarrow P(j) >,$$

where P(i) and P(j) are process, or pseudo-process, names.

To refine our MAID we recognize that, formally, an arc on the PG is identified by its endpoints. For bound graphs -all arcs connected- we may augment our MAID with structural information. If each vertex in a PG partition is labeled, then we generate the following MAID:

$$\text{MAID(2)} ::= < P(i).M \dashrightarrow P(j).N >,$$

where P(i) and P(j) are the communicating processes, and M and N are vertex labels. Using this scheme we have unique recognition of all message arcs, but we are restricted to completely bound program graphs.

Our next step is to recognize that we may associate a functional name with each arc. Our MAID now becomes:

$$\text{MAID(3)} ::= < P(i) \xrightarrow{F} P(j) >,$$

where F represents the functional identifier. Using this scheme we allow dynamic binding of modular, non-canonical graphs, but still have ambiguity for arcs with the same function name.

In our final attempt we combine the functional and structural MAIDs and generate the following MAID:

$$\text{MAID}(4) \;::=\; < P(i).M \xrightarrow{F} P(j).N >,$$

where M and N are vertex labels, and F is the arc function. This scheme combines the advantages of structural and functional MAIDs by allowing dynamically bound graphs to achieve complete visibility and uniqueness.

For a definable subclass of possible program graphs, each MAID can distinguish all message arcs. Under particular operating conditions or design restrictions it may be satisfactory to adopt a less precise MAID for some or all of the process pairings.

Each of the MAID fields serves a different role in the overseer recognition process. P(i) and P(j) identify the source and destination PG partitions, respectively. As stated previously (section 2.2), P(i) and P(j) may be physical processes or may each be subdivided into pseudo processes. P(i) and P(j) partially identify the ends of a PG arc (MFG vertex).

The function field serves to "unbind" the structures of PG partitions. Whereas, in the structure only MAID, MAID(2), each arc is identified by a member of the set < P(i).m X P(j).n >, in the function only MAID, MAID(3), each arc is identified by a member, possibly duplicate, of the set < F (P(i) X P(j)) >. Without functional identification, communicating processes are tightly bound to a fixed structure.

The structural fields, when added to the functional only MAID, provide clarification by each overseer partition and do not pass through the communications system. In the act of receiving and sending messages, each process notifies its local overseer of any required structural clarification. Structural clarification of the remote process' arc ends is assumed to be done by remote overseer partitions.

With the exception of process names, each of the fields may or may not be required for message-arc identification. Figures 3.4.1-4 give examples of process mixes in which each of the four proposed MAIDs affords satisfactory detail. In Figure 3.4.1 a mix of three processes which exchange at most one message in each direction is described by MAIDs of type 1. In Figure 3.4.2 a pair of processes which exchange more than one message in each direction is described by structural only MAIDs of type 2. In figure 3.4.3 two processes, one of which is a general server process, are described by function only MAIDs of type 3. In Figure 3.4.4 a process mix which contains arcs with duplicate function descriptions is described by MAIDs of type 4. In this last example we demonstrate that structural clarification may be eliminated where ambiguity does not occur (arcs a,b, and c).

3.5 Control Sequence Ambiguities

For each type of MAID presented in the previous section, certain ambiguities in control sequence verification exist. We review the ambiguities here and then comment on basic uncertainties that underlie all MAIDs.

MAID(1) uses process name only. For any pair of processes, multiple paths in the same direction cannot be resolved. MAID(2) uses process name and graph structure. This scheme has no ambiguities as to path recognition, but suffers from the inadequacies of early binding. MAID(3) uses process and function names. Multiple paths may be distinguished, but paths which call upon the same function are not resolvable. MAID(4), which combines process name, function name, and structure, suffers no uncertainty in path recognition.

Underlying all these MAID formulations is the desire to validate vertex initiations and terminations on the program graph. But, because of the delay between a process posting a message and the overseer receiving it, the time that messages arrive in the overseer may not accurately reflect vertex activation times. If the overseer is to have tight influence over vertex activations and terminations, processes must wait for overseer approval before proceeding. The natural problem here, of course, is that potential concurrency is reduced. Thus any implementation will probably choose to allow processes to proceed beyond the points at which messages are injected into the communications system, with the understanding that control faults may occur downstream in the program execution.

3.6 Reentrancy

Other than identifying message arcs the overseer has the task of emulating the changing process contexts. Various conditions govern the creation, deletion, and restructuring of particular instances of program graphs. We discuss in the next sections several dynamic properties of overseer partitioned program graphs. Program graph reentrancy is discussed in more detail in [Pic76].

The desire for reentrancy dictates that reentrant graphs be replicated for each context instantiation. Recursion and reentrancy, from the point of view of control structure and overseer manipulation, are seen as nearly identical problems.

In any executing environment of thin-wire process structures conventions must be established for creating and deleting process contexts. We assume an environment of pre-existent modules, each of which has zero or more active contexts. Each new process-process conversation implies, logically, that a unique process context is created. This mechanism takes different forms in different environments, but the basic operation is the same:

```
P(i)<-[Request_Context ( ARGUMENTS )] ->P(j)
                    :
P(i)<-[ACKNOWLEDGE]                   ->P(j)
                    :
P(i)<-[Arbitrary_Message_Sequence]    ->P(j)
                    :
P(i)<-[Destroy_Context_P(j)]          ->P(j)
```

Context creation/deletion effects the binding and unbinding of local and remote program graph partitions. In the ARPANET the operations are handled by the out-of-band ICP and CLOSE protocols. In DCS, since multiple process-process conversations may be multiplexed on single channels, the context handling primitives are handled by in-band protocols.

An example from the ARPANET is the file handler service module (FTP). Prior to issuing any file commands the ICP protocol (ICP) must be used to effect a unique process context within the service module. When the file transfer operations are complete, matching CLOSE commands are used to sever the communications link.

An example from DCS is the I/O Handler. Whenever the OPENFILE function is issued, a unique process, denoted by the Logical File Name -LFN- is created within the IOH. All subsequent file commands are addressed by the LFN or pseudo-process name. The CLOSEFILE function causes the destruction of the IOH sub-process [DCOS74].

Consideration of reentrancy, the Create/Destroy functions, and the ARPA ICP function leads into an interesting redesign of the ARPA ICP. We now see in modular function handling, as in the case of the DCS I/O Handler, that arguments may be required before a valid context is created within a called module. ICP, as it now stands, creates the context first, and then waits for the arguments. A modified ICP, which conforms better to the dynamics of context management, passes arguments to the called process before the opening of the normal send and receive lines, and the creation of the context, is allowed to proceed.

Pipeline, or GOTO, control structures may also imply control graph reentrancy. In these control structures, slightly different context creation/deletion rules are implied. Consider, for example, a process P(i) which desires to execute a GOTO operation -with parameters- to another process P(j). Once the GOTO is complete, the context representing P(i) is to disappear. This operation is represented as follows:

```
P(i)<-[Request_Context ( ARGUMENTS )] ->P(j)
                    :
P(i)<-[ACKNOWLEDGE]                   --P(j)
                    :
P(i)<-[Arbitrary_Message_Sequence]    ->P(j)
                    :
P(i)<-[Destroy_Context_P(i)]          ->P(j)
```

The difference between this and the previous control transfer sequence is that the caller's, rather than the callee's, context is deleted once the conversation is complete. Current examples of this type of control transfer do not exist on the ARPANET or on DCS, but may be used in implementing parallel pipeline operations.

The overseer must have knowledge of whatever mechanism is used for binding local and remote program graph partitions and contexts. Many implementations are possible, depending on the operating system and communications system structure, but context creation and deletion must be integrated carefully into the overseer operation.

3.7 Graph Restructuring

Reentrancy and queuing imply a form of dynamic graph restructuring in which all or portions of a graph are replicated. However, a more general graph restructuring may be desired and may be able to remove certain dynamic validation functions from the processes themselves. Most of the applications of graph restructuring seem to stem from the need to restrict availability of functions within a module. Graph restructuring may be thought of as a form of capability based addressing, where caller's capabilities are determined by the callee [Wei73].

Two examples within a file handler module are, 1) a file opened for read-only access must not allow any writes, and 2) a sequential file must not allow any random access requests. These access restrictions may be most conveniently handled by enabling and disabling paths on the local overseer's graph partition. More complex restructuring primitives are certainly possible, but don't seem to be required for the control constructs presented in this paper.

4 Path Verification

Path verification adds another dimension to the capability of the communications system overseer. Defined simply, it is the ability of the overseer to approve, to deny, or to revoke communications paths. The criteria which governs path creation is determined by the particular protection scheme in use. Although this research does not delve into protection, it is helpful to examine the capability of the overseer with respect to a representative protection scheme.

Path verification is useful both in protecting a distributed process from its own misbehavior and in isolating non-cooperating processes from each other. The latter is also known as encapsulation, and may be applied either at the process level or at the machine level [Bai75b,Bis73].

The problems of protection in message based systems have been broached elsewhere [Wei73,Bai75,Zel75], but, with the exception of Bailey, the potential role of the communications system has not been considered. As a result, the existing techniques which ignore the communications component are weak within environments of vulnerable host machines. Protection/path verification belongs in the overseer, and offers a significantly improved level of correctness.

The primitive operations usually associated with communications path manipulation are as follows:

o Allocate a port.

o DeAllocate a port.

o Send a message to a port.

o Receive a message from a port.

The dynamics of port allocation and deallocation are very closely tied to the dynamics of process creation and deletion (section 3.6) - when a new process is invoked, a new pair of ports is also created. Thus it is only natural to apply other concepts normally affiliated with processes and process hierarchies.

One effort [Zel75] views ports as capabilities and affiliates with each process a capability vector. Additional attributes associated with each port/capability include:

o Ownership Privileges

o Protection from modification by other processes

o Ability to be passed to other processes

The overseer is the natural agent to effect these port or capability based attributes.

Port passing is especially significant to the overseer in terms of how it effects the binding of names in program graphs. A desired feature of any system of distributed processes is to be able to pass a port name through several levels of reentrant procedure calls. The overseer should allow this deferred binding and also be able to detect invalid bindings in the lowest level of the nested call sequence. To do this, capability based conventions must exist for passing ports between processes. We are not concerned whether the convention be caller inherits callee rights or visa versa, or what the exact nature of the rights associated with ownership and non-ownership of capabilities. We are concerned that port names may be passed between processes, and that the overseer knowingly participates in the passing of port names.

A simple example illustrates the utility of path verification and its relation to control sequence verification. Figure 4.1 shows a system of three nested processes, where the first process supplies the second process with the name of the third process. To accomplish the control structure addressing a MAID of type 1 - process name only - is used. Arguments are passed by arcs a and c, and responses are passed by arcs b and d. Three columns are used to represent the message arc ID, information passed along the arc, and context action implied by the passage of the message on each arc. The overseer must have these three items of information for each partial control graph arc. Note that the message which passes along arc a contains not only arguments for the called subroutine but also the capability for process P(2) to establish a path to P(3). Arc c's representation indicates that the port defining the P(3) connection is deferred. When P(2) places a message on arc c the overseer must determine that the capability passed from P(1) to P(2) justifies the creation of the P(2)-->P(3) path.

5 Single Channel Monitoring

A valuable attribute of the overseer is the ability to allow dynamic monitoring of interprocess communications. Lacking this facility, artificial maneuvers are required, such as recompiling the affected modules in order to address message deliveries to a special monitoring module. The SCM feature of the overseer simplifies the task of monitoring the actual data streams and, in addition, supports a fine grain of selectivity on messages to be rerouted to the monitor module. The implication of this last feature is that the overhead of message rerouting need be only minimal.

In order for Single Channel Monitoring to be effective, two features are required:

1. The Overseer must have the ability to accept message filters, i.e. data structures which identify messages of interest.

2. A special process must exist, responsive to a human user, and capable of analyzing and synthesizing communications streams.

In the following sections we examine each of these requirements. Prior to our discussion, however, we note that the existence of SCM modules and Overseer filters potentially increases the vulnerability of systems.

Given the capability to intercept and alter data streams between processes, SCM modules have the potential for wreaking havoc in a mix of distributed processes. In defense of using SCM modules it should be noted that this problem is shared with more classical debugging tools, in that the setting of breakpoints and the ability to alter both instructions and data may wreak havoc on a single processor system. The solution for single processor systems, which uses addressing protection schemes to limit a developer's interference to his own name space, has an analogue in message systems through the application of capability based addressing or path verification by the overseer.

5.1 Properties of Message Filters

Communicating processes should ideally never need to know, other than by degraded performance, of the insertion of special monitoring processes. Monitoring processes should be able to be switched in dynamically, with no internal renaming required on the part of the monitored processes. If the communications system has the capability to reroute messages, based upon message filters, and subject to the addressing restrictions enforced by path verification, then the dynamic readdressing required to switch monitoring modules in and out may be conveniently achieved.

At first glance, ignoring the overseer's capability to monitor control structures, one might design filters to simply capture the entire message flow on channels (e.g. an existent line in line switched communications systems, or all messages between a given pair of processes in message switched systems). While the ability to capture total conversations is important, the ability to factor out messages using a finer grain of selectivity is paramount. A filter should not only be able to capture messages between a given pair of processes on a given channel, but must also allow capture of single messages which represent flow of primitive control tokens.

Thus, to identify messages of interest, filters must have several attributes. First, filter selectivity must range from coarse - e.g. all messages on a given channel - to fine - e.g. the message representing the traversal of a specific control arc -. Message selectivity may be based on several classifying factors. Process name, subchannel arc, function, and graph structure are identifying entities which, taken individually or in logical combination, contribute to the delineation of classes of messages of interest. In its most sophisticated form, an overseer implementation may allow set operations, such as intersection or union on the classes of messages identified by the primitive classification operations.

Another attribute of filters is that they must reflect the dynamic properties of control structures. The messages which bind and unbind contexts to control graphs should be describable by the filter descriptors. This facility may be effected either by chaining filters - e.g. a filter to detect the context transition followed by another to identify messages of interest in that context - or by special functions such as "ENVIRONMENT OF" for nested calls.

Each filter should have, as another of its attributes, a name or Filter IDentifier (FID) by which its actions may be referenced. If, for example, a monitoring process has multiple filters outstanding, then captured messages may be correlated to the particular filter which captured them by including a named reference to the filter. Any given monitoring process can correlate the actual outstanding named filters with the code which processes each of them by referral to the FID.

In more sophisticated systems, the overseer may accept the definition of arbitrarily large sets of potentially active message filters. In such an environment, message filters would first be defined - or declared - by the monitoring process to the overseer, and then be activated/deactivated as needed through low overhead enable/disable commands.

Given the ability of the overseer to store multiple filters, another feature is possible which allows chaining of filters. Chaining is the technique which allows the tracing of context sensitive information flow. If, for example, all RETURNs from a module appear alike, and it is desired to capture only the RETURN from a given sub function call, then a chain of two filters - one to detect the CALL, followed by one to detect the RETURN - allows the correct selectivity. This operation is effected through the NEXT_FILTER field, which describes the next filter to be invoked given a successful message match.

An additional filter feature required for tracing context sensitive information flow is back referencing. When, for example, a filter is dependent upon context information defined in a previous filter, then some mechanism is required to pass that context information onward. Back referencing is one way in which information detected by current filters may be passed on to future filters.

The final attribute which message filters require is a specification of the action to be performed on detected messages. The most useful actions appear to be the following:

- o NULL -- Do Nothing at all. Most likely to appear when using the chained filters option.
- o LOG -- Send a summary message to the monitoring module. Data to be summarized include a copy of the MAID header and length fields.
- o COPY -- Send an exact copy of the message to the monitoring module, but allow the original message to proceed untouched.
- o CAPTURE -- Reroute the complete message to the monitoring module.

ACKNOWLEDGMENTS

The research proposed in the above document is based on developments during the past two years. Some of the basic contributions to the practical use of the theoretical methods have been already acknowledged by references in the paper. Additional credits should go to Jon Postel for his thesis research in the use of Graph Models for protocol analysis and to Philip Merlin for his pioneering work on the recoverability model .

REFERENCES

ARPA . ___, "ARPA Network Current Network Protocols", ARPA Network Information Center #7104, Stanford Research Institute, Menlo Park, CA (NIC).

Bai75a Bailey, D.J. "Network Structure and Security", Proceedings of ACM IPC workshop, (March 23-24, 1975).

Bai75b Bailey,D.J. "Central Computing Facility Planning Study: Technical Overview," LA-5752, Los Alamos Scientific Laboratory, Los Alamos, New Mexico, (March 1975).

Bis73 Bisbey,R.L., G.J. Popek "Encapsulation: An Approach to Operating System Security," ISI/RR-73-17, University of Southern California Information Sciences Institute, (October 1973).

Cai75 Caine,S.H., E.K. Gordon "PDL - A Tool for Software Design," Proc. National Computer Conference, (May 1975).

Cer72 Cerf,V.G. Multiprocessors, Semaphores, and a Graph Model of Computation, Ph.D. Dissertation, ENG-7223, Computer Science Dept., U.C. Los Angeles, (April 1972).

Cro72 Crocker,S.D., J.F. Heafner, R.M. Metcalfe, J.B. Postel "Function-Oriented Protocols for the ARPA Computer Network," AFIPS Conference Proceedings, Volume 40, (May 1972).

DCOS74 Rowe,L.A., E.J. Earl, A.D. Foodym, F.R. Heinrich "Distributed Computer Operating System - Programmer Guide," U.C. Irvine Distributed Computer Project, Technical Report, (April 1974).

Foo75 Foodym,A.D. Ph.D. Thesis in preparation at U.C. Irvine.

FTP ___, "File Transfer Protocol," ARPA NIC #17759.

Gor73 Gord,E.P., M.D. Hopwood "Nonhierarchical Process Structure in a Decentralized Computing Environment," Technical Report 32, U.C. Irvine, Dept of ICS, (June 1973).

Gos71 Gostelow,K.P. Flow of Control, Resource Allocation, and the Proper Termination of Programs. Ph.D. Dissertation, ENG-7179,Computer Science Dept., U.C. Los Angeles, (December 1971).

Hol68 Holt,A.W., H. Saint, R.M. Shapiro, and S. Warshall "Final Report for the Information System Theory Project," Rome Air Development Center, Applied Data Research, Inc., New York, contract AF 30(602)-4211, (1968).

Hol69 Holt,A.W. and F. Commoner "Events and Conditions," (Parts 1-3), Applied Data Research, Inc., New York, (1969).

ICP Postel, J.B. "Official Initial Connection Protocol,". ARPA NIC #7101, (June 1971).

Lau75 Lausen,S. "A Large Semaphore Based Operating System," CACM 18, 17, (July 1975).

Pat70 Patil,S. Co-ordination of Asynchronous Events, Ph.D. Dissertation, MAC-TR-72, MIT, Cambridge, Mass., (1970).

Pic75 Pickens,J.R. "A Study on Program Graphs and Their Generated Message Flow," Tech. Report #65, Dept. of Info. and Computer Science, U.C. Irvine, (May 1975).

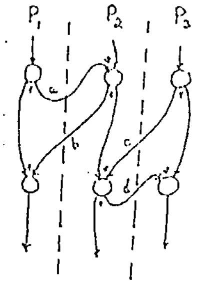
Pic76 Pickens,J.R. Debugging and Monitoring of Distributed Control Structures, Ph.D. Dissertation, Dept. of Electrical Engineering and Computer Science, U.C. Santa Barbara, Santa Barbara, CA, (Early 1976).

Pos74 Postel,J.B. A Graph Model Analysis of Computer Communications Protocols, Ph.D. Dissertation, ENG-7410, Computer Science Dept., U.C. Los Angeles, (January 1974).

Wei73 Weinstock,C.B. "A Survey of Protection Systems," Computer Science Dept., Carnegie-Mellon University, (July 1973).

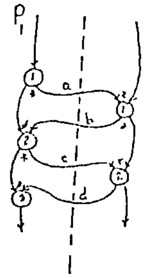
Zel75 Zelkowitz,M.V. "A Proposal on Process Hierarchy and Network Communication," Proc. of ACM IPC Workshop, (March 23-24, 1975).

P. 1 of ?



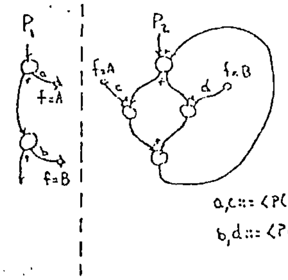
$a ::= \langle P(1) \dashrightarrow P(2) \rangle$
$b ::= \langle P(2) \dashrightarrow P(1) \rangle$
$c ::= \langle P(3) \dashrightarrow P(2) \rangle$
$d ::= \langle P(2) \dashrightarrow P(3) \rangle$

Figure 3.4.1 Structure Requiring Type 1 MAID



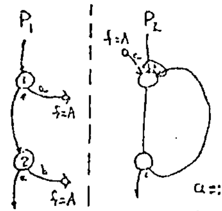
$a ::= \langle P(1).1 \dashrightarrow P(2).1 \rangle$
$b ::= \langle P(2).1 \dashrightarrow P(1).2 \rangle$
$c ::= \langle P(1).2 \dashrightarrow P(2).2 \rangle$
$d ::= \langle P(2).2 \dashrightarrow P(1).3 \rangle$

Figure 3.4.2 Structure Requiring Type 2 MAID



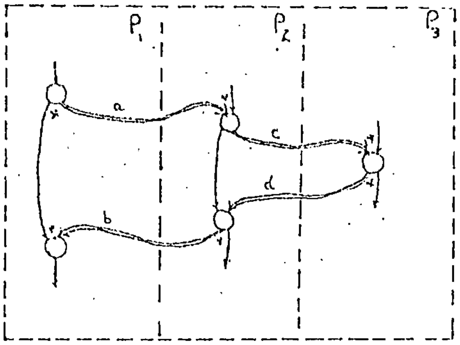
$a,c ::= \langle P(1) \xrightarrow{A} P(2) \rangle$
$b,d ::= \langle P(1) \xrightarrow{B} P(2) \rangle$

Figure 3.4.3 Structure Requiring Type 3 MAID



$a ::= \langle P(1).1 \xrightarrow{A} P(2) \rangle$
$b ::= \langle P(1).2 \xrightarrow{A} P(2) \rangle$
$c ::= \langle P(1) \xrightarrow{A} P(2) \rangle$

Figure 3.4.4 Structure Requiring Type 4 MAID



arc	MAID	data	Context Action
a	$\langle P_1 \dashrightarrow P_2 \rangle$	[Port(P_3), Args(P_1, P_2)]	Create P_2
b	$\langle P_2 \dashrightarrow P_1 \rangle$	[Response(P_2, P_1)]	Delete P_2
c	$\langle P_2 \dashrightarrow * \rangle$	[Args(P_2, P_3)]	Create P_3
d	$\langle * \dashrightarrow P_2 \rangle$	[Response(P_3, P_2)]	Delete P_3

Figure 4.1 Path Verification, Nested Call

David Farber is an Associate Professor of Information and Computer Sciences at the University of California at Irvine. His career has included research and development efforts in Programming languages, operating systems, and computer and communications systems architecture. He has held technical and managerial positions at Bell Telephone Laboratories (1956-1967), the RAND Corp (1967-1968) and the Xerox Corp. (1968-1970). He is the Principal Investigator of the NSF Distributed Computer Systems Project, the ARPA Network Security and the Local Communications Project at UCI. He has been an ACM National Lecturer and an IEEE Distinguished Visitor. He is a Member of Sigma Xi. Currently Prof Farber is a member of the National Science Foundation Computer Science and Engineering Advisory Board and a member of the Hewlett-Packard Science Advisory Board.

John Pickens graduated from the University of California at Santa Barbara with BSc, MSc and PHd degrees in Electrical Engineering and Computer Science in 1968, 1974, and 1976 respectively. From 1968 to 1971 he was a systems programmer designing interactive signal processing systems with CHI Inc. in Santa Barbara. From 1971 to 1976 he was actively involved with the ARPA Computer Network, designing network graphics software and microprocessor based terminals and evaluating user interface issues. He performed this communication systems research from 1974 to 1975 as part of the ARPA Network Security Project at the University of California at Irvine. John is currently with RNR INC. in California, and is responsible for software development in voice and data communications systems.