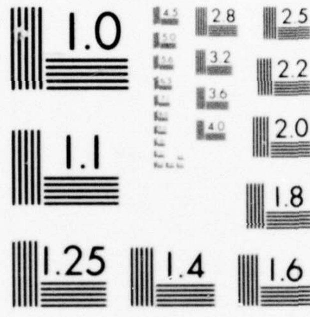
AD-A050 124 PATTERN ANALYSIS AND RECOGNITION CORP ROME N Y F/G 9/2
HIGH-ORDER LANGUAGE EXTENSIONS FOR CONCURRENT PROCESSING.(U)
DEC 77 M N CONDICT, C A LANDAUER, J M MORRIS F30602-76-C-0358

UNCLASSIFIED RADC-TR-77-394 NL

| OF |

AD A050124

END
DATE
FILMED
3-78
DDC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A050124

AD No.
DDC FILE COPY

RADC-TR-77-394
Final Technical Report
December 1977

HIGH-ORDER LANGUAGE EXTENSIONS FOR CONCURRENT PROCESSING

Mr. Michael N. Condict
Dr. Christopher A. Landauer
Dr. John M. Morris

Pattern Analysis and Recognition Corporation

Approved for public release; distribution unlimited.

DDC
RECEIVED
FEB 16 1978
B

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-77-394 has been reviewed and is approved for publication.

APPROVED:

ARMAND A. VITO
Project Engineer

APPROVED:

ALAN R. BARNUM, Assistant Chief
Information Sciences Division

FOR THE COMMANDER:

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISCA) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-77-394	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) HIGH-ORDER LANGUAGE EXTENSIONS FOR CONCURRENT PROCESSING		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report
		6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) Michael N. Condict Christopher A. Landauer John M. Morris		8. CONTRACT OR GRANT NUMBER(s) F30602-76-C-0358
9. PERFORMING ORGANIZATION NAME AND ADDRESS Pattern Analysis and Recognition Corporation 228 Liberty Plaza Rome NY 13440		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F 55971409
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISCA) Griffiss AFB NY 13441		12. REPORT DATE December 1977
		13. NUMBER OF PAGES 86
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

DDC
RECEIVED
FEB 16 1978
B

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Armand A. Vito (ISCA)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

High-Order Language
Concurrent Operations
Concurrent Processing
Concurrent Programming
JOVIAL

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report describes extensions to the JOVIAL computer language which will permit the JOVIAL programmer to exploit the characteristics of advanced computer equipment capable of concurrent non-sequential processing. Major sections of the report include a mathematical model of the processor, a detailed glossary of specialized terms used in concurrent processing, and specifications for the language extensions. Other sections include essential background material and conclusions derived from their research.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

i 390 101

TABLE OF CONTENTS

Section		Page
1.	Introduction	1-1
2.	Chronological Report	2-1
3.	Development of the Definitions	3-1
3.1.	Research Sources and Conflicting Usages	3-1
3.2.	Self-Consistency and Ambiguity	3-2
4.	Some Considerations in the Design of the Language Extensions	4-1
4.1.	Simplicity and Clarity Considerations	4-1
4.2.	Reliability Considerations	4-3
4.3.	Implementation Considerations	4-7
4.4.	Suggested Implementation Strategies	4-8
5.	A Formal Model of Concurrent Processes	5-1
5.1.	Motivations	5-1
5.2.	Numbers	5-2
5.3.	The Machine	5-4
5.4.	Representation	5-8
6.	Definitions of some Terms in Concurrent Processing	6-1
6.1.	Guide to the Definitions	6-1
6.2.	Definition of some Terms in Concurrent Processing	6-2
7.	The Proposed Extensions	7-1
7.1.	Primitive Constructs	7-1
7.1.1.	Start	7-1
7.1.2.	Stop	7-3

White Section
Buff Section
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A
PRECEDING PAGE BLANK-NOT FILMED

7.1.3.	Lock	7-4
7.1.4.	Unlock	7-5
7.1.5.	Pause	7-5
7.1.6.	Resume	7-6
7.1.7.	Await	7-7
7.1.8.	Shared Variables	7-7
7.1.9.	Process Variable	7-9
7.1.10.	Alive (Standard Function)	7-10
7.1.11.	Dormant (Standard Function)	7-10
7.1.12.	Priority (Standard Function)	7-11
7.1.13.	Authority (Standard Function)	7-11
7.1.14.	Free (Standard Function)	7-11
7.1.15.	Available (Standard Function)	7-12
7.1.16.	Clock (Standard Function)	7-12
7.2.	High Level Constructs	7-12
7.2.1.	Concurrent Compound Statement	7-13
7.2.2.	Conditional Critical Region	7-14
7.2.3.	Semaphore Operations (Standard Functions)	7-16
7.2.4.	Concurrent for Loop	7-16
7.2.5.	Queue	7-18
7.2.6.	Oldest, Newest	7-20
7.2.7.	Drop, Extend	7-20
7.2.8.	Insert, Remove (Standard Functions)	7-21
7.2.9.	Full, Empty (Standard Functions)	7-22
7.2.10.	Length (Standard Function)	7-23
8.	Conclusions and Recommendations	8-1

EVALUATION

The RADC System Architecture Evaluation Facility (SAEF) is a computer emulation facility to assist in the evaluation of hardware/software/firmware tradeoffs. An integral part of SAEF is the concept of concurrent processing. Concurrent processing has a greater ability to process real-time, high data rate Air Force problems. However, it is very difficult to program with present day serial High-Order Languages (HOL), including the Air Force command and control HOL, JOVIAL.

The objective of this effort is to develop JOVIAL language extensions to allow programming of concurrent processing activities including the proper synchronization of these activities.

The concept developed in this contract is the first step toward the HOL capabilities for describing concurrent processes.

ARMAND A. VITO
Project Engineer

SECTION 1

INTRODUCTION

This report describes extensions to the JOVIAL computer language which will permit the JOVIAL programmer to exploit the characteristics of advanced computer equipment capable of concurrent (non-sequential) processing. Major sections of the report include a mathematical model of the processor (Section 5), a detailed glossary of specialized terms used in concurrent processing (Section 6), and specifications for the language extensions (Section 7). Other sections include essential background material and conclusions derived from this research.

The JOVIAL computer language was developed for the Air Force as a tool to aid command and control system development. Based on ALGOL 58, JOVIAL has the following characteristics:

1. As a general purpose language, JOVIAL may be used for scientific and engineering problems involving numeric computation, for business problems involving large data files, for logically complex problems involving symbolic data, and for other problems involving a balance between data storage and program execution time.

2. JOVIAL is readable and concise, utilizing self-explanatory English words and the familiar notations of algebra and logic. Since

comments may be intermixed with program instructions, a JOVIAL program may serve as its own documentation.

3. Because of its ALGOL-like block structure, JOVIAL permits the subordination of detail without loss of detail; the general outlines of the program may be easily grasped by the reader.

4. JOVIAL is easily learned by programming personnel, permitting the rapid training of new staff.

5. The language is relatively machine-independent, reducing the time required for transfer of programs to new equipment or operating systems.

Computer languages such as JOVIAL are intended for sequential execution. Only one process is assumed to be active at any time, and control passes in sequential fashion from one instruction to the next. Although modern compilers are capable of modifying the order in which instructions are actually executed, the JOVIAL programmer can always proceed as if statements were to be executed in sequence.

The development of more advanced computer hardware has, however, made it desirable to extend JOVIAL to permit its use with concurrent processors. "Concurrent" and other specialized terms used in this report are defined in Section 6. Here, concurrent processing is taken to include computer operations which have one or more of the following characteristics:

1. Two or more processes may be active during the same time span. Thus, unlike sequential processing, in which each process follows in a well-defined order, concurrent processing permits several processes to be executing at once.

2. In one form of concurrent processing, several identical processes may be performed in parallel, on different data items. Such array processing might, for example, simultaneously multiply corresponding pairs of entries in two vectors.

3. A sequence of operations might be performed on many data items in pipeline fashion; when the first operation has been performed on a data item, it is passed along to the second operation, while the first operation is performed on the second data item, and so on for all processes and data.

4. Content-addressable computer storage permits the simultaneous search of many memory addresses for a specific data item, the location of which is not known in advance.

The goal of the research reported here has been the design of extensions to the JOVIAL language to permit the programmer to specify concurrent operations, of which the four examples given above are typical. In this way, the JOVIAL programmer can take advantage of concurrent processing hardware, without the need for the use of specialized, machine-dependent languages. The ease of programming in a higher-level language, and the relative machine-independence

of the JOVIAL language, can thus be used in a context which exploits available concurrency in various implementations.

This final report documents the project and presents the language design. In addition to this introductory section, the report includes the following sections:

Section 2 of this report contains a chronological narrative of the project development.

In Section 3, criteria for the development of definitions of terms are discussed.

Section 4 describes the manner in which the proposed language extensions for concurrent programming were developed.

Section 5 presents a formal model of concurrent processes, to provide the conceptual basis for development of the language.

Section 6 contains definitions of essential terms in concurrent processing.

In Section 7, detailed proposed extensions to JOVIAL for concurrent processing are presented.

Section 8 provides conclusions derived from the project.

Section 9 consists of a Bibliography of works consulted.

SECTION 2

CHRONOLOGICAL REPORT

This section reviews the research that was undertaken during this project, which extended from July 1976 through August 1977. This chronological survey serves to place the project in historical perspective.

The initial two weeks of the contract period were spent in reviewing articles and books by authorities in the field of concurrent processing, drawn from the Bibliography included at the end of this report. The purpose of this review was to obtain data concerning accepted use of terms in concurrent processing.

A tentative list of words and proposed definitions was then compiled, consisting of all those relevant words which were used by several different authors, other words considered useful or important, and all words specified in the Statement of Work. The list included only terms relevant to concurrent processing and not dependent on particular types of hardware.

The proposed list was thoroughly reviewed and discussed by PAR's technical staff with experience of interest in concurrent processing. The purpose of this review was to propose new terms and/or revised definitions. The discussion included persons with a variety of technical backgrounds, in order to introduce diverse viewpoints and to produce broadly acceptable definitions.

Following discussion with RADC, some of the proposed definitions were revised by adding explanatory notes and examples, and some new terms were added.

Next, a set of tentative language structures, which had been developed for PAR's initial proposal, were reviewed with regard to convenience, clarity, and power. Several additional structures were considered.

Potential advantages and disadvantages in the use of the JOVIAL language as the basis for the proposed language design were also reviewed with RADC. On the basis of this discussion, it was decided that JOVIAL would be used as the base language.

By October 1976, we were ready to study a large set of concurrent functions, including semaphores, process control, conditional waits, communication, and others, in order to be able to select a consistent and powerful subset of primitive operations. The problems which might arise through the use of these functions were analyzed to determine whether they are intrinsic to the function or can be avoided by proper constraints on the use of the functions. Finally, the feasibility of implementing these functions on today's machines and operating systems was discussed by the project team and other interested personnel.

The JOVIAL J73 dialect was studied to determine the types of syntactical extensions which would be consistent with the rest of the language, without introducing ambiguity into the language. Precise syntax and semantics were developed for some of the concurrent processing primitives, and additional

high level functions were developed and analyzed for convenience, power, and reliability.

To provide conceptual foundations for the language design, a formal mathematical model of a concurrent processing system was developed. This is intended to serve as the basis for more precise definitions of most of the terms in concurrent processing which were heretofore only informally described. It also allowed a better awareness of the properties and complexities involved in concurrent processing.

By February 1977, the basic outlines of the proposed language had been completed. From this point forward, the major goal was that of testing existing structures for consistency and completeness, and reviewing additional features for possible inclusion. The research approach, which relied heavily on regular discussions among project members and other interested personnel, meant that a fairly large number of persons were involved in the project as critics and sources of additional ideas. These discussions were continued through April 1977.

Details of the formal concurrent processing machine were worked out. A small language for programming this abstract machine was designed, and investigations were begun to determine how well the machine models the important aspects of present-day concurrent computers and operating systems.

The proposed language extensions were under constant review during this period. Reliability considerations arose in the analysis of the proposed

extensions. It was found that there is a clear trade-off between increased power and safety from mistakes or misuse.

During May 1977, the design of test methods for determining the quality of the concurrent extensions was begun. These tests attempt to measure the user acceptability of the extensions (in terms of convenience, power and clarity) and to measure their reliability (the likelihood that the programmer will use them correctly).

The power of the formal concurrent processing machine was studied by using it to describe other machines. It appeared that some modifications of it would be necessary in order to model concisely all of the proposed concurrent functions.

By September 1977, final revisions of the definitions of concurrent processing terms were completed, reflecting insights gained from the study of the mathematical model. The syntax and semantics of some of the proposed extensions were modified after tests showed them to be inadequate in some respects. New constructs and standard functions were defined. Finally, conclusions and recommendations based on results of this project were developed. The next few sections of this report will document the research in more detail.

SECTION 3

DEVELOPMENT OF THE DEFINITIONS

3.1. RESEARCH SOURCES AND CONFLICTING USAGES

One of the major stumbling blocks to advancement in the understanding of concurrent (parallel) processing has been the multiple and conflicting usage of many of the new terms in the field. This is understandable in an area of knowledge which is expanding so rapidly, but it hardly contributes to lucid communications between researchers. In our efforts to improve the situation with a set of precise definitions for a carefully chosen set of terms, we made several difficult decisions regarding the accepted usage, as seen in the literature, of some of these terms. For example, we found the phrase "associative processing" to be used to mean what we have called "concurrent processing" and also to mean "array processing." The phrase "parallel processing" has been used to mean any of "order-independent processes," "concurrent processing," "multiprocessing but not multiprogramming," and several other types of processing. Our definitions of terms appear in Section 6, and the Bibliography (Section 9) details some of the sources referred to in constructing the list.

The main guideline we followed in resolving conflicts was to attempt to identify the important concepts in the area of concurrent processing, regardless of the names used to refer to such concepts, and to identify concepts which were important but had not heretofore been given generally accepted

names. In the latter cases, we attempted to choose a meaningful name from non-technical English. Finally, where all existing names in use for a certain concept seemed equally bad choices, because of misuse or lack of mnemonic quality, we chose a new name from non-technical English.

3.2. SELF-CONSISTENCY AND AMBIGUITY

Limits of precision in the English language have made it impossible to eliminate completely ambiguity in the definitions, but we have tried to limit such ambiguities by first defining, as clearly as possible and with several examples, a set of basic concepts, which could then be used to define precisely all the other more complicated terms. Thus, if these first definitions are well understood, or at least taken on faith, the definitions of the others should be just as well understood. The basic concepts include: <u>resource</u>, <u>program</u>, <u>process</u>, <u>initiate</u>, <u>terminate</u>, <u>processor</u>.

Equally important and desirable is the consistency of the definitions; that is, they must not contradict one another. One of the reasons that the formal model was developed (see Section 5), was to provide a tool for defining the terms even more precisely (in terms of the operation of the model), and to ensure that the definitions were consistent. While the potential of the model has not yet been fully realized in this area, it did guide us in revising several of the definitions to remove inconsistencies and to clarify their application.

SECTION 4

SOME CONSIDERATIONS IN THE DESIGN OF THE LANGUAGE EXTENSIONS

4.1. SIMPLICITY AND CLARITY CONSIDERATIONS

The literature abounds with proposals for new language structures with which to do concurrent processing. It is therefore easy for a language designer to fall into the trap of attempting to include all of them in his concurrent programming language. This can result in a mishmash of alternative ways of doing the same thing, with no clear choice between them. As with the definitions of terms (see Section 3) we avoided this pitfall by selecting the most basic, minimum set of structures, from which all other structures can be defined. Of the published alternative proposals, we chose as the basic set those which were most easily understood and had the least complicated syntax and semantics. Reliability of use was not of prime concern for these structures since they are intended to be used only when the more restricted, high-level constructs are not powerful enough.

The same requirements for simplicity and ease of use were also adhered to in the choice of the high-level constructs. Since they are equivalent to combinations of the basic constructs, their semantics can be described precisely by means of a transformation rule, showing how one construct in any valid program can be changed into another, without changing the semantics of the program. To be a little more lucid, while less formal, we have, instead, shown two equivalent program fragments as examples, one using the high-level

construct and the other using a combination of basic constructs. It is believed that the examples are general enough to define the equivalence relation for all cases. The examples do not necessarily show the proposed way of implementing the high level structures, but rather the simplest way of illustrating their meaning. The complete syntax and semantics of the proposed extensions are found in Section 7. In the following subsections we justify some specific choices which were made among several alternative constructs.

1. Brinch Hansen's critical regions (see ACQUIRE statement) were used instead of monitors (procedures containing data together with procedures for accessing the data, and enforcing mutually exclusive access to the data), because critical regions seem more flexible and thus easier to use. Also, it is relatively easy to obtain monitors through the use of critical regions, by placing the body of each of several procedures in a critical region on the same shared variable.

2. The concept of locking and unlocking (see LOCK, UNLOCK) shared variables was chosen over semaphores because it is easier to see directly how it can be used to provide mutually exclusive use of a variable. Actually, locking a variable is equivalent (except for the feature of multiple locks - see Reliability) to doing a wait semaphore operation on a semaphore (initialized to 1) devoted to enforcing exclusive access to that variable. Similarly, unlocking the variable is equivalent to a signal operation on the semaphore. As seen in Section 7, in the definitions of semaphore functions, it is trivial to define them using LOCK and UNLOCK on integers.

3. The START statement, with the code of one process appearing inside the code of its parent process, was chosen instead of the separately declared processes of Brinch Hansen's Concurrent PASCAL and the tasks of PL/I. It is not clear why the statements to be executed by one process should be required to be physically distant from the statements of the parent process, especially if they are quite simple. It was felt that this detracts from readability of programs more than it would improve modularity. The "fork" operation, proposed by others, is even more basic than the START statement, but was deemed too easy to misuse (see Subsection 4.2.).

4. QUEUEs consisting of a bounded number of similar components were defined to be as much like TABLEs as possible, to facilitate familiarity and avoid a proliferation of syntax rules. Again, simplicity dictated defining a minimum number of queue operations, with others constructed as standard functions from these.

4.2. RELIABILITY CONSIDERATIONS

Although all of the proposed extensions have legitimate uses for certain tasks, it must be kept in mind that the primitive constructs (especially LOCK, UNLOCK and START) are just that -- primitive. Their use should be reserved for performing functions which cannot easily or efficiently be performed with the more restricted concurrent processing constructs. Some of the possible pitfalls and the manner in which they are avoided by the use of high-level constructs are outlined below:

1. The statements most likely to cause deadlock, or other errors if improperly used, are the LOCK and UNLOCK statements. Forgetting to lock a variable before use and then unlocking it, or locking it and then failing to release it to other processes, can cause obvious problems. The ACQUIRE statement was designed to ensure that all locks and unlocks are matched properly. Less obvious is the deadlock caused by waiting (see AWAIT) for a change in a condition of a variable which is locked by the waiting process. Finally, deadlock can be caused by the interaction of two processes both attempting to lock two or more variables. If process P holds variable A and attempts to lock B, but process Q already holds B and is attempting to lock A, neither one will ever complete execution of their LOCK statements. This is the reason the LOCK statement was defined with a list of variables, instead of just one. The list will always be processed according to some pre-defined ordering of all variables (probably address order). Thus, deadlock cannot occur from this cause unless nested LOCKs are used.

3. One of the constructs that has been proposed by others to initiate new processes is the fork statement, which usually initiates a new process in the same state as the original except that they are each given some way to discover whether they are the parent process or the child process. Neither process is confined to any section of the program, and the parent-child relationship is obscured. Also, if the processes fail to "separate," by discovering which of the two processes they are and acting on that information, they may produce

non-deterministic results. The START statement was designed to overcome these difficulties by restricting the new process to executing a separate block of statements, with an automatic termination when it reaches the end of the block. In this way, the new process can be seen to be nothing more than an invocation of the block, but one in which the invoker does not wait for the block to be completed.

The START statement, however, is not entirely free of reliability problems. Just as with the use of a fork statement, we cannot be sure how many processes can be active at any given statement, because we do not know when they finish, and have not synchronized them. The concurrent compound statement (see COBEGIN, COEND) or concurrent FOR loop is usually powerful enough to replace the use of the START statement and is less prone to error, because a single sequential process enters the statement and a single process (the same one) leaves the statement at the bottom. In fact, if the processes inside the concurrent COBEGIN or FOR statement operate on disjoint sets of variables and are otherwise order independent, the statements are equivalent to their sequential counterparts.

4. It was found difficult to make safe and generally useful rules to determine which processes may control the execution of other processes (see PAUSE, RESUME, STOP). Many concurrent processing implementations state by definition that a process can control another if it has higher "priority." Also, higher priority processes will execute faster and wait less for resources to become

available. But it seems to be a confusion to have control permission and speed - two independent concepts - depend on the same attribute of processes. To illustrate the problem with this approach consider the following two examples:

a. A system-accounting process could be initiated to gather statistics about another process, or to restrict the process in some way. It should be a low-priority process so as not to place much of a computing burden on the system, since it is pure "overhead"; but the monitored process, which is of higher priority, should not be allowed to abort the accounting process, for security reasons.

b. A process performing a task with strict timing constraints (such as reading a fast magnetic disk) might need high priority to operate correctly, but the lower priority process which requested it may legitimately want to abort it when the latter discovers that the former is in error, especially if the high-priority process is expected to be lengthy or expensive.

Another approach to the problem of control permission is to allow a parent process to control execution of its child processes (and possibly further descendants), but not vice versa. This rule also falls down under certain circumstances. For instance, in a concurrent data base management system, a user process might initiate a system process to do an update on the data base, then decide the transaction was a mistake and wish to abort the update process.

However, premature termination of the process could leave the data base in an inconsistent state and so cannot be allowed.

For these reasons, it was deemed necessary to allow a process, when initiating another, to specify control permission by means of an "authority" value, an attribute of all processes (see START, process variable). In order for one process to control the execution of another process, it must have greater or equal authority. Note that a system process may protect itself from being controlled by the user process which initiated it, if it is always required to be called indirectly from another system process, which will initiate it with a high authority.

4.3. IMPLEMENTATION CONSIDERATIONS

Although this project did not require implementation of the language extensions, it would be irresponsible for a language designer glibly to define operations which can be implemented, on most machines, only at great cost and with gross inefficiencies. So, for each proposed language feature, we outlined possible implementation schemes on various kinds of machines. The machines considered included:

1. Array processors.
2. Processor networks (with or without shared memory).
3. Multiprogrammed operating systems on sequential computers (with or without control of interrupts).
4. Single user operating systems on sequential computers.

As a result of such considerations, several changes were made to the extensions, including:

1. Requiring certain variables to be declared SHARED.

2. Requiring all functions referenced in an AWAIT statement to be "pure," so that their returned value depends only on their input parameters.

3. Prohibiting a process from branching outside the confines of the START statement which initiated it, except for procedure calls.

4.4. SUGGESTED IMPLEMENTATION STRATEGIES

Following are some notes on possible implementation methods for specific concurrent constructs:

1. LOCK, UNLOCK. The difficulty of implementing these functions varies greatly with the type of machine involved. But Dijkstra has shown that, as long as reads and writes of shared memory can be performed as indivisible actions (where the exclusion might be enforced by a memory controller or other low-level hardware), then the functions can be implemented, albeit not very efficiently. Moreover, they can be implemented in a queued fashion without threat of indefinite overtaking. The implementation is much simpler and faster if it is possible to do an indivisible "test and set" operation on shared

variables. That is, the value of a shared variable may be copied to local (unshared) storage in a process, and then changed before any other process can read it. While not many conventional computers have such an instruction, it is possible to accomplish the same thing if the machine can be temporarily "sequentialized" by turning off its interrupt capability. The algorithm used with the "test and set" operation involves incrementing an integer associated with a particular shared variable immediately after copying it to a local variable, then waiting for the local copy to become equal to another shared integer which is incremented by processes as they release (UNLOCK) the shared variable.

2. START, STOP, PAUSE, RESUME. On some operating systems, these are simply calls to the executive. An existing process may request a fork-like operation, or may specify that a different program is to be invoked as a concurrent process. On sequential machines without this feature, a concurrent program could be translated into a sequential program with a built-in executive that simulates concurrency by successively executing parts of different logical processes. This method is to be avoided, if possible, because the set of routines which make up the executive will probably be quite large, and will tend to duplicate the functions of the actual system executive, resulting in confusion and complexity.

In networks of processors, it may be advisable for reasons of simplicity to restrict each processor to performing one logical

process, placing an upper bound on the number of processes allowed to be alive simultaneously. Then it is not clear whether starting a new process has any meaning at all since a new processor cannot be automatically materialized to perform it. However, it might be useful to define the stopping of a process as the execution of a halt instruction on one of the processors, and to define the start of a new process as the restart of one of the halted processors. If processor time is inexpensive, PAUSE and RESUME could be defined in terms of the execution of a "busy" wait loop.

3. Standard Functions. The standard functions which return information about processes could look such information up in a list of active processes, maintained by a controlling executive, or by the processes themselves.

4. AWAIT. This statement is the most difficult and costly of all to implement, but is so powerful and useful that it was chosen over the more limited "event-processing" which others have used as a synchronization primitive. (Awaiting one of these events is like executing an AWAIT statement where the logical formula is limited to being a single logical variable, which must have been declared in a special way, and whose value must be set with a special statement.) The only reasonably efficient way to implement the AWAIT statement seems to be to have a linked list associated with each variable declared SHARED, and containing the process identifications of all processes currently awaiting a logical formula in which the shared variable is

important. The head of the list could be a word of storage allocated beside the shared variable. Then every assignment to a shared variable must be followed by a test to see if the list of waiting processes is empty. (This probably requires only one or two machine instructions.) If not empty, every process in the list must be resumed so that it (or its executive) recalculates the value of its logical formula and decides whether to continue or suspend again.

5. SHARED variables. The SHARED declaration was introduced to permit the compiler to generate the extra data structures necessary with such variables. (See implementation of AWAIT and LOCK.) These will probably include a pointer to a linked list of processes waiting for the value of the variable to change, and a pointer to a queue of processes waiting for access to the variable or a pair of integers by which processes enforce queued access themselves (as described in LOCK implementation).

SECTION 5

A FORMAL MODEL OF CONCURRENT PROCESSES

5.1. MOTIVATIONS

This section describes a mathematical model for investigating concurrent processing in computers. The purpose of the model was to give a firm mathematical foundation to a set of English-language definitions of terms of importance in concurrent processing. It is our hope that the informal definitions are adequate, but we realize that another authority is needed in the inevitable cases of ambiguity.

The existing models of concurrent processing do not include all of the features that are essential to a complete model. In particular, we insist that the model be flexible enough to permit some data to be considered as programs. This requirement tends to complicate the model, but it is balanced by the further requirement that it be possible to implement a substantial portion of the model on an ordinary (Von Neumann) computer.

The model will be described in terms of a particular idealized computer called the OSMA machine, together with rules for definition and execution of programs on the machine.

An OSMA machine contains a set of variables which completely describe its state and situation. On an ordinary computer, these variables would include

all storage and all local or internal registers. The machine also has a set of currently active processes, whose action can be enabled or disabled according to conditions on some of the variables.

The action of a program can be described by a function from the set of input values to a set of output values. Since we wish to include all parts of an ordinary computer in a single OSMA machine, we will not speak directly of input or output devices. We will therefore assume that there is a single set of variables on which each process operates. This subset is contained in some large set of variables belonging to the machine itself.

5.2. NUMBERS

Any digital computer deals with symbols, which have by historical convention been associated with numerical quantities. Therefore, our machine will have a fixed set of symbols, which act as possible values of variables. We denote this set by N, and call the elements of N "numbers" (take note that this term will be used only in this sense throughout this section). It follows then that a process can be described by a function from one set of values of variables to another.

The set of possible values of a variable in an ordinary computer is finite, and the set of all finite sequences of such values is countably infinite. For mathematical simplicity, we assume that the set N is countably infinite, since it allows a single variable to contain the equivalent of a full program.

A major problem from the mathematical point of view is that programs are self-referential. Since the output of one program can be another program, it must be possible for any program to be represented by a number (element of N). However, the set N^N of all functions from N to N has cardinality

$$|N^N| > |N| ,$$

for any set N with $|N| > 1$. This fact implies that no matter what set of numbers is used (as long as there is more than one number), there are more possible functions than there are function representations. Therefore, we can only use a subset of all possible functions for our machine, and so we need to define a suitable collection of functions. Fortunately, such a collection exists.

We will use the results from the theory of recursive functions in mathematical logic (see Hermes, Yasuhara in the Bibliography), the recursive functions afford the best known candidate for the intuitive idea of effectively computable functions. In addition, every computer has only a finite number of components, and any such machine can represent only recursive functions.

For this exposition, we assume that N is the set of non-negative integers. This choice is compatible with the standard expositions of recursive function theory, and with our choice of the size of N. It can be circumvented (e.g., we might take N to be the set of all finite sequences of characters

from a finite alphabet), but in this case the model would require some modification.

Since the variable values must describe function definitions and not functions, we must find a scheme for translating function definitions into elements of N. It must also be assumed that the OSMA machine can interpret an element of N as a program. The mechanism by which this interpretation is performed need not be described. We need only to know that there is a way, which will be clear when the translation in the other direction (i.e. from programs to numbers) is shown to be one-to-one.

5.3. THE MACHINE

The configuration of an OSMA machine consists of variables and instructions.

There is a countable set V of variables, labeled by numbers, and a method for allocating and freeing disjoint countable subsets of V. Also, every running program has two kinds of variables, global and local. The global variables are common to all programs, and the local variables are specific to a single invocation of a program.

Specifically, the odd numbers might correspond to global variables, and for each positive integer n, the nth set of local variables might consist of those which correspond to numbers congruent to 2^n (modulo 2^{n+1}).

The instruction set I is a countable set of functions on sequences of numbers of various lengths, and it is assumed that I contains all the primitive recursive functions. (The precise definition of primitive recursive functions can be found in Yasuhara.) All we need here is that this set includes most of the functions found in computer programs.

It is permissible for I to contain other functions which are not primitive recursive. We will see an important example of such a function, the oracle, later in this subsection.

The state of the OSMA machine is a specification of the values of its variables. The state of a process is a specification of the values of its global and local variables, and the number representing its program (i.e. its static description).

If there is to be meaningful interaction between the components of a concurrent architecture, it must be possible for requests for resources to arrive arbitrarily close together. In any actual device, such access implies and requires some kind of arbitration to determine which request is granted and which is refused.

In order to allow for multiple requests and arbitration, we have determined that an OSMA machine will have a sequential controller (i.e. instruction decoder), so that exactly one instruction will be executed at any one time. The OSMA machine is still a concurrent processor, since the instructions are arbitrarily, and in most cases unpredictably, interleaved.

The machine language for an OSMA machine is, of course, the set N of numbers, since we have already claimed that every program is to be a number. In the rest of this section, we will describe the interpretation of programs written in the OSMA machine language. We will also define the terms used to describe the parts of OSMA machine language programs.

Each program uses the common set of global variables and a unique set of local variables (each of these sets is countable). The program must know which variables are local and which are global. Any system by which this distinction is made is acceptable.

An OSMA machine language program is considered as a sequence of transition definitions. A function expression is an application of a function $f \varepsilon I$ to an appropriate argument list of variables (either global or local). A predicate expression is also a function expression, where the function values are interpreted as truth values, zero for false and non-zero for true.

A transition definition is a triple consisting of a condition, an action, and an invocation. A condition is simply a predicate expression used as a gate for the action. An action is either a halt instruction (to cause the process to terminate) or a finite nonempty sequence of assignments of the form $d \leftarrow e$, where e is a function expression and d is an address expression, i.e. a function expression whose value is used as a variable. By convention, the halt instruction is the case of no assignments (thus the two actions may be considered under one heading).

The invocation part of a transition definite is considered to be null if it is zero. If it is nonzero, it causes a new process to be invoked with a unique set of local variables. The value of the invocation is a variable reference, and the value of that variable is interpreted as a program. However, no subsequent change in the variable has any effect on the program that is interpreted (it is as if the program is copied elsewhere for interpretation).

The OSMA Intermediate Source Language is a high-order language which explicitly reflects the structure of a program, and its organization into the type of hierarchy described above.

An oracle is a function which is designed to model an interactive terminal. In particular, it is used as a function of a specific number k of variables (considered as "displayed output"), although its value (considered as "typed input") need not be the same for two invocations, even if the arguments are the same.

Each oracle must be in the original instruction set I, since it cannot be given a definition as a primitive recursive function.

We describe the execution of programs on the OSMA machine. The machine begins with an initial state and a finite collection of processes which are active. At this point, the machine is ready to select an instruction.

The selection and execution of instructions is always carried out in the same way. For any state of the machine, zero or more of the active processes have transition definitions for which the condition part is true. Such transition definitions are called enabled steps. It is quite possible for a process to have more than one enabled step. If no step is enabled, then the machine has reached a deadlock, and cannot proceed. This situation must be considered an error condition.

If there are any enabled steps, then the instruction part of exactly one of them is executed. As a result, the process containing the instruction may halt (i.e. disappear from the set of active processes), or a change of the current state may occur, as determined by a set of assignments in the instruction.

In addition, if the invocation part of the instruction is nonzero, a new process, whose static description is given by the specified program, is placed in the set of active processes, with its local variables all set to zero. At this point, the machine is ready for a new selection.

5.4. REPRESENTATION

It is our aim in this section to describe the representation of all of our structures by numbers. The representation function will be described recursively, and each kind of structure will be mapped in such a way that it is uniquely determined by the representing number (given the type of structure).

Since the functions in I form a countable set, we begin by associating with each $f \in I$ a number $|f| \in N$.

The variables in a program are easily represented. The global variable v is represented by an odd number $|v| \in N$, and the local variables are similarly represented by even numbers.

In order to describe the representation of combinations of objects, we need a way to represent the structure in terms of its parts. In particular, pairs of objects can be represented by a single number if we can find a "combination" function: one that maps pairs of numbers uniquely to single numbers. Most textbook descriptions of the recursive function theory describe such a function on pairs, and a few have explicitly given the function θ_2 below.

We have found a simple description for a collection of functions θ_k which map k-tuples of numbers uniquely to numbers. Moreover, the function is invertible. This fact means that every k-tuple of numbers can be mapped to a unique single number, and also that every single number represents a unique k-tuple of numbers (for a given k).

These functions (except for θ_2) have not appeared in an easily accessible source, and we have not seen them elsewhere. We digress for a moment here to describe these functions and some of their properties.

We denote by N_k the set of all k-tuples $(x_1, 111, x_k)$ of numbers (i.e. $x_i \varepsilon$ N for $1 \leq i \leq k$). For k = 1, this set is just N. Define θ_1:N N to be the identity map on N.

For $k \geq 2$, we will define functions

$$\theta_k : N^k \rightarrow N$$

which are bijective (i.e. one-to-one and onto).

Note that $\theta_1(x)$ maps $\{(x,y) | \; x,y \geq 0, \; x + y = m\}$ bijectively onto $\{0, 1, \ldots, m\}$, since the conditions on x,y are equivalent to $0 \leq x \leq m$, y m - x.

Define $\theta_2 : N^2 \rightarrow N$ by

$$\theta_2(x,y) = \binom{x + y + 1}{2} + x,$$

so that θ_2 maps $\{(x,y) | x,y \geq 0, \; x + y = m\}$ bijectively onto the set

$$\{\binom{m + 1}{2}, \ldots, \binom{m + 2}{2} - 1\},$$

so that θ_2 maps $\{(x,y) | \; x,y \geq 0, \; x + y \leq n\}$ bijectively onto the set

$$\{0, 1, \ldots, \binom{n + 2}{2} - 1\}.$$

It follows immediately that θ_2 is bijective.

Similarly, for $k \geq 2$, we define $\theta_k : N^k \rightarrow N$ by

$$\theta_k(x_1, \ldots, x_k) = \binom{n + k - 1}{k} + {}_{k-1}(x_1, \ldots, x_{k-1}),$$

where $n = \Sigma(1 \leq i \leq k)x_i$. It can be shown as above that θ_k is one-to-one and onto.

Write N^o for the set containing the empty sequence, and N^∞ for the union

$$N^\infty = \cup(k \geq 0)\ N^k,$$

and define a function $\theta_\infty : N^\infty \rightarrow N$ by taking $\theta_\infty(()) = 0$ and for $k \geq 1$, $n \varepsilon N^k$,

$$\theta_\infty(n) = 2^{k-1} + 2^k\ \theta_k(n).$$

Then it is clear that θ_∞ is also a bijection.

The process by which θ_k^{-1} is computed is fairly simple. Given k and n, we choose the largest integer m for which

$$\ell = \binom{m + k - 1}{k} \leq n$$

(there is one, since m = 0 gives 0 on the left), and then compute

$$(x_k, \ldots, x_{k-1}) = \theta_{k-1}^{-1} (n - \ell),$$

and finally

$$x_k = m - \Sigma(1 \leq i \leq k-1)\, x_i .$$

For the case k = 1, $\theta_k^{-1}(x) = x$, so the inverses are also computed recursively.

The inverse of θ_∞ is computed as follows: given n, if n = 0, then $\theta_\infty^{-1}(n) = ()$, the empty sequence. If $n \neq 0$, then there is a largest power of 2 dividing n, i.e. let k be the largest integer for which

$$2^{k-1} \text{ divides } n.$$

Then $k \geq 1$, and we compute

$$\theta_\infty^{-1}(n) = \theta_k^{-1}((n - 2^{k-1})/2^k),$$

using the recursive procedure for θ_k^{-1}.

We will use these functions to define a representation of programs by numbers. This representation has, as a side effect, the representation of all structural units of a program as numbers.

A function expression (or a predicate or an address expression) e is an application of a function f to a sequence of $n \geq 0$ variables $x_1, \ldots, x_n$. We then write

$$|e| = \theta_2(|f| , \theta_\infty(|x_1| , \ldots |x_n|)).$$

An assignment is a pair $d \leftarrow e$, and we define

$$d \leftarrow e = \theta_2(|d| , |e|).$$

An instruction a is a halt, or a finite sequence of $r \geq 1$ assignments, so we write $|a| = 0$ for a halt, and

$$|a| = \theta_\infty((|d_i \leftarrow e_i| \; 1 \leq i \leq r))$$

for the sequence $(d_i \leftarrow e_i \; |1 \leq i \leq r)$ of assignments.

A transition definition t has three parts: a condition c, an instruction a, and an invocation v. We define

$$|t| = \theta_3(|c| , |a| , |v|).$$

Finally, a program p is a finite sequence $(t_i \; |1 \leq i \leq r)$ of transition definitions, and we define

$$|p| = \theta_\infty((|t_i| \; |1 \leq i \leq r)).$$

We assume that $r \geq 1$, so that $|p| \neq 0$ for a valid program.

We now claim that given the type of a structure, the integer representation uniquely determines the structure. The different structure types are:

program

transition definition

instruction

assignment

function expression

variable

function

The result follows from the definitions, since at each stage of the construction, where a function θ_k of θ_∞ was used, its arguments could range over all possibilities.

An OSMA machine language program (i.e. an element of N) can therefore be interpreted (using a θ-function) as a sequence of transition definitions, each of which is a triple; each triple is interpreted as a condition, an instruction and an invocation; etc. The interpretation is uniquely recoverable from the value of the representation, given the type of structure, so that the representation of each structure can be interpreted in a reasonable and consistent fashion.

SECTION 6

DEFINITIONS OF SOME TERMS IN CONCURRENT PROCESSING

6.1. GUIDE TO THE DEFINITIONS

Throughout the definitions, the following conventions are used:

1. A word underlined in a definition is defined elsewhere.

2. The word or phrase which is being defined in a sentence is enclosed in quote marks.

An attempt was made to order the definitions so that the most primitive concepts, not definable completely in terms of other concepts, appear first. It will be noted that several of these early definitions (variable, program, process, processor) refer, circularly, to themselves. This seems unavoidable. It is hoped that these terms are basic enough and their meanings well enough agreed on that ambiguity is minimized.

The synonyms given are not claimed to be exactly the same in meaning as the word being defined. They are, rather, words which others have used to refer to similar concepts, and are included as an aid to the reader who may be more familiar with one of them than the word chosen to appear at the left. In most cases, this choice was governed by three criteria:

1. The word was more widely or precisely used in the literature.

2. The word that corresponded most closely to its non-technical ("Webster") definition was often chosen for mnemonic reasons (e.g., concurrent instead of parallel).

3. When no clear choice was apparent because all existing words in use were equally bad choices, a new one was taken from non-technical English that, hopefully, satisfied criterion 2.

6.2. DEFINITION OF SOME TERMS IN CONCURRENT PROCESSING

<u>resolution</u> In any consideration of processing, there are many conceptual levels at which activity takes place. "Resolution" will be defined here to be the smallest (or shortest) action that is to be treated, for the purpose of discussion, as a single atomic action.

<u>Examples</u>:

To the naive user of a computer program which is accepting and executing commands of some sort, the smallest resolution known is the execution of a single command. The author of the program, however, is concerned with a lower level of resolution when he constructs the program, namely the execution of single statements, instructions, or procedures. Similarly, the

resolution with which the designer of the computer views the same activity may extend down to the level of electronic signals in solid state circuits.

It is important to note that activity which appears to involve several independent concurrent processes at one level of resolution can often be seen as parts of the same sequential process at a more inclusive level of resolution. This does not affect the need to treat the activity as separate processes at the lower level. An example is viewing a multi-programmed operating system on a sequential computer first at the user program level, then at the level of single machine instruction executions.

Synonyms: grain size

resource Those things upon which processes act and which are required by processes to perform some desired function.

Examples:

In a computer system where the processes are invocations of user programs, typical resources are mass-storage devices, such as tapes, disks, etc. and input/output devices, such as CRT terminals and line-printers. To a process consisting of a

functional unit of a computer, a resource might be a data bus or an internal register. The resources of an abstract FORTRAN process include its variables and the files upon which it acts.

<u>program instruction</u> A sequential "program" consists of a set of rules which explain how (i.e. with what parameters), and in what order, to execute instructions (and when to stop) from a known set of available instructions. Each "instruction" is a program itself, when viewed at a lower level of <u>resolution</u>.

Every program is defined with respect to a particular <u>processor</u>, since each <u>processor</u> may have a different set of instructions available.

<u>Examples</u>:

A FORTRAN program; micro-code in the control store of a micro-programmed computer which controls the execution of a single machine instruction; a flow chart; a cook-book recipe.

<u>Synonyms</u>: algorithm, procedure

<u>process</u> A particular execution, from start to finish, of a <u>program</u>. Every process is a series of unique events which by definition occurs only once. Two processes from the same <u>program</u> are

distinct if they are initiated at different times or places, or act on different resources, or are active on different processors.

Synonyms: task, job

processor That mechanism, whether physical or logical, which performs (or executes) a program (also called supporting a process).

Examples:

The concept of "processor," as with several other terms, depends on the choice of resolution. At one level, a sequential computer together with a multiprogramming, time-sharing operating system might look like an (almost) arbitrary number of identical processors to users at remote terminals. This view is as appropriate, for purposes of most analyses, as the view from the machine instruction level, where there appears to be only one sequential processor. Other examples: the human brain; the logic circuitry of a desk calculator; an abstract FORTRAN machine consisting of a FORTRAN compiler, an operating system and computer, and a library of support subroutines which allow FORTRAN programs to execute under the operating system.

Synonyms: Machine, programming language

advancing A process is said to be "advancing" whenever it is alive and not dormant. Nothing is implied about whether its supporting processor is currently executing one of the instructions of its program or has finished one instruction and not yet begun another. All that is important is whether or not the process is able to continue.

Example:

When viewing a multiprogramming, time-sliced system as a set of identical processors with one or more available for each user job, the jobs are said to be advancing even during those periods of time when they could have used the CPU, but it was not available until a later time-slice.

Synonyms: active, running executing

dormant A process is "dormant" whenever it is alive but not progressing towards completion, because it is waiting for some resource to become available or for some other condition to become true.

Example:

In a typical operating system, a job (process) might be dormant because it is waiting for information to become available at the read/write head of a slow mass storage device.

Synonyms: waiting, paused, blocked, sleeping

order independent Two processes are "order independent" if, whenever there is a choice at a particular time of executing an instruction in either one (i.e. they are both advancing), the final outcome does not depend on which processor executes an instruction first. Processes which act on disjoint resources, and which therefore do not communicate in any way, are always order independent.

cooperating Two processes, A and B, are "cooperating" if A changes the state of some resource (or creates the resource) after which B executes instructions which depend on the state or existence of that resource, and B effects A similarly. Processes must be concurrent in order to cooperate.

Example:

If A and B are a user of a remote computer terminal and a process in the computer, respectively, and if the process is the invocation of a program which obeys some sort of types commands, and, finally, if the commands chosen by the user may depend on the results of previous commands, then A and B are cooperating processes.

lifetime
active
alive

The "lifetime" of a process is the time period from when it is initiated until it terminates. A process is said to be "alive" or "active" during its entire lifetime.

concurrent

Two processes are "concurrent" if and only if their lifetimes overlap (are not disjoint). In other words, concurrency requires that there be some period of time during which both processes have been initiated and neither has terminated.

resume

A process "resumes" (or "resumes activity") when it begins advancing after being dormant. One process is said to "resume" another when an instruction in it directly causes the latter process to resume activity.

asynchronous

Two processes are "asynchronous" if their relative rate of execution does not affect the outcome of either. Their order of initiation and termination may, however, be important, unlike order independent processes. It is common to say that one process starts up another "asynchronous" process (i.e. the two processes are asynchronous) if the first process needs to know only when the second is initiated and when it terminates.

Example:

If A and B are, respectively, a user job in a computer and a device handler program which services such jobs, it is common

for A to request B to initiate a device operation. Then A continues without waiting for completion, depending on B to inform it when the operation is complete. Under these circumstances, A and B are asynchronous processes.

<u>multi-programming</u> A technique for executing several logical <u>processes</u> <u>concurrently</u> on one physical <u>processor</u> by interleaving their execution, although only one <u>process</u> may occupy the <u>processor</u> at a time. Multiprogramming is used to increase efficiency in multi-user computer systems by allowing one user's job to use the central processor while others are waiting for slow mass-storage <u>resources</u> to become ready.

<u>Synonym</u>: multitasking

<u>multi-processing</u> The simultaneous execution of <u>programs</u> on several <u>processors</u> with some form of communication between <u>processes</u>. It is usually used to divide jobs logically or functionally and to increase computer power.

<u>array processing</u> The simultaneous execution of a single sequence of <u>instructions</u> by several <u>processors</u>, with possibly different inputs and outputs. The I/O can be thought of as arrays or vectors with the ith <u>processor</u> receiving the ith input and producing the ith output. The <u>processors</u> are synchronized in that they are all

executing the same instruction in the sequence at any given time, but with different data. Computers built to do array processing at the machine instruction level (i.e. the instructions are single machine instructions) have been called single-instruction multidata machines (SIMD).

Synonym: vector processing

pipe A "pipe" exists from one process, P, to a concurrent process, Q, if P produces an output data sequence which is an input data sequence for Q.

pipeline A "pipeline" is a sequence of pipes connecting several processes so that output of the ith process in input to the $(i+1)^{th}$ process in the pipeline.

Example:

The concept of a pipeline is useful to both hardware and software engineers but for somewhat different reasons. A hardware pipeline (e.g., between subsystems in a CPU) is usually the result of decomposing a program, P, into an equivalent sequence of programs to be executed as pipelined processes. The program, P (which may be, e.g., execution of a single machine instruction), usually needs to be performed many times. For

this reason, the component programs are designed to repeat and to run concurrently, so that they can simultaneously be engaged in execution of different occurrences of P. This increases component utilization and rate of execution of successive P's, called throughput.

A software pipeline may be used in the same manner, but more often it is the result of combining several independent programs rather than decomposing one. If a program's input and output are allowed to be directed through arbitrary pipes (chosen before each execution of the program) then processes which have no knowledge of each other may be flexibly interconnected to form output filters, input preprocessors, message buffers, etc.

pipeline processing The use of pipes to solve a problem.

distributed processing The loose connection of several independent processors, each with its own storage, in order to allow tasks to be split off for local execution and to allow them to draw on the combined computational power of several processors (e.g., in the operation of a large multi-user computer center). It is a currently popular alternative to centralized processing facilities that require larger, more powerful, and more expensive super-computers. It is a subset of multiprocessing involving little communication and synchronization of the processors.

associative processing The processing of data which is accessed by specifying content of words in memory (or parts of words) instead of locations.

associative processor A processor (usually involving a large amount of concurrent activity) specifically designed to do associative processing.

process synchronization The control of the order in which processes advance relative to each other, in order to insure that interprocess communication and resource utilization occur repeatably and do not depend on what other processes may be alive in the system.

mutual exclusion The prohibition against two or more processes using a resource simultaneously. When one process begins to use the resource, all others requiring it must wait until it is released.

Synonym: lock-out

critical region A "critical region" is a portion of a program requiring mutually exclusive use of some resource in order to produce a repeatable result. Each critical region is defined with respect to the use of one resource. No two processes should be in regions of their programs which are critical with respect to the same resource at the same time.

deadlock When each of a chain of processes is dormant, waiting for a condition which can only be caused by the next process in the chain, and the last is similarly waiting for the first, then the processes are "deadlocked." Also, any process waiting for a deadlocked process is itself deadlocked. None of them will progress without outside intervention.

indefinite overtaking When several processes are waiting for exclusive use of a resource, and one or more may never be allowed to continue because other processes always obtain the resource first, then the waiting processes are said to be "indefinitely overtaken."

concurrent processing The use of concurrent processes.

Synonym: parallel processing

co-routine Two or more cooperating processes are "co-routines" if their periods of progress are mutually exclusive (i.e. only one advances at a time, with the others dormant).

subroutine A subprocess, P, of a process, Q, is a "subroutine" of Q if they are co-routines (i.e. Q becomes dormant when P is initiated and does not resume until P terminates).

Example:

If the executions of different procedures in a FORTRAN program are viewed as different processes, then, during a subroutine call from Subroutine A to Subroutine B, we say that the process executing B is a subroutine of the process executing A.

descendent — A process, Q, is a "descendent" of a process, P, if Q is initiated by P or by any other descendent of P.

Synonym: child process

task — A "task" is a process together with all descendents of it. Note that a task contains other tasks, unless it is a single process.

sub-process — A "sub-process" of a process, P, is a descendent of P whose lifetime is surrounded by P's (i.e. it starts after P and terminates before P).

Example:

In the example given under asynchronous processes, B is a sub-process (but not a subroutine) of A.

suspend — A process is said to "pause" or to "suspend" activity when it stops advancing and becomes dormant. Similarly, one process "suspends" or "blocks" another when it causes the latter process to become dormant.

Synonym: block, pause

initiate — A new process is "initiated" when a processor begins to execute a program.

Synonym: start, begin, birth

terminate — A process "terminates" when its processor finishes executing the program from which it was initiated.

Synonym: stop, finish, die

SECTION 7

THE PROPOSED EXTENSIONS

7.1. PRIMITIVE CONSTRUCTS

The following constructs form the basis for the entire set of concurrent processing operations. All other constructs can be described precisely (see examples) by showing the basic constructs to which they are equivalent. Therefore, it is important to understand the meaning of these constructs before proceeding to the higher level ones.

7.1.1. Start

Syntax:

```
start:statement ::= START process:label;  statement
process:label ::= process:name
                  /nil
process:name ::= name
```

Semantics:

Execution of the "statement" is begun by a new process concurrent with the process which executed the "start:statement". The original process continues execution immediately at the statement following the

"start:statement". If the "process:label" is not nil, the authority and priority (see PROCESS VARIABLE) of the new process are taken from the declaration of "process:name". If it is nil or priority is not specified in the declaration, the new process will have equal priority to the one executing the "start:statement". Similarly, if "process:label" is nil, or authority is not specified in the declaration of "process:name", the new process will have lower authority than the one executing the "start:statement". The identification of the new process is put in the process variable "process:name" (see PROCESS VARIABLE) and may be used to suspend, resume, or halt the new process. The new process terminates when it finishes execution of the "statement", is stopped with a STOP statement, or attempts to branch to a location outside of the "statement". It may, however, execute procedure calls. Any variables or procedures which were available to the original process are also available to the new process. Any dynamically allocated variable will remain allocated and available until all processes having access to it are terminated. Implementation note: the above features of variable usage by concurrent processes probably require that, for any procedure whose execution can cause the initiation of new processes, its local variables cannot be allocated on a stack, since, in general, they will not be deallocated in stack order. A more general dynamic storage allocation method must be used.

Examples:

```
START  CHILD;                         START;
  BEGIN                                  X:=F(Y);
    P1;                                  :
    P2;                                  :
  END;
  :
  :
```

7.1.2. Stop

Syntax:

```
stop:statement ::= STOP process:specifier;
process:specifier ::= process:variable
                      /nil
```

Semantics:

A "stop:statement" causes execution of the process identified by the "process:specifier" to terminate and the process is permanently destroyed. If the "process:specifier" is nil, the process which executes the "stop:statement" terminates. A "stop:statement" can appear at any procedure nesting level. The "actual:output:params" of any procedure whose call is still active in the process being terminated will not be set. The process executing a "stop:statement" must have equal or greater

authority than the process being terminated, or this statement has no action.

7.1.3. Lock

Syntax:

lock:statement ::= LOCK variable:list

Semantics:

The variables in the "variable:list" are locked so that no other concurrent processes may access them, until they are unlocked using the UNLOCK statement. Any other process which attempts to read, set, or LOCK any of them will wait in a queue for access to the variable. All variables locked to a process are automatically unlocked when it halts or is aborted. If any of the variables in the "variable:list" was already locked by this process, an additional lock is placed on it. It will not be released to other processes until as many UNLOCKs as LOCKs have been performed on it by its owner process. When a "process:variable" is locked, the process identified by it is "frozen" (not suspended) in the state it was in at the time of being locked. Thus, if it was dormant, it cannot be resumed; if it was advancing, it cannot be suspended; if it did not exist (the "process:variable" did not correspond to any currently alive process), the "process: variable" cannot be used in a START statement to initiate a new process. If any of these actions are

attempted while the "process:variable" is locked by a different process than the one issuing the action, the issuing process waits in a queue for access to the process. When the "process:variable" is unlocked, the attempted action is completed normally.

Examples:

```
LOCK A, B;                          LOCK X[I+5], Y;
```

7.1.4. Unlock

Syntax:

```
unlock:statement ::= UNLOCK variable:list
```

Semantics:

One lock is removed from each variable in the "variable:list" which is currently held (LOCKed) by the process performing the UNLOCK. Then any variable having zero locks on it is released, so that other concurrent processes may access it. All variables locked by a process are unlocked implicitly when the process terminates.

7.1.5. Pause

Syntax:

```
pause:statement ::= PAUSE process:specifier;
```

Semantics:

The process identified by the "process:specifier" (or the process executing the "pause:statement", if the "process:specifier" is nil) is suspended and will not advance to its next instruction until it is resumed using the RESUME statement. The process executing the "pause: statement" must have equal or greater authority than the process identified by the "process:specifier" or the statement has no action.

7.1.6. Resume

Syntax:

resume:statement ::= RESUME process:variable;

Semantics:

The suspended process (see PAUSE) identified by the "process: variable" is resumed. It continues execution from the point where it was suspended. If it was not suspended, or if the process executing the "resume:statement" does not have equal or greater authority than the process to be resumed, the statement has no action.

7.1.7. Await

Syntax:

```
await:statement ::= AWAIT logical:formula;
```

Semantics:

The process executing the "await:statement" is suspended until the "logical:formula" becomes true, even if only momentarily. Of course, the "logical:formula" is not guaranteed to remain true when the process resumes. Execution continues with the following statement. Any functions referenced in the "logical:formula" must be "pure" functions; that is, their return value depends only on the value of their input:parameters. Any variable in the logical:formula must be declared SHARED if a change in its value is expected to cause resumption of the process.

7.1.8. Shared Variables

Syntax:

```
item:declaration ::= shared:specifier  followed by the standard
                                        JOVIAL (J/73) item:declaration

table:declaration ::= shared:specifier  followed by the standard
                                        JOVIAL (J/73) table:declaration

shared:specifier ::= SHARED
                     /nil
```

Semantics:

Using the word "SHARED" in the declaration of a variable declares that the variable may become available to more than one process and that mutually exclusive use of the variable is to be enforced both automatically and by the explicit use of the LOCK and UNLOCK statements. Whenever a SHARED variable is used in an executable statement, the variable will be automatically locked immediately before each reference and unlocked immediately afterwards. This means that a shared variable which is currently locked by another process cannot be accessed until unlocked, even if the process attempting the access has not explicitly locked it.

Variables which are not declared SHARED cannot be locked using the LOCK statement. Both the LOCK and UNLOCK statements have no effect on such variables. Also, no automatic enforcement of mutually exclusive access is performed.

Any variable which appears in the logical:formula of an AWAIT statement must be declared SHARED if a change in its value is expected to cause the resumption of the process executing the AWAIT. If any such variable is not declared SHARED, the process is not guaranteed to resume.

An actual:parameter in a procedure:call must be declared SHARED if and only if the corresponding formal:parameter is decla[illegible] SHARED.

7.1.9. Process Variable

Syntax:

```
item:description ::=  as in JOVIAL J/73 plus:
                      / process:description

process:description ::= P priority:specifier
                          authority:specifier

priority:specifier ::= number
                     / nil

authority:specifier ::= , number
                      / nil

formula ::=  as in JOVIAL J/73 plus:
           / process:variable

process:variable ::= process:name
```

Semantics:

This kind of item declaration declares a variable which can hold a process identification. When a process is initiated using the START statement with a "process:variable", some unique identification (probably an integer) of the new process is stored in the "process:variable" (see

START). Then the "process:variable" can be used to refer uniquely to that process. The "priority:specifier" controls how quickly the process will pass through queues while waiting for resources, and possibly the rate at which it will execute instructions. The "authority:specifier" defines which processes have permission to control the execution of one another. Only a process of greater or equal authority may abort, suspend, or resume another. When one "process:variable" is assigned to another, they identify the same process until either one is changed.

7.1.10. Alive (standard function)

Syntax: ALIVE (process:variable);

Semantics:

This logical function returns true if the process identified by the "process:variable" currently exists and false otherwise.

7.1.11. Dormant (standard function)

Syntax: DORMANT (process:variable);

Semantics:

This logical function returns true if the process identified by the "process:variable" exists and is dormant (suspended) and false otherwise.

7.1.12. Priority (standard function)

Syntax: PRI (process:variable);

Semantics:

This function returns an integer equal to the priority of the process identified by the "process:variable".

7.1.13. Authority (standard function)

Syntax: AUTH (process:variable);

Semantics:

This function returns an integer equal to the authority of the process identified by the "process:variable".

7.1.14. Free (standard function)

Syntax: FREE (variable);

Semantics:

This logical function returns true if the "variable" is not locked by any process, including the one issuing the call.

7.1.15. Available (standard function)

Syntax: AVAIL (variable);

Semantics:

This logical function returns true if the "variable" is either free, or is locked by the process issuing the call.

7.1.16. Clock (standard function)

Syntax: CLOCK ();

Semantics:

This function returns a real number equal to the elapsed time, in seconds, since some fixed past time.

7.2. HIGH LEVEL CONSTRUCTS

All of the following structures could be implemented using the preceding constructs. (This may not, however, be the most efficient method.) The examples attempt to be general enough to show this relationship without any ambiguity. They are NOT the suggested way of implementing the constructs, but serve only to clarify the description of semantics.

7.2.1. Concurrent Compound Statement

Syntax:

```
statement ::= label* simple:statement
            / label* BEGIN
              compound:statement+ label* END
            / label* COBEGIN
              compound:statement+ label* END
```

Semantics:

The first two alternatives in the above rule are defined as in standard JOVIAL (J/73).

When the third alternate construct is used, the programmer is declaring that the statements in the "compound:statement" may be executed in any order or simultaneously. If any variable is used by more than one of the statements, it should not have its value changed by any of them, except while locked (see LOCK). Failure to adhere to this restriction can prevent the program from producing repeatable results.

Example:

```
COBEGIN                                 START P1; S1;
     S1; S2; ... Sn;                         .
COEND                     means:             .
                                             .
                                        START Pn; Sn;
                                        L1:  IF ALIVE(P1); GOTO L1;
                                             .
                                             .
                                             .
                                        Ln:  IF ALIVE(Pn); GOTO Ln;
```

7.2.2. Conditional Critical Region

Syntax:

```
critical:region ::= acquire:list  entry:condition;
                    statement
acquire:list ::= ACQUIRE variable:list
                 / nil
entry:condition ::= WHEN logical:formula
                    / nil
```

Semantics:

Either the "acquire:list" or the "entry:condition" must be specified. When the "logical:formula" becomes true, even momentarily,

an attempt is made to lock all of the variables in the "variable:list". If the "logical:formula" is still true when the LOCK succeeds, then the "statement" is executed, after which the variables are unlocked. If the "logical:formula" is not still true, however, the variables are unlocked and the preceding actions are repeated. The variables which make up the "logical:formula" are not locked unless they also appear in the "acquire:list", except that if the "acquire:list" is nil, every variable in the formula is locked, which guarantees that it will remain true during the execution of the "statement". A nil "entry:condition" has the same effect as "WHEN TRUE"; that is, the critical region is entered unconditionally.

Example:

```
ACQUIRE A,B                               LOOP:  SUCCESS = FALSE;
  WHEN X Y+B;                                       AWAIT X Y+B;
   statement                                        LOCK A, B, X, Y;
                                 means:             IF X Y+B;
                                                      BEGIN  UNLOCK X, Y;
                                                        statement
                                                        SUCCESS = TRUE;
                                                      END
                                                    ELSE;  UNLOCK X, Y;
                                                    UNLOCK A, B;
                                                IF NOT SUCCESS; GOTO LOOP;
```

7.2.3. Semaphore Operations (standard functions)

Syntax: WAIT (integer:variable);

SIGNAL (integer:variable);

Semantics:

These standard functions are defined as follows:

```
PROC WAIT (:SEMVAR);
 BEGIN
   ITEM SEMVAR S;
   ACQUIRE SEMVAR
     WHEN SEMVAR 0;
     SEMVAR=SEMVAR-1;
 END;
```

```
PROC SIGNAL (:SEMVAR);
 BEGIN
   ITEM SEMVAR S;
   ACQUIRE SEMVAR;
     SEMVAR=SEMVAR+1;
 END;
```

7.2.4. Concurrent for Loop

Syntax:

```
for:clause ::= FOR concurrency:specifier
                   control:variable : control:clause
concurrency:specifier ::= ALL
                        / nil
```

Semantics:

If the "concurrency:specifier" is not nil, it is assumed that all of the iterations of the FOR loop can be done in any order or concurrently. The calculation of replacement or increment values for the control variable will be done sequentially, and a separate process will be initiated to perform each iteration. However, each of these processes will be given a separate, local copy of the control variable, so that when it is replaced with its next value (which happens just after initiating the process), the local control variable remains unaffected.

Example:

```
FOR ALL I: 1 THEN NEW(I)                "PR" is a TABLE of
  WHILE I≠X;                                 process variables:
  USE(I);                    means:     T = 0;
                                        FOR I: 1 THEN NEW(I)
                                          WHILE I≠X;
                                        BEGIN ITEM L
                                          T = T + 1;
                                          LCLI = I;
                                          START PR[T]; USE(LCLI);
                                        END;
                                        FOR T:T BY -1 WHILE T =1;
                                          AWAIT NOT ALIVE(PR[T]);
```

7.2.5. Queue

Syntax:

```
queue:declaration ::= ordinary:queue:declaration
                    / specified:queue:declaration
ordinary:queue:declaration ::= shared:specifier QUEUE queue:name
                               queue:organization:part
                               ordinary:entry:specifier
queue:organization:part ::= allocation:specifier
                            queue:size:specifier
                            structure:specifier
                            packing:specifier
queue:size:specifier ::= [integer]
specified:queue:declaration ::= shared:specifier QUEUE queue:organi-
                                                  zation part
                                words:per:entry
                                specified:entry:specifier
queue:name ::= name
queue:ref:function ::= OLDEST (queue:specifier)
                     / NEWEST (queue:specifier)
queue:specifier ::= queue:name / queue:item:name
variable ::=  same as in standard JOVIAL J/73 plus:
           / queue:ref:function
```

Semantics:

A QUEUE variable is an ordered, variable length sequence of entries (which are just like TABLE entries), with the restriction that only the most recent entry in the queue and the oldest entry in the queue can be accessed (see OLDEST, NEWEST), and that only the oldest entry can be removed from the queue (see DROP, EXTEND). Also, the number of entries currently in the queue may be discovered by using standard functions (see LENGTH, FULL, EMPTY). The "queue:size:specifier" declares the maximum number of entries allowed in the queue at the same time.

Note that a QUEUE declaration is much like a table declaration, except that access to the queue is substantially different. Because of the automatic enforcement of first-in/first-out order, queues are especially useful for sending messages from one (or several) process(es) to another (or several other) concurrent process(es). This kind of usage is commonly known as "pipelining".

Examples:

```
QUEUE Q1 [100] S;
```

```
QUEUE Q2 [50]
  BEGIN
    ITEM FLD1 U;
    ITEM FLD2 S;
  END;
```

7.2.6. Oldest, Newest

Syntax: OLDEST (queue:specifier);

NEWEST (queue:specifier);

Semantics:

These functions provide the only means of accessing the information in QUEUE's. They may appear in formulas or in the left side of assignment statements, since they are defined as variables. The queue identified by the "queue:specifier" must not be empty (LENGTH = 0) when either of these functions is used, or an error results. The functions are used on the left side of an assignment statement to define the value of the oldest or newest entry (or part of an entry if using a "queue:item:name") in the queue identified by the "queue:specifier". When used in formulas, they return the value of the oldest or newest entry (or part of an entry if using a "queue:item:name") in the queue. Often, it will be more convenient to use the standard functions INSERT and REMOVE instead of OLDEST, NEWEST, DROP, and EXTEND.

7.2.7. Drop, Extend

Syntax: DROP (queue:name);

EXTEND (queue:name);

Semantics:

These functions provide the only means of removing and adding entries to a QUEUE. Use of the DROP function removes the oldest entry from "queue:name" and the entry is effectively destroyed. Subsequent uses of the OLDEST function will no longer refer to this entry, but to what was previously the next-to-oldest entry. The LENGTH function will return a value one less than before. If, however, there are no entries in "queue:name" when a DROP is attempted, an error results.

The EXTEND function is used to add a new entry to "queue:name". Subsequent uses of the NEWEST function will now refer to the new entry. The value of the queue entry is undefined until it is established using the NEWEST function. The LENGTH function will return a value one greater than before. If, however, "queue:name" is full when an EXTEND is attempted (number of entries equals the integer specified by the "queue:size: specifier" in the declaration of "queue:name"), an error results.

7.2.8. Insert, Remove (standard functions)

Syntax: INSERT (formula : queue:name);

REMOVE : variable, queue:name);

Semantics:

The "formula" and "variable" must be of the same type as the entries of "queue:name". These standard functions are defined loosely as follows:

```
PROC INSERT (VAL:Q);
 BEGIN
  ITEM VAL ...;
  QUEUE Q  ...;
  ACQUIRE Q
   WHEN NOT FULL(Q);
   BEGIN
    EXTEND (Q);
    NEWEST(Q) = VAL;
   END
 END;
```

```
PROC REMOVE (:VAR, Q);
 BEGIN
  ITEM VAR ...;
  QUEUE Q  ...;
  ACQUIRE Q
   WHEN NOT EMPTY(Q);
   BEGIN
    VAR = OLDEST(Q);
    DROP (Q);
   END
 END;
```

7.2.9. Full, Empty (standard functions)

Syntax: FULL (queue:name);

EMPTY (queue:name);

Semantics:

The logical function FULL returns true when the number of entries in "queue:name" is equal to the integer used in the "queue:size:specifier" in the declaration of "queue:name", and false otherwise. The logical function EMPTY returns true when there are no entries in "queue:name", and false otherwise. All queues are initially empty.

7.2.10. Length (standard function)

Syntax: LENGTH (queue:name);

Semantics:

This function returns an integer equal to the number of entries currently in the queue specified by "queue:name". All queues are initially of zero length.

SECTION 8

CONCLUSIONS AND RECOMMENDATIONS

In the development of extensions to JOVIAL for concurrent processing, we have attempted to delve into one of the newest, least understood fields of computer science, with the goal of making concurrent processing into a useful tool for designing complex computer software, rather than a liability which can introduce non-determinacies and confusion even into simple programs.

This project has drawn heavily on the work of others, notably Brinch Hansen, Dijkstra, and Hoare, in addition to making several new proposals for concurrent language extensions. The set of definitions of concurrent processing terms proposed here is, we believe, the most complete and precise such list yet completed, although it may still fall somewhat short of the ideal of complete freedom from ambiguity. The definitions were successfully used in discussions concerning the design of the language extensions, and proved to facilitate the communications of ideas by removing the necessity for lengthy explanations of terminology.

The proposed language extensions should prove to be reliable and powerful when they are actually implemented and tested empirically. Since the real test of the quality of a language is its acceptance by and usefulness to its users, implementation and test of the language extensions will provide a source of suggestions for revision and possible expansion of the design presented in this report. We therefore recommend the implementation of the proposed language extensions for the purpose of further analysis and testing.

Development and study of the mathematical model for concurrent processing (Section 5 of this report) has been fruitful in improving the quality of both the definitions and the language extensions, by simplifying and clarifying essential concepts. Further development of the model will be helpful for actual implementation of the language extensions. In addition, the model may be used to describe complex parallel hardware in a precise and understandable manner, and thus may assist in the design of a concurrent hardware description language. For these reasons, further study and development of the model is proposed.

BIBLIOGRAPHY

Anderson, J.P., "Program Structures for Parallel Processing," Communications of the ACM, Vol 8, No 12, pp 786-788, 1965.

Baer, J.L., "A Survey of Some Theoretical Aspects of Multiprocessing," ACM Computing Surveys, March 1973.

Basili, Victor R. and John C. Knight, "A Language Design for Vector Machines," SIGPLAN Notices, pp 39-43, March 1975.

Brinch Hansen, Per, "Concurrent Programming Concepts," ACM Computing Surveys, pp 223-245, Dec. 1973.

------"The Nucleus of a Multiprogramming Concepts," ACM Computing Sur- ACM, Vol 13, No 4, pp 238-250, 1970.

------Operating System Principles, Prentice-Hall, pp 55-131, 1973.

------"The Purpose of Concurrent PASCAL," SIGPLAN Notices, pp 305-309, June 1975.

------"Standard Multiprogramming," Communications of the ACM, Vol 15, No 7, pp 574, 1972.

Coffman, E.G., M.J. Elphick, and A. Shoshani, "System Deadlocks," Computing Surveys, Vol 3, No 3, pp 67-78, 1971.

Dijkstra, E.W., Cooperating Sequential Processes, Reprinted by Academic Press, New York 1968.

------"Solution of a Problem in Concurrent Programming Control," Communications of the ACM, Vol 8, No 9, pp 569, 1965.

------"The Structure of "THE" Multiprogramming System," Communications of the ACM, Vol 11, No 5, pp 341-346.

Erickson, David B., "Array Processing on an Array Processor," SIGPLAN Notices, pp 17-24, March 1975.

Genuys, F. (edited by) Programming Languages, Academic Press, New York, 1968.

Gosen, J.A., "Explicit Parallel Processor Description and Control in Programs for Multi and Uniprocessor computers," AFIPS 1966 Fall Joint Computer Conference, Spartan Books, pp 651-660, 1966.

Habermann, N.A., "Synchronization of Communicating Processes," Communications of the ACM, Vol 15, No 3, pp 171-176, 1972.

Hermes, H., Enumerability, Decidability, Computability, Springer-Verlag, New York, 1969.

Hoare, C.A.R., "Towards A Theory of Parallel Programming," International Seminal on Op. Sys. Techniques, 1971.

------"Monitors: An Operating System Structuring Concept," Communications of the ACM, Vol 17, No 10, pp 549, 1974.

Horning, J.J. and B. Randell, "Process Structuring," Computing Surveys, Vol 5, No 1, pp 5-30, 1973.

Kane, J. and S. Yau, "Concurrent Software Fault Detection," IEEE Transactions on Software Engineering, SE-1, No 1, pp 87-99, 1975.

BIBLIOGRAPHY (Continued)

Lamport, L., "The Parallel Execution of DO Loops," Communications of the ACM, pp 83-93, Feb 1974.

------"On Programming Parallel Computers," SIGPLAN Notices, pp 25-33, March 1975.

Lipton, Richard J., "Reduction: A Method of Proving Properties of Parallel Programs," Communications of the ACM, pp 717-721, Dec 1975.

Loren, Harold, Parallelism in Hardware and Software: Real and Apparent Concurrency, Prentice-Hall, 1972.

North, N., "A Note on Program Structures for Parallel Processing," Communications of the ACM, Vol 9, No 5, pp 320-321, 1966.

Opler, A., "Procedure Oriented Language Statements to Facilitate Parallel Programming," Communications of the ACM, Vol 8, No 5, pp 306-307, 1965.

Presser, L., "Multiprogramming Coordination," ACM Computing Surveys, pp 21-45, March 1975.

Sintzoff, M. and A. Van Lamsweerde, "Constructing Correct and Efficient Concurrent Programs," SIGPLAN Notices, pp 319-327, June 1975.

Tesler, L.G., "A Language Design for Concurrent Processes," AFIPS 1968 Spring International Comp. Conference, AFIPS Press, pp 402-408, 1968.

Thurber, K.J. and L.D. Wald, "Associative and Parallel Processors," ACM Computing Surveys, pp 215-255, Dec 1975.

BIBLIOGRAPHY (Continued)

Yasuhara, A., Recursive Function Theory and Logic, Academic Press, New York, 1971.