Missouri S&T Library and Learning Resources

Scholars' Mine

Doctoral Dissertations | Student Theses and Dissertations

Summer 1986

A semantic basis for parallel algorithm design

Roger E. Eggen

Follow this and additional works at: https://scholarsmine.mst.edu/doctoral_dissertations

 Part of the Computer Sciences Commons

Department: Computer Science

Recommended Citation

Eggen, Roger E., "A semantic basis for parallel algorithm design" (1986). *Doctoral Dissertations*. 612.
https://scholarsmine.mst.edu/doctoral_dissertations/612

This thesis is brought to you by Scholars' Mine, a service of the Missouri S&T Library and Learning Resources. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

A SEMANTIC BASIS FOR PARALLEL ALGORITHM DESIGN

BY

ROGER EGGEN, 1947-

A DISSERTATION

Presented to the Faculty of the Graduate School of the

UNIVERSITY OF MISSOURI-ROLLA

In Partial Fulfillment of the Requirements for the Degree

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

1986

John R Metzner
Advisor

David D Stygall

Thomas T Sager

Daniel C. St. Clair

Orin K Oaster

ABSTRACT

As computing demands increase, emphasis is being placed on parallel architectures. To efficiently use parallel machines, software must be designed to take advantage of these machines. This research concentrates on an abstraction of algorithm design to permit the expression of parallel programs. The abstraction emphasizes thought about algorithms at a high level as opposed to algorithm implementation at a statement level. A model based on data flow allows algorithm expression using flow diagrams. The model specifies operating system requirements that support parallel programming at a module level. Paths are used to carry data between modules. Data enter modules through ports. Module activation is triggered by the satisfaction of data availability conditions. Continual module presence within the system, dynamic activation criteria, and a high level of programming distinguishes this model from other parallel programming systems.

ACKNOWLEDGEMENT

The author wishes to give a special thanks to Dr. John Metzner for his stimulating discussions and guidance in this research. His role as major advisor provided invaluable assistance during the development and writing of this dissertation. These results would not have been possible without his foresight and intuitive observations.

Appreciation is also extended to Dr. Arlan DeKock for his assistance in procuring financial support from AMOCO oil company and the Computer Science Department.

The efforts of Dr. Thomas Sager, Dr. Paul Stigall, Dr. John Hamblen, Dr. Orrin Crosser and Dr. Daniel St.Clair, as members of the advisory committee, are gratefully appreciated.

Thanks are also extended to my family for their support during this work.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

I. INTRODUCTION

A. MOTIVATION - WHY DATA FLOW?

1. A Need For Parallelism. To meet increased computing demands, both software and hardware improvements are required. Computing in the areas of meteorology, cryptography, image processing, and sonar and radar surveillance, using real time processing, requires computation speeds surpassing even the current super computers [24]. Computing speeds have been increasing due primarily to improvements in hardware design. These improvements are approaching an upper limit as traditional hardware advances are limited by the physical characteristics of the components. Computers must support a great deal of concurrency to achieve a significant increase in performance. Data flow architectures offer a solution to the problem of efficiently exploiting concurrency of computation on a large scale [33]. Appropriate software must be developed to use a data flow architectures effectively. Data flow programs and program design offer a solution.

2. What Is Data Flow? Program development using data flow is conceptually different than programming with traditional procedural languages. There are no program counter sequencing instructions or a global memory. Side

effects are eliminated to allow concurrent instruction execution. The sequence of execution in data flow depends only on data availability.

Directed graphs represent data flow programs. Actors are nodes which contain operators. Links are paths in the graph that carry the output of one actor to the input of another actor as described by the directed graph. Data presence in all consuming links causes an actor to "fire" which consumes the data from those links and produces results on the supply links.

Figure 1 shows a data flow graph which corresponds to the expression: (B**2 - 4*A*C) / 2*A. As shown, traditional data flow actors contain one instruction per actor. These instructions are generally at the level of an add or multiply. The example shows the typical level of parallelism exploited by these graphs. An instruction level of parallelism or fine parallelism is represented by Figure 1. Each actor consumes data values and produces results. Concurrency, in this example, is possible since the first three actors fire simultaneously. By using a data flow graph, it is possible to expose all of the parallelism in an algorithm and highly concurrent computation becomes a natural consequence of the data flow concept [32].

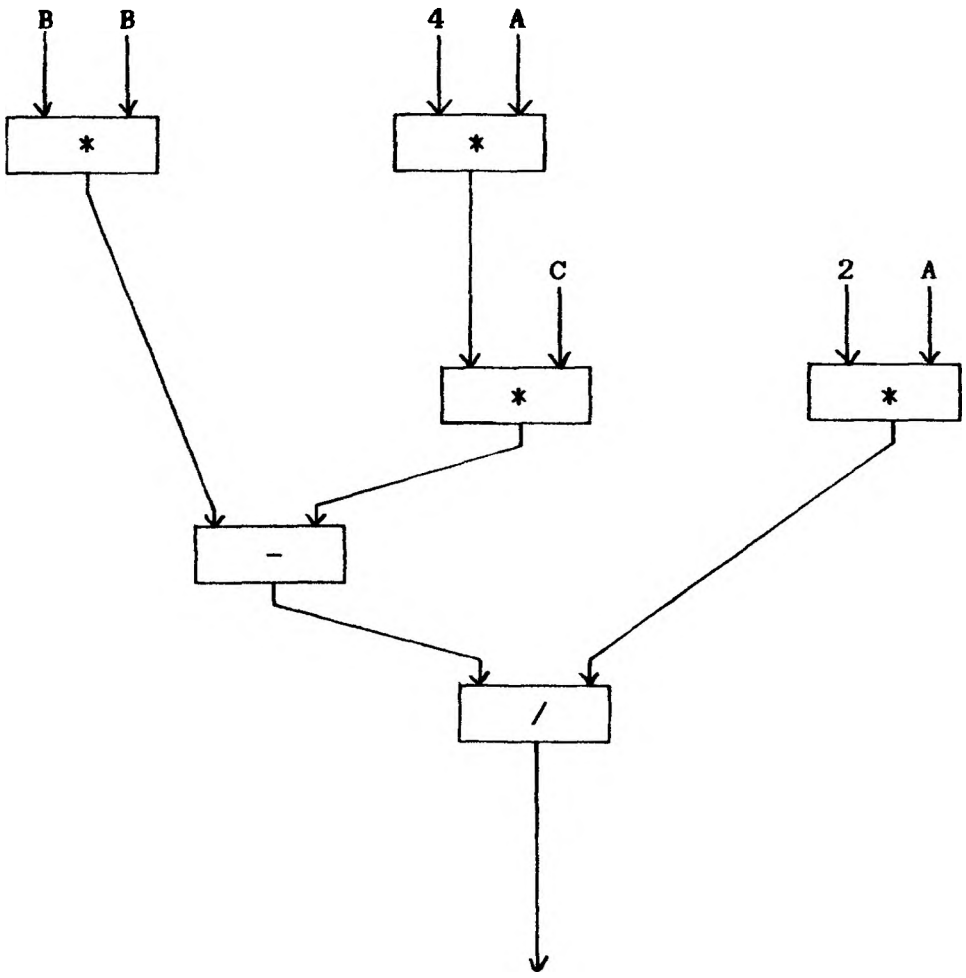

FIGURE 1
SAMPLE DATA FLOW GRAPH

3. A Second Application Of Data Flow. The time and effort involved in software design and development has proven to be significant. As hardware prices continue to fall, there is growing pressure to increase productivity of application programmers. Many companies are trying to find ways to substantially improve productivity [43]. Efficiency is enchanced by improved software design techniques. Such concepts as structured programming and top-down design have improved programmer productivity. Algorithm design is further enhanced by using pseudo-coding and reusable routines. A routine can be developed to perform a task. If the routine is general enough, it can be used in a wide variety of programs. Some operating systems provide a SORT routine that can be used in a variety of applications.

The data flow concept can be extended to include program design. Top-down design as well as modular construction is supported through data flow. Traditional data flow can be extended to include many instructions in an actor. An actor consisting of many instructions is called a module and data flows between modules in paths. Independently developed modules interact only through data exchanges. The data flow graph is a convenient way to show all data interaction between modules. Benefits from using data flow are:

- no global storage
- visible module dependencies

- reusable modules
- reduced complexity
- natural, consistent application view
- easier application development [43].

Modules are developed independently without knowledge of other modules supplying input or receiving output. The modules can often be placed in a variety of programs. The program design is specified by a directed graph representing flows of data between modules. The modules initially represent large tasks which are subdivided into smaller modules as the development proceeds. The data flow methods are used during the refinement process. Data flow usage in the design of programs is a natural development as data flows from module to module like the flow of paper from desk to desk in a big office environment.

4. Need For Parallel Algorithm Development. Since parallel architectures are now a reality, [5, 44, 45, 49] parallel algorithms are necessary. To capture all of the parallelism possible in an algorithm, parallel design and development from the initial design phase are required. Since data flow is a radically different concept, new and different thoughts about algorithm design are encouraged. Just as algorithm design and implementation were initially difficult, some effort must be given to the design and development of algorithms which are conceptually different. This effort realizes benefits in computing performance and time of program development.

5. Content Of Dissertation. This paper is written to: 1) offer a medium for the development and expression of parallel algorithms and 2) promote a correspondingly different view of computing systems. Data flow is the vehicle for the expression of these algorithms. Operating system features are described to provide support for the programs used to represent algorithms. The features of the operating system are developed only to the extent of establishing the environment in which data flow algorithms are to operate.

The second goal of this paper is to promote a different view of computing systems. Traditionally, programs in the form of jobs or job steps are entered into a system, executed, and removed from the system. Consider a computing system where programs are always resident. The programs remain idle until data presence "activates" the program. The program remains active as long as data processing is required. Upon completion of its processing, the program becomes dormant or "goes to sleep". The program does not leave the system, but waits for further processing. In this environment, there is no human-given command which starts programs, but rather the presence of data, indicating required processing, determines program activation. Data flow at a "higher" level distinguishes this approach to concurrent programming from others. This concept will be emphasized throughout the paper.

Chapter 1 shows why data flow is chosen. This chapter also gives an introduction to the vehicle used in the expression of parallel algorithms, design decisions, and a review of the goals and qualities of the vehicle. Chapter 2 describes the basic components that are at the disposal of software designer. Chapter 3 explains the support required of an operating system to create the programming environment. Chapter 4 discusses programming implications such as: nesting, replication, file handling, common modules and examples. Chapter 5 presents conclusions and ideas for further research.

B. CURRENT RESEARCH.

1. Introduction. Since parallelism is accepted as a promising solution to increased demands for computing, a variety of research is taking place. The research spans a very broad spectrum covering the development of parallel architectures to software design processes. Two approaches to the development of parallel architectures are being followed: an extension of the von Neumann model and new designs patterned after data flow. Some parallel programming languages are being developed which extend procedural languages like Pascal and PL/I, while new languages like LZ are based on graphical representations [12]. Programs written in procedural languages are manipulated by translators which attempt to recognize implicit and explicit parallelism. The translators produce a form of the program suitable for execution on large-scale, parallel processing systems while graph-based languages can be executed directly on an appropriate parallel architecture by directly interpreting the graph.

Parallelism is also being used during the software system design phase. If the system defined can be divided into a number of independent tasks, many software engineers can be assigned to develop the system. This practice is commonly employed in the development of a large system. Parallelism is sought during the initial design phase and retained through system implementation.

Concurrent system development is possible by using a parallel approach [43]. Data flow is a natural model used during the design and implementation phases.

2. Parallel Architectures. Extensions to the von Neumamn computing model fall into five categories: systolic arrays, associative processors, tree machines, array processors, and multiple CPU computers. Systolic arrays are a collection of synchronized special purpose processors with a fixed interconnection network. This model is not frequently used. An associative processor is a special purpose processor built around an associatve memory which allows the simultaneous searching of the whole memory for some specified contents. Again, this model is not frequently used. A tree-structured searching machine has capabilities falling somewhere between an associative processor and an array processor. The array processor has a set of synchronized arithmetic units capable of performing the same operation on different data. This processing system is called SIMD. SIMD is a single instruction executing on multiple data streams. The arithmetic units execute the instruction on the data paths in parallel. Multiple-CPU computers (MIMD) consist of a number of fully programmable processors, each capable of executing its own program. MIMD (Multiple Instruction Multiple Data) means that multiple processing units with individual instructions execute on different data sets. The processors communicate via shared memory. MIMD models

may differ in two respects: the processors may be synchronous or asynchronous and the number of processors may be fixed or unbounded [36].

Data flow architectures are so new that categorization is impossible. Arvind and others at MIT are developing an architecture designed to execute data flow programs with a tagged token. Their paper describes how data flow programs are mapped onto the hardware [5]. The Manchester data flow machine, being developed at the University of Manchester, functions on a labeled data flow model. The units of the machine are connected in a ring and consist of a processing unit, token queue, matching unit and node store. The tokens flow around the ring and contain three information fields and a control field. The information fields of the tokens carry data, the label, and the token's destination node address. The tokens are matched by using the control field and passed to the processing unit for execution, which produces a new set of tokens [49]. Other researchers are developing architectures designed to execute parallel programs and studying different architecture designs to estimate computing efficiencies [6,44].

Data flow architectures are characterized by the lack of global storage. Processors communicate via a message passing network rather than through shared global memory. There is no "control" sequencing instructions, but the presence of data causes instruction execution.

3. Language Development. Research into languages covers a wide spectrum from extensions to existing languages, such as FORTRAN and Algol, to new languages, such as the single assignment languages for data flow machines. These languages have the following attributes in common: they are all textually based languages and attempt to exploit parallelism near the machine instruction level. DFL, a data flow language which is a Pascal-like parallel language, provides a typical example [33]. A program written in DFL represents a data flow graph. Concurrency is almost always implicit, in contrast with Concurrent Pascal, where all concurrencies are specified explicitly by the programmer.

ID, CAJOLE, LAPSE, VAL and DDM1 are data flow languages being developed at the University of California (Irvine), the University of London, Manchester University, MIT and University of Utah, respectively. Of these languages, DDM1 is the only graph based language. They are functional in nature and are translated into data flow graphs for execution [1]. LZ [12], developed at Oxford Polytechnic, and PROGRAPH [27], developed at the University of Ottawa and the Technical University of Nova Scotia, are graph-based languages, which do not need to be converted to a data flow graph before execution. These languages allow parallel activity on a data-driven abstract machine.

4. Data Flow Models. In order for data flow to be an effective model of concurrent program execution, an appropriate representation of data flow is required. The representation should be simple so that algorithm development is not complicated by the representation and rich enough to allow real problem solutions. Petri nets, e-nets, colored petri nets and time-extended petri nets have been used for this purpose. Petri nets are composed of places, transitions, functions defining sources of input, and functions defining destinations of the output. Places represent conditions and transitions represent events. The input for each transition is defined by a function specifying the places from which each transition receives its input. Another function defines the places where each transition sends its output. A marking of a Petri net is an assignment or distribution of tokens to places in the net. Tokens always reside in places. A transition may fire if each of its data receiving places has at least one token. When firing, a transition removes a token from each of its receiving places and writes a token to each of its places associated with data output. Firing of a transition means execution of the operation contained within the transition. During execution data is consumed from receiving paths and produced in supply paths to allow the firing of connected transitions.

E-nets are extended Petri nets which consist of locations and transitions. Each location is either a

resolution location or a token location. A resolution location is associated with its own resolution procedure and is used to control token flow.

Colored Petri nets are an extension of the basic petri net in which colors are associated with the tokens. The firing rule for each transition is defined by a function of input-tokens which produces colored output-tokens. A transition is enabled if there is a set of colored tokens, one token from each receiving place, and the set of colors where each color taken from the set of colored tokens is matched to an entry in the table of transitions.

Time extended Petri nets are an extension of the basic Petri net designed to represent time-resolved behavior of interacting parallel processes, holding of system resources, and to allow specifications of models as data structures [19].

The major concern of the programmer in a Petri net-like environment is determining the exact sequence of the various atomic operations that make up the application (mostly arithmetic and data-moving operations and decisions). A basic position underlying this dissertation is that a soarser-grained parallelism is initially more conducive to devising concurrent algorithms, that the programmer should be concentrating on the flow of data through functions that correspond more closely to familiar "real world" functions [31]. This work provides a vehicle

for the programmer that is simple enough to allow expression of parallel algorithms without being concerned with the exact sequence of the various atomic operations. The vehicle is established to allow the programmer to express the algorithm naturally, as data flows from module to module.

C. OVERVIEW - THE BASIC BUILDING BLOCKS.

1. Introduction. Figure 2 shows a graphical view of the programming environment. Each box labeled TERM represents a terminal or interactive device. The TERMINAL INTERFACE (TI) supports terminals and passes messages between terminals and programs in the INTERNAL PROGRAM ENVIRONMENT (IPE). The IPE contains programs composed of modules interconnected by data paths. These programs are entered by application programmers and designed to perform specific functions. DEVICE represents printers, plotters, tape drives and other similar peripheral equipment associated with computing systems. Messages are passed from the TERM to the TI, from the TI to the IPE. Execution begins when messages of sufficient quantity arrive in the IPE allowing activation of modules within the IPE. As a result of active modules in the IPE, messages are either sent to the TI or to devices as required. Messages sent to the TI are routed to the appropriate TERM. Messages contain data and separators. The separators distinguish the end of one message from the beginning of another. Messages which pass through the terminal interface also include a source and destination address.

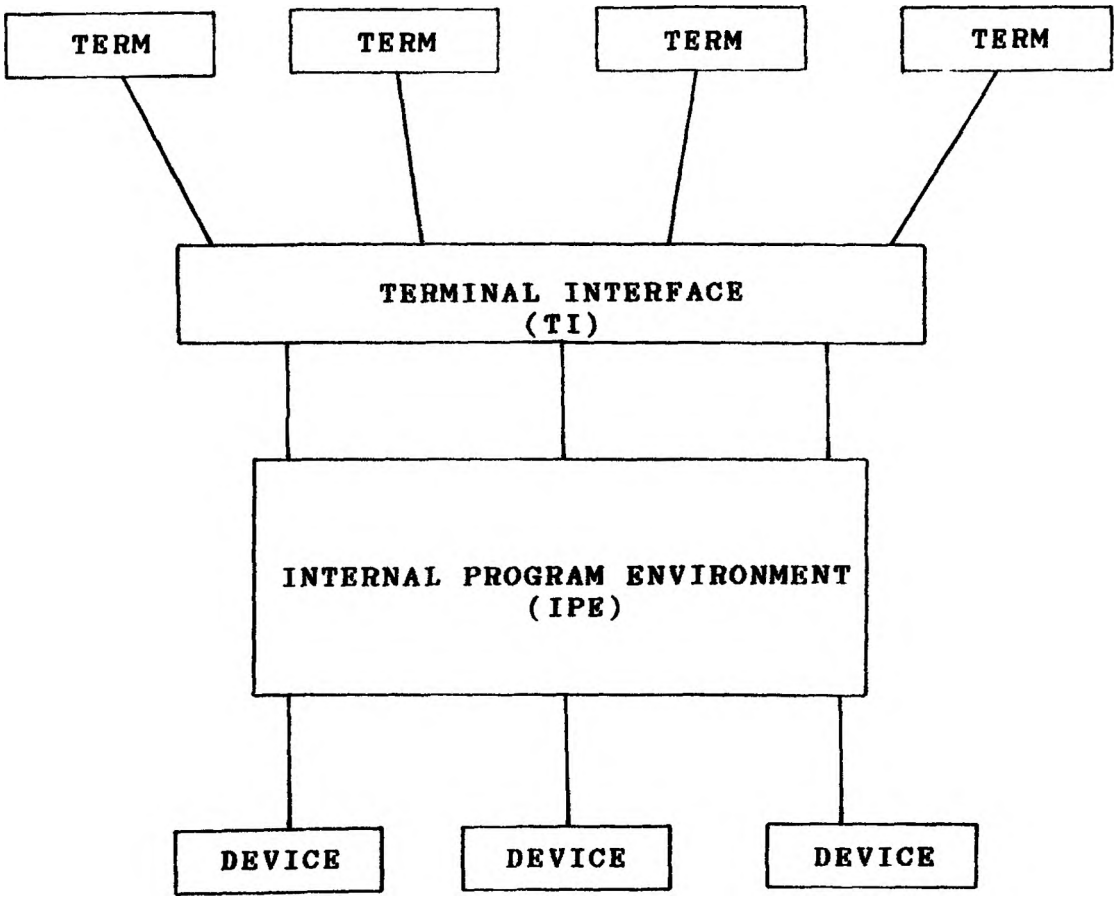

FIGURE 2
PROGRAM ENVIRONMENT OVERVIEW

2. Terminals. The terminals send messages to and receive messages from programs in the IPE. The dialogue between terminals and programs in the IPE is supported by the TI. Data flows from TERM to TI and back again through the paths which are represented by the lines between TERM and TI.

3. Terminal Interface (TI). The TI supports the dialogue taking place between programs in the IPE and the terminals. Messages are routed by a dynamic "source-destination" table as described in section 4. Messages originating at a terminal contain destination program identities part of the message. The user determines the destination by supplying the name of the receiver. The name is supplied at terminal activation time and may be changed by the user at any time during the terminal session. The TI maintains a table of the destination name and user-terminal-identifier pairs for each active terminal. The destination name determines which program or programs receive messages. Associated with each terminal is a unique terminal identifier. The user terminal identifier is retained with a message so that replies can be routed correctly.

Messages originating in a program are either private, semi-private or public. A private message is routed to a specific terminal as determined by the terminal identifier and is generated as a response to a previously received message from a specific terminal. The TI appends the user

terminal identifier to the message when passing it to the IPE. The program, when responding to a message, retains the identifier with the response. Using the retained identifier, the TI routes the return message to the appropriate terminal. Invalid data is an example which requires a response to the specific user terminal entering that data.

A semi-private message originating in a program is destined for a group of terminals. The message is generated to inform terminals performing a particular function of a change meaningful to those users. Suppose a program in the IPE responds to orders of a department store. Upon depletion of a particular item from stock, a message is sent to all terminals placing orders indicating the depletion of that item. The TI determines the receiving terminals by a list of terminal identifiers.

Public messages generated in the IPE are sent to all active terminals. System messages important to all users fall into this category. Messages relating to "system modification" or "system status" are examples of this message type.

4. Internal Program Environment (IPE). The IPE is composed of PROGRAMS, PATHS and PORTS. Programs are composed of MODULES, PATHS and PORTS. The paths and ports of a program perform the same function as the paths and ports in the IPE.

Figure 3 shows an internal program environment. PR1 and PR2 represent two programs in the system. Program 1 contains 3 modules named M1, M2 and M3 and program 2 contains modules M4, M5 and M3. Note that module M3 in this example is duplicated for use in more than one program. Also, any name the programmer desires can be assigned to a port. Port names are used as data transfer devices and hence have only local meaning; therefore, duplication of port names within a system is allowed. The lack of restrictions associated with port names enhances program generality. IPE port names, A, B, C, D, E and F, are used to communicate to the TI and devices which reside outside the IPE. These ports, on the IPE, are known by the system to perform specific functions, such as printing.

Figure 4 shows a module, as represented in a program graph, as a box containing a name. The module is composed of a series of traditional programming instructions, possibly from a high level language such as Pascal, LISP or PL/I. Modules can be included in a program a number of times to improve performance, as resources permit, or to perform the same function on different data.

Figure 5 shows a module with its ports. The ports are identified by a letter at the edge of the module. They pass data from a module to a path or from a path to a module as requested by the module. Ports offer a unified mechanism for communication [11].

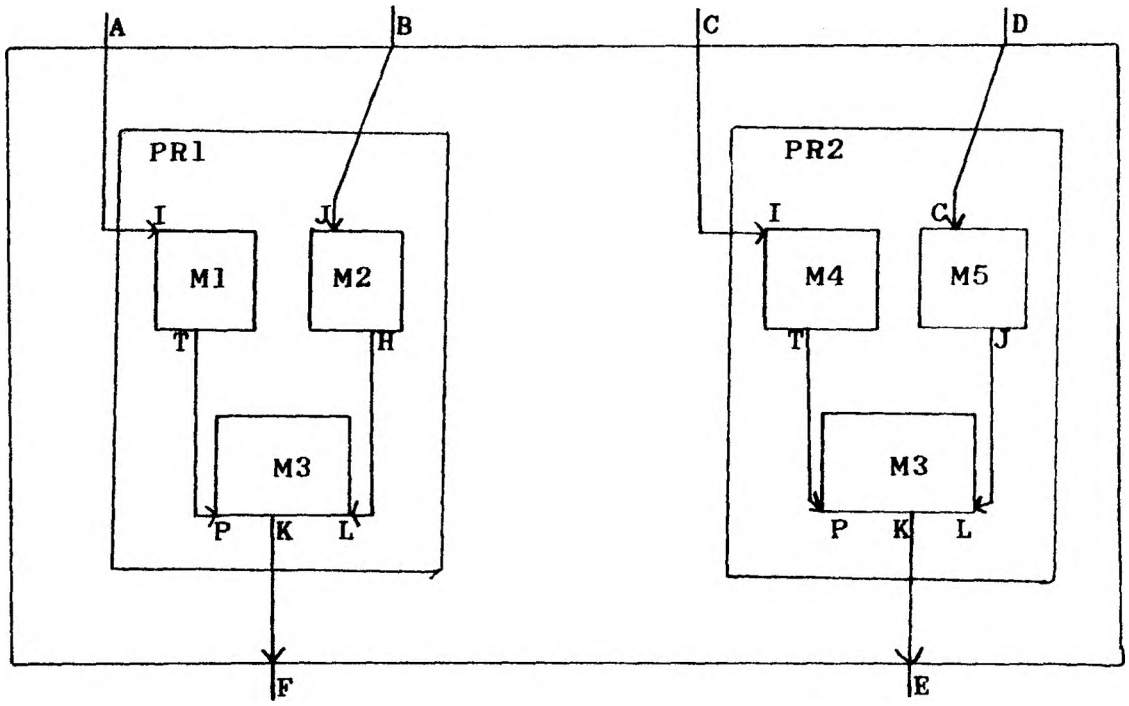

FIGURE 3
INTERNAL PROGRAM ENVIRONMENT (IPE)

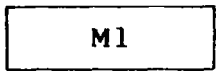

FIGURE 4
MODULE

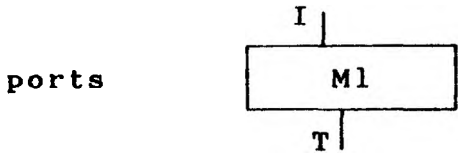

FIGURE 5
MODULE AND PORTS

Figure 6 shows the basic program construct. A module, its ports and the paths which carry data to or from other modules or devices, make up all programs.

Programs communicate with the TI interface by sending messages to ports. Ports A, B, C and D on the edge of the IPE in Figure 3 are being used to pass messages between the program's modules and the TI by placing messages in the associated paths. Ports I, J, T, H, P, K, L and C of Figure 3 are associated with the respective modules. Each module receives messages from and sends messages to its ports. A module communicates only with its ports and is not aware of the source of its messages received or the destination of the messages supplied. Modules can be developed in an independent manner. Knowing the data requirements and the processing functions of a set of modules allows programs to be constructed by connecting ports with data paths.

5. Devices. Ports F and E of Figure 3 are being used to communicate to DEVICES. Devices are distinguished from terminals in that a device is not interactive. A dialogue does not take place between program modules and a device. Messages pass to or from the device in the same manner as messages pass between modules within a program, so modules communicate with devices like modules communicate with modules in the program, except that the message traffic can only be one-way.

paths

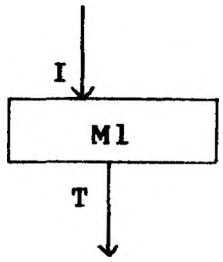

FIGURE 6
MODULE, PORTS AND PATHS

6. Application Programmer Tasks. The application programmer has two distinct tasks when creating programs. First, the modules are constructed to receive, manipulate and send data. Second, the paths are determined which allow data to flow in the appropriate manner. A description of modules provides the programmer with an explanation of the data requirements, functioning of the module, and data to be supplied (in messages) by the module. With this description, a programmer can envision the desired modules and plan their interconnection to create a program. By using a structured, top-down approach, the necessary modules are identified. These modules consist of "old" modules and "new" modules. Old modules are those previously developed and can be included immediately in the program. New modules must be developed by the programmer to perform tasks specific to the program being developed. After the modules are established, the programmer's attention turns to specifying the necessary data paths among them. The programmer's task is finished when data flows between modules allowing them to perform the desired functions. Notice that the flow of messages is then causing the activation of programs, rather than the traditional user-supplied command or job submission.

D. DESIGN DECISIONS - GOALS.

1. Vehicle Aspects. The vehicle presented concentrates on a number of aspects of programming and program execution support. These aspects consist of a terminal environment, internal programs, devices, and operating system support features. The terminal environment is the appearance provided to the user by the terminal interface. The appearance consists of user-friendly features such as split screens or menus. The user can enter modifications to the terminal interface to create desired features. Internal programs are composed of a network of modules. Devices are the physical components, outside of the programming environment, which are used by programs. Sample devices are disk, tape, plotter and printer. Operating system support features allow programs to execute in this environment. The allocation of resources, activation and deactivation of modules and message transfer are just a few of the tasks performed by the operating system.

The purpose of this work is two-fold; to encourage thought about the development of parallel algorithms and to describe the operating system support for internal program modules. The description of the operating system support is required to set the environment for the expression of the parallel algorithms. The programs are constructed by connecting message paths among program

modules. The operating system support and program expression provide the vehicle for the specification of parallel algorithms. Using this vehicle allows programming to move away from the thought pattern of sequential program expression to the development of truly parallel algorithms. It also promotes algorithm expression at a level that does not distract the programmer with low-level details. This research concentrates on the operating system requirements to support this programming vehicle. The operating system support of the internal programming environment is emphasized since new algorithms can be imagined to reside in this environment. The remaining facilities are included to aid in envisioning a complete system.

Support of the IPE has two major aspects: programming support as viewed from the inside of the module and programming support as viewed from the outside of the module. The parallel algorithm can be expressed only when the environment outside the module is in place, so this research concentrates on support facilities external to the module. Support related to message passing, data buffers and activation of modules are some examples of this support. Internal module expression is in pseudo-code in sufficient detail to allow full understanding of the algorithm. Other features, such as the terminal interface, are included to aid the reader in envisioning the power and flexibility of this approach. The data flow

model of programming and program development is followed throughout the programming process. All computing interaction, between users and programs, programs and devices or program modules is accomplished within the data flow model.

2. A Different Concept. The programming vehicle described is a different concept in programming, yet, many traditional programming practices still apply. Structured, top-down design is common to both the traditional program design and to programs expressed within this developed system. The algorithms look and perform differently since no central control or global updatable memory exists. A program module begins execution "on demand". That is, data presence determines the need for module activation. The module's execution continues until the module relinquishes control by issuing a request to the operating system for "sleep". After receiving the sleep request, the operating system allows the module to become dormant. A module in the dormant state does not leave the system but waits for reactivation. The concept of jobs entering and leaving a system is replaced by modules always resident. The traditional job or job step no longer exists. After satisfying an activation criteria determined by data presence, the operating system provides the necessary resources and reactivates the module. A user can add and delete modules

in the system through an editing facility. This facility is a mechanism allowing the creation and modification of program graphs where modules are the nodes and edges represent paths. The specification of module names, ports and interconnecting paths permits the addition and deletion of modules.

3. Qualities. Programming with this vehicle offers a number of advantages. Parallelism is expressed at a high level. Low-level program considerations are delayed until the program design is complete. This high level of programming allows parallelism to be recognized and exploited fully during the program development phase and during program execution. The vehicle, while presenting a new computing concept, does not call for radically different program development practices. Top-down structured program development is fundamental to the success of program development with this vehicle. Most significantly, new modes of thought about algorithms are required since sequential program execution no longer exists. The programmer now considers functions or modules that execute in parallel. The overhead of processor switching is reduced by this model.

Modules that are continually present replace the concept of jobs entering and leaving a computing system. The modules become active when data is present and go dormant when additional processing is no longer required. The concept of modules always present, becoming active

when data exists, is consistent with the data flow model.

When modules are independent, relying only on the presence of data, independent development is possible. Programmer productivity is increased when individual programmers develop modules independently. The specification of data requirements provides the guide for parallel program development. Module programming is done in a familiar procedural language.

Algorithm communication is enhanced by the pictorial representation natural to the data flow programming environment. Pictures present ideas more effectively as demonstrated by graphs and diagrams used in other applications. Programmer efficiency is enhanced when using this representation since complex program designs are clearly represented through data flow graphs.

Many of the problems encountered when attempting parallelism are eliminated with this version of the data flow concept. Synchronization and module communication overhead is reduced when a message passing system is used. Synchronization is no longer required since each module's execution is dependent only on the presence of data. Artifacts such as control tokens are no longer required.

The vehicle used for representing algorithms is a natural extension of familiar processes. Data flows between modules in much the same way paper passes between desks or offices in a business. Many tasks are accomplished in a parallel manner in that environment. An

analogous concept is provided by this vehicle for algorithm expression.

In general, parallelism is not hindered by side effects within this model. Each module executes independently of other modules' execution. The modules are interdependent only through data explicitly flowing in established paths.

4. Summary. Chapter 1 has introduced the basic components along with the motivation and rationale for this work. The need for parallel algorithm development is demonstrated. Traditional data flow and this extension shows the basis of the vehicle described. Data flow is a natural model for this work. Current research shows a recognition of the need for parallel software, but restricts itself to parallelism at a low level. This work demonstrates the benefits of parallelism at a higher level. The basic components of the vehicle are modules, paths and ports. Modules are independent and remain in the computing system in a dormant state until data presence causes activation. The programmer, when developing a parallel algorithm, determines the required data paths between modules and then determines the internal details of the module. The advantages of this vehicle increase the desirability of its further development.

II. THE DATAFLOW PROGRAMMING ENVIRONMENT

A. INTRODUCTION.

A program is comprised of modules and paths interconnecting modules' ports. Modules contain instructions which manipulate data, cause data to be consumed from ports and cause data to be supplied to ports. Ports, which offer a general mechanism capable of supporting all types of communication, are used to pass data between a path and a module. Paths carry data in the form of messages between ports of modules. A programmer constructs programs by determining appropriate modules and then interconnecting ports of those modules with paths. Modules are passive when entered into the system. Data presence, satisfying a specific set of conditions, determines module activation. The set of conditions used for module activation is called a trigger. One trigger is associated with each module and is a list of boolean expressions and associated entry points. A trigger is true if any of its boolean expressions are true. A module is a candidate for activation when its trigger is true. Many modules may be active at one time permitting highly parallel activity. This chapter presents these building blocks - modules, triggers, ports and paths - that comprise the semantic base of this form of data flow programming.

Figure 7 shows a portion of a program. Modules, ports and paths are shown as they are used in a program. Each module corresponds only with its ports and the port passes data between the module and the path. The operating system provides support for data passage between ports.

B. MODULES.

1. Module Introduction. A program consists of modules interconnected by data paths. The program modules store and manipulate data. The module is a basic component of a program and corresponds roughly to a task in traditional programming environments. The module is comprised of programming instructions shown in this paper in pseudo-code. The instructions can be expressed in any supported language, including the new parallel languages when implemented. If necessary, the programming language can be expanded to communicate with ports. Additional instructions allow polling of ports for data presence, perform data consumption and supply data to a port. Any number of instructions comprise a module, however, the number of instructions should be limited so that potential parallelism is not lost. A module may also be defined to consist of other modules and the paths among them. This nesting capability is discussed in detail in Chapter 4.

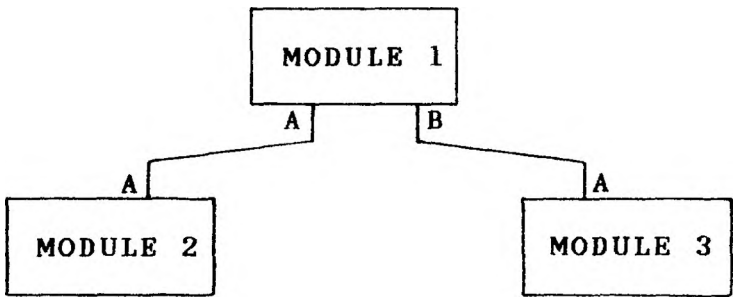

FIGURE 7
MODULES, PORTS, PATHS

Figure 8 shows a module as represented in a program graph. A program graph is a pictorial representation of an algorithm consisting of many modules and paths. Each module is shown as a rectangle and is identified by a unique name. The name identifies the module for use in a variety of programs. Associated with the module name is a description of its function which explains the data supplied and received by the module and all manipulations performed on the data. With this information a programmer can use modules previously constructed and provide information about new modules which are candidates for inclusion in other programs. A module may appear in a program more than once. Parallelism and performance of similar functions on different data occurs when the same module appears in a program multiple times.

Both versions, demand flow and supply flow, of data flow semantics are supported. In demand flow, program modules respond to data requests. In supply flow, a module supplies data as it is generated. Supply flow modules are activated by the presence of data. During the activation, the module may naturally supply data to a port causing activation of other modules. In demand flow, a module makes a request for data from its port. The request is passed along by the operating system to the supplying module. The requested module is activated and supplies the desired data. The operating system supports the source-destination details of a request allowing the

FIGURE 8
MODULE

modules to execute independently of each other. Module dependence exists only through data supply and consumption. If sufficient data and resources exist, concurrent module execution is possible.

The simultaneous supply and update problem of a module capable of responding to data requests is easily solved in this environment. Since a module's activation depends on the satisfaction of its trigger, some data ports can be deactivated while others are active. Hence, a module can be limited to satisfying a request or updating its encapsulated data.

A module may be composed of multiple interdependent procedures and therefore may have multiple entry points. Associated with the conditions for activation of a module is an entry point name. This name identifies the address where control is given when a module becomes active.

2. Module Example. The example in Figure 9 shows a module as presented in a program along with the description and pseudo-code required for understanding. Module specification is concerned with a description of the data supplied and consumed, a general description of the function of the module and the instructions of the module that determine the module's function. Each module must include the data description, the function description and its instructions to be effectively used in a program.

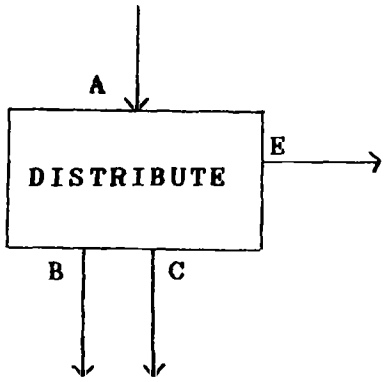

FIGURE 9
MODULE DISTRIBUTE

GENERAL DESCRIPTION: The module's name is DISTRIBUTE. The module receives data from port A. The module supplies some of the data to port B and some of the data to port C as predetermined by a set of criteria. For example, numbers greater than zero can be supplied to port B and numbers less than zero to port C. Non-numeric data is written to port E. The module is activated if any data exists at port A. The distribute module demonstrates a general concept of routing data as determined by a specific criteria.

DATA DESCRIPTION: Data is consumed from port A. Each data item is consumed and processed before another item is read. In the example, numeric data items greater than or equal to zero are written to port B, numeric data items less than zero are written to port C and non-numeric data items are written to port E. Figure 10 shows the pseudo-code for the Distribute Module.

This module might be used in an accounting program to separate credits and debits. After the specification of the data requirements, the module can be used any time a split facility is required. The programmer includes the name of the module and specifies the data paths. The module remains in the computing environment and performs its function whenever data is present. The activation criteria is the module's trigger which is explained in section E. The complete module is shown in Chapter 4 which includes the module's trigger.

```
MODULE PSEUDO-CODE:  MODULE DISTRIBUTE.
  WHILE DATA EXISTS AT PORT A DO.
        READ FROM PORT A TO VARIABLE X
        IF X IS NUMERIC THEN DO.
              IF X IS >= 0 WRITE X TO PORT B
              ELSE WRITE X TO PORT C.
        ELSE WRITE X TO PORT E.
  SLEEP (activation criteria).
  END.
END MODULE DISTRIBUTE.
```

FIGURE 10
MODULE DISTRIBUTE

C. PORTS.

A port is a named data entry or exit point that provides data transfer between one or more paths and a module. A letter near the boundary of a module represents a named port as shown in Figure 11. The module communicates only to ports and requires no additional information about the surrounding environment. The module is independent of the paths connecting its ports with other modules' ports in a program. There can be one path or many paths connected to a port. The module performs in the same manner in either case. Ports, which provide all communication to the module, are capable of communicating all data types. A port offers a mechanism for communicating with devices, terminals, other programs and the operating system. Data passes through a port as a response to an input or output command executed in the module. Programming languages are easily imagined to be extended to include commands of the type READ FROM PORT A and WRITE X TO PORT C where A and C represent port names and X a data value. The module is also capable of querying a port to determine if data is present without actually receiving the data.

During program design, the application programmer must determine the data to be received and supplied by all modules. Upon determination of the data requirements, the programmer establishes ports which represent the point of

data receipt or supply. Section B has shown an example of module communication with ports. All communication is initiated by the module and effected by the operating system.

One or more paths may be connected to a port. The port multiplexes the data received from multiple paths. When the module requests data from the port, the data is received in a FIFO manner. The operating system holds the data in buffers until the module executes a Read command at which time a specific quantum of data is transferred to the module. Multiple paths can be connected to a supply port as well as a receiving port. The fan-in and fan-out of paths from ports are discussed in Section D. The operating system is responsible for providing the port to port communication. Modules are not aware of paths connected to their ports.

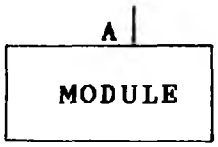

FIGURE 11
PORT REPRESENTATION

In Figure 12, A, B, C, D and E represent ports. The dot on port D indicates the capability of responding to requests. The inquiry port supplies data upon demand. A port is the only facility through which data transfers to and from the module. A port is used in one of three manners:

i. An input port only receives data, i.e. another module writes data to an output port, the data is transferred along a path, and the data is received at the input port.
ii. An output port only provides data, i.e. to paths for transfer to other modules.
iii. An inquiry port, which is a special output port, provides data in response to specific requests. e.g. a module may supply named records upon request.

An inquiry port listed above (iii) is an output port which is capable of responding to requests. The requested module must be active and execute an input command to receive the request from the port. The request can be used to trigger module activation. After determining what is requested, the module supplies the response to the same port. A port capable of receiving requests has a bold dot associated at the origin of the path which marks it as a port capable of responding to requests.

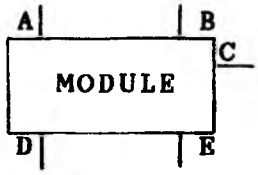

FIGURE 12
MULTIPLE PORTS

D. PATHS.

The data path is a representation showing data transfer between modules' ports. Data is written into a port by a module. The operating system transfers the data from the supply port to the receiving port as indicated by the connecting path. Then conditions for activation of the receiving module are evaluated. If sufficient data exists, the module is allocated resources and begins execution at the specified entry point. During execution, the module issues an input command to the port where data exists, consuming the data. Special markers are included with the data to distinguish the beginning and end of data records. The data remains in a path until the receiving module "consumes" it. Buffers are maintained by the operating system to hold the data between the time a supplying module writes data into a path and the time a receiving module reads the data. The operating system buffer support is transparent to the executing module.

Figure 13 shows a module with five ports and paths connected to them. The unconnected ends of the paths are connected to ports of other modules. The arrows indicate the direction of data flow. If the arrow points to the port, data flows to the module (data received). When the arrow points away from the port, the data flows from the module's port (data supplied). The dot, associated with port D in Figure 13, indicates the capability of

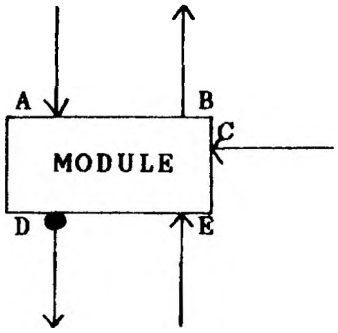

FIGURE 13
INPUT, OUTPUT, REQUEST PATHS

responding to inquiries. This path receives requests for data and the module supplies the corresponding data through the same port, D. Supply flow occurs at port B. Data is written to port B and the operating system places data in the receiving module's port as shown by the path. No demand occurs for this data. Data written to a port allows activation of other modules. Data in paths connected to ports A, C, and E is read by the module for further processing. Data presence at these ports is used to determine if activation of the module is possible.

1. Fan-in. Figure 14 shows data from more than one source merging at a port. Paths connected in this manner are said to fan-in. With this type of connection, data originating from a number of sources is supplied to a module through a common port. The data is read at the port with no concern for the origin of the data. A merger of data records from an unknown (to the module) number of modules is possible by the fan-in of paths, allowing all data to be received through a single port. The distribution of data is external to all modules, delaying the binding of the number of receiving data paths associated with a module until the module is included in a program graph.

Activation criteria are not effected by paths having fan-in. The criteria are concerned with the presence of data at a port, not the source of that data, so modules

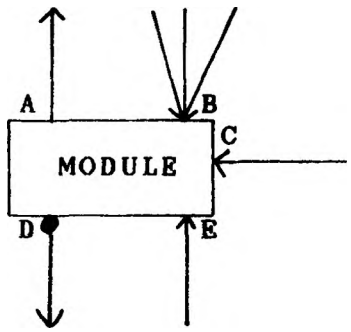

FIGURE 14
FAN-IN PATHS

can be developed independently of the paths associated with their ports. The data from multiple sources is added to the buffer associated with the port. Data presence in the buffer is used to determine activation of the receiving module.

2. Fan-out. Figure 15 shows multiple paths originating at one port (E). This concept is called data fan-out. Two types of data fan-out are possible. The first, as shown in Figure 15, is fan-out associated with a supply port (data output port type ii). When fan-out occurs as demonstrated by the paths associated with port E, a copy of the supplied data is placed on all paths. The receiving modules get exact copies of the same set of data. The data is placed in each path as an action of the operating system, the module is unaware of the distribution of its data. The number of paths connected to a port is transparent to a module. The programmer requests data duplication in the program graph by including multiple paths at a port.

If multiple paths originate at a port which responds to data inquiries, as in port F of Figure 16, selective data response is required. Data is supplied to only the requesting module. Inquiries are queued at the port and satisfied in the order of receipt. The operating system monitors the source of the inquiry and routes the response accordingly. The supplying module services one inquiry at

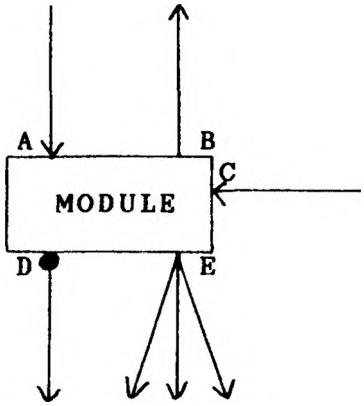

FIGURE 15
DATA SUPPLY FAN-OUT

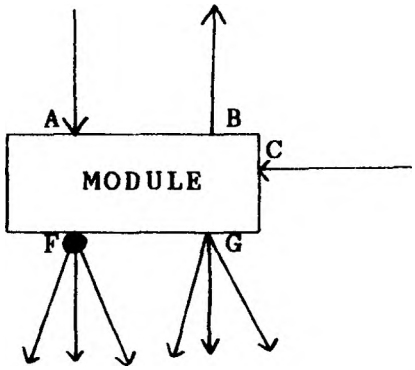

FIGURE 16
DATA INQUIRY - FAN-OUT

a time. The response is placed in the appropriate path by the operating system.

3. Branch-out. Figure 17 shows paths that branch-out from a port. These paths differ from paths that fan-out since a copy of the data is not placed in both paths. A branched-out path is marked with a circle at the branch point. The operating system determines the path with the least amount of data waiting to be processed. New data is placed in that path. Distributed paths are used to improve performance by separating the data. Identical receiving modules are used when the data is branched at a path. Modules with high computing demands can be replicated with data branched to them to improve high data demand processing.

Paths are specified when the program is entered into a computing system. During execution of the program, paths remain fixed. There is no facility envisioned for dynamically changing the path-port association while a program is executing. Only during editing, when modules are added or deleted from a program, may paths be changed.

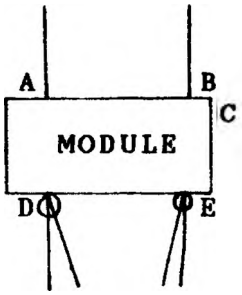

FIGURE 17
DATA BRANCHED-OUT

E. TRIGGERS.

A module becomes a candidate for activation when its trigger is satisfied. A trigger is a list of predicate expressions which are true or false as determined by the presence or absence of data at ports. An entry point is associated with each predicate expression and indicates the instruction in the module where execution begins when the module is activated due to the satisfaction of that expression. One trigger is associated with each module. Port names, the modifier EOF, and boolean operators, AND, OR and NOT, comprise a predicate expression. The boolean value of a port name is determined by the presence or absence of data at that port. For example, A is true if a data message exists at port A. EOF B is true when the end of file (EOF) marker has arrived at port B. (A OR EOF B) is true if either of the above conditions are true. When a predicate's value is true, the module is a candidate for activation with control passed to the entry point associated with the predicate. A set of predicate-entry pairs constitutes a module's trigger.

Triggers are supplied with the module and maintained by the operating system. Triggers are dynamic. During the module's existence new triggers may be supplied to allow different data presence conditions to meet activation criteria. The operating system receives triggers at two epochs in a module's life: at module

birth and each time a module ceases execution with a "put me to sleep" request. The module remains inactive when the current trigger is unsatisfied. The module's trigger is tested when data arrives by evaluating its predicates to determine if an activation criterion has been met. If so, (and resources are available,) the operating system activates the module and control passes to an entry point expressed in the trigger. Upon consumption of the data, a module establishes a new trigger and relinquishes control to the operating system by issuing a "put me to sleep" request. The module does not leave the system, but releases resources for use by other modules. When data presence again satisfies a predicate of its trigger, the module is reactivated.

Triggers are the sole activation criteria of a module. Each trigger is supplied with the module when it is entered into the system (at module birth) and when it is put to sleep. The syntax of the trigger is shown by the following Backus-Naur form rules:

- -

Note: the following symbols are meta-symbols belonging to the formalism, and not symbols of the TRIGGER.

::= : { }

The brackets denote possible repetition of the enclosed symbols zero or more times. In general,

A ::= {B}

is short form for the purely recursive rule:

```
A ::= < empty > | AB
```

```
< TRIGGER > ::= TRIG  < NAME > : < PREDICATE > ,
                                        < ADDRESS >;
    { < NAME > : < PREDICATE > , < ADDRESS > ; }
< ADDRESS > ::= < LETTER >  { < LETTER OR DIGIT > }
< NAME > ::= < LETTER > { < LETTER OR DIGIT > }
< LETTER OR DIGIT >  ::=  < LETTER >  |  < DIGIT >
< LETTER >  ::=  A | B | C | D | E | F | G | H |
                 I | J | K | L | M | N | O | P | Q |
                 R | S | T | U | V | W | X | Y | Z
< DIGIT >  ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
                    8 | 9 | _
< PREDICATE >  ::=  < PRED >  |  < EMPTY >
< EMPTY >  ::=
< PRED >  ::=  < TERM >  |  < PRED >  OR  < TERM >
< TERM >  ::=  < FACTOR >  |  < TERM >  AND
                                        < FACTOR >
< FACTOR >  ::=  < PORTNAME >  |  ( < PREDICATE > )  |
                 < PREOP > < PORTNAME >  |
                 < PREOP > ( < PREDICATE > )
< PREOP >  ::=  NOT  |  EOF
< PORTNAME >  ::=  < LETTER >
```

The value of a predicate expression is true or false. Evaluation of the predicate is required to determine

activation of a module. ADDRESS is the entry point name of the module where control passes when the module is activated due to satisfaction of the associated predicate.

1. Trigger Expression.

EXAMPLE 1:

TRIG ONE: (A AND B) OR NOT(EOF C OR D), ENTRY_1;

This example has the following meaning:

TRIG is a key word marking the beginning of the trigger. ONE is the name of the predicate used to identify this predicate from other predicates in a trigger list. (A AND B) OR NOT(EOF C OR D) is the predicate evaluated to determine if activation of a module is possible. The parentheses show typical grouping with evaluation being performed inside-out. A, B, C, and D are port names in the associated module. A port name like A is considered true if any incoming data exists in the path attached to that port. Standard boolean evaluation is performed for AND, OR and NOT. EOF C is considered true if a special marker (EOF) is the next communication to be entered through port C. NOT (EOF C) is the opposite of EOF C and considered true when the special marker (EOF) is not the next communication to be entered through port C. In general, the (EOF) marker is used to identifty completion of the data received. The empty predicate is considered false. Predicates within the trigger are labelled ONE, TWO, etc. as shown in the examples. The predicates are evaluated in order of the numerical label with the first

predicate satisfied determining module entry point. Two predicates can specify the same module entry point. The predicate appearing first will be checked for module activation before later appearing predicates. Multiple predicates specifying the same entry point are equivalent to a predicate containing each of the multiple predicates logically combined. A trigger that does not clearly specify the activation criteria for a module will result in an error.

EXAMPLE 2:

```
TRIG  ONE:  A  OR NOT  B, ENTRY_1;
            TWO:  NOT C  AND B, ENTRY_2;
```

This example shows two entry points into a module. If no data is present at port B or if there is data present at port A then the module can be activated with control passed to entry point name, ENTRY_1. If no data is present at port C and data is present at port B, the module can be activated with control passed to entry point name, ENTRY_2. A module, at sleep time, can update either or both of the predicates by specifying the name, ONE or TWO in this example, and the new predicate. A predicate whose name is not specified will be left unchanged from one activation to the next.

The predicates of a trigger are evaluated in order with the entry name of the first predicate found true receiving control. The order is determined by the ONE,

TWO, etc. labels associated with each predicate. ONE is evaluated first, TWO second determined alphabetically. The labelling of predicates places a precedence on the evaluation of predicates. The first predicate found true will cause execution to begin in the module at its associated entry point address. Special cases can hold the ONE or TWO labels and the general case predicate placed at a lower priority.

A trigger is associated with each module. A predicate may be associated with each entry point in a module. When the predicate is satisfied, the module is scheduled for activation and, when activated, control passes to the entry point associated with the predicate. The module will begin execution immediately if processing resources are available to support the execution of the module. When resources are not immediately available, the module will be activated when resources become available with no re-evaluation of the trigger. The operating system maintains a trigger for each module. The list of predicates in a module's trigger is scanned sequentially and control passes to the address name associated with the first predicate found true. The list of predicates in the trigger is evaluated each time data arrives at a port. Since only one predicate is found true for a module, each module begins execution at a well-determined entry point. If the module is supported by a parallel programming language, further parallelism can be attained within the

module. When a module finishes execution at one entry point, it submits new predicates with entry points for the module's trigger. The module executes a "put me to sleep" command causing the operating system to update the list of predicates for the trigger of the module and releases resources. In the case of other predicates having been satisfied during its activity, the module can become immediately active with control passing to the entry associated with the first predicate satisfied is the list of predicates.

2. Trigger Example. Consider the example of Section B.1. To increase flexibility of the module, an additional port can be included to allow entry of the number used as a split point. That is, the numbers greater than the split point are supplied to port B and the numbers less than the split point are supplied to port C. The module requires input from port S as shown in Figure 18 before input from port A since port S provides the splitting point. Triggers allow the proper execution without significant modification to the instructions within the module. The addition of the trigger is shown in this example.

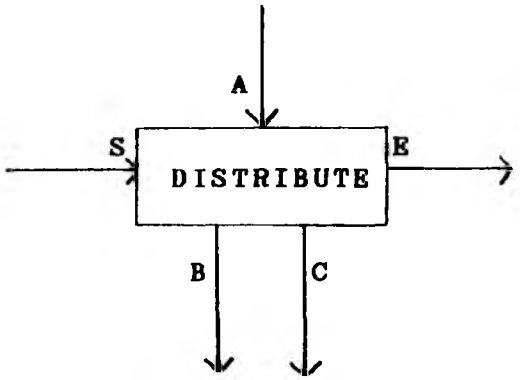

FIGURE 18
ENHANCED MODULE DISTRIBUTE

The module elements are:

GENERAL DESCRIPTION: The module's name is Distribute. The module receives a data element from port S. The data received from port A is split with the numbers greater than or equal to the data element of port S supplied to port B and numbers less than the data element of port S supplied to port C. Non-numeric data is written to port E. The trigger associated with the module determines the module's activation.

DATA DESCRIPTION: Data is consumed from port S. After port S has provided its data item, each data item is consumed from port A and processed before another item is read. Numeric data items greater than or equal to the item from port S are written to port B, numeric data items less than the item from port S are written to port C and non-numeric data items are written to port E.

The module design can be used wherever data separation on any key is desired. The module is basically the same as in Section 2.B but is more general due to the split point being bound to the module during execution rather than at module construction. It is shown in Figure 19.

Line 1 gives the operating system the initial trigger when the module is entered into the system. Lines 2 through 7 will be executed first due to the predicate in

```
   MODULE DISTRIBUTE.
1  MODULE TRIGGER:  TRIG ONE:  S, ENTRY_ONE;
                        (*initial trigger supplied when the
                        module is entered into the program*)

2  MODULE PSEUDO-CODE:
3        ENTRY_ONE:
4             READ FROM PORT S TO VARIABLE SP.
5             NEW TRIGGER:  TRIG TWO:  A, ENTRY_TWO;
6        SLEEP
7        END ENTRY_ONE.

8  ENTRY_TWO:
9                  READ FROM PORT A TO VARIABLE X
10                 IF X IS NUMERIC THEN DO.
11                      IF X >= SP WRITE X TO PORT B
12                      ELSE WRITE X TO PORT C.
13                 ELSE WRITE X TO PORT E.
14            SLEEP
15       END ENTRY_TWO.
16       END DISTRIBUTE.
```

FIGURE 19
MODULE DISTRIBUTE

the trigger. Upon completion of lines 2 through 7, the new trigger will include "TRIG A, ENTRY_TWO". This new predicate follows the old one so that lines 8 through 16 will be executed in response to arrivals at port A only until a new value arrives at port C. This module is organized to continue operation in this manner, controlled by the structure of the triggers.

This example illustrates that if the trigger is comprised of a predicate list, only selective predicates need be changed. The operating system will update its list when acting on the module's sleep request. The predicates not included in the statement will be retained in the new predicate list. The name associated with the predicate determines which predicates are updated. Predicates can be eliminated from the list through the null predicate.

F. SUMMARY.

This chapter introduced the concepts of modules, ports, paths and triggers. Emphasis is placed on a module always being resident in a computing system. The module specification is shown by its inclusion in a program graph, the general description of the module function, a description of the data consumed and supplied, and a listing of pseudo-code describing its actions. Ports are used as a communication mechanism capable of supporting all types of data messages between paths and modules. The ports are the only link a module has to its external environment. All data communication takes place through the ports and is specified in the instructions for data receipt or supply. Two types of ports exist: input and output. Input ports only receive data. Output ports supply data either as a result of data requested or data generated naturally. Multiple paths may lead into a port, allowing the module to receive data from a number of sources through one port. Multiple paths may lead from a port, allowing the module to supply data to a number of modules through one port. The fan-in feature allows a module to accept data messages from an arbitrary number of sources without the module providing separate ports for each possible source. The fan-out feature allows data to be distributed to a number of destinations without affecting the module. The binding of the number of paths connected to a port is delayed from module creation time

until the time a module is included in a program graph.

Paths represent the flow of data between modules' ports. The programmer is challenged with two tasks when developing a program: to determine the necessary modules and their function and to determine the paths among them. The paths govern the data flow from module to module. Module development time is reduced if the path construction is carefully done. Data transfer is supported by the operating system and is transparent to the module. Operating system facilities queue data messages on paths until receiving modules are ready to consume them. Paths can fan-in and fan-out from a port allowing a single port associated with a module to supply data to a number of modules' ports or to allow a single port to receive data from a variety of modules. All this allows module development tasks to focus upon triggers and the processing they are to initiate, relieving the programmer from including instructions in the module to effect inter-module communications. Programmer efficiency is increased through this facility and modules are more general.

Triggers are used to determine module activation. They consist of a list of predicates and associated entry points. The predicates evaluate to true or false depending on data presence at module's ports. The operating system evaluates a module's predicates whenever data arrives at one of its ports. The first predicate found true is used for the activation of the module with

control passed to the entry point associated with the predicate. Dynamic triggers allow changing the data availability conditions which are to cause module activation. The trigger is included with a module when the module is entered into the system and new or altered predicates are provided when a module becomes dormant. The operating system maintains the list of triggers and performs the evaluations.

III. OPERATING SYSTEM IMPLICATIONS

A. INTRODUCTION.

The discussion of operating system support facilities is contained in this chapter. Support of the programming environment includes message management, suspending and waking a module, replication of modules by the operating system, the terminal interface, support for devices such as printers and plotters, and processor to module binding. These support features are transparent to the user. The support is an extension of traditional operating system principles already in existence.

B. MESSAGE MANAGEMENT.

1. General. The designer of a program has two tasks. First, modules are developed to manipulate data. A part of module development is a description of data supplied and received at each port. Secondly, the programmer must determine the necessary paths for data transfer between modules. Path determination is aided by the data description associated with each module. The construction of paths yields a directed graph which shows the source and destination of each message path. The paths are shown by lines connected to the ports of modules. Each path unambiguously determines the source

and destinations of a message without aid from the supplying module or the receiving module. Paths are bound to ports of modules after module development is complete, during the inclusion of a module in a program. Data is written to a port associated with the supplying module and is routed to the receiving module's port as determined by the path. Upon satisfaction of the trigger, the receiving module is activated. Data is transferred to the receiving module when execution of a "READ" instruction causes data to be consumed through a port.

2. Buffering. The necessary data paths interconnecting modules are determined by the programmer with aid from the description of data consumed and supplied. Each module has a description of its data requirements associated with the module. Using the description, the programmer can include the appropriate paths that direct the flow of data between modules. A receiving module may not be ready to receive the data when it arrives at its port. The module might be suspended, requiring satisfaction of its trigger for activation, or might be processing other data. Therefore, buffers are required to hold the arriving data until a "READ" instruction allows data to be moved into the module.

The buffering is performed by the operating system and is done for each receiving port. A buffer is storage which holds messages until the receiving module reads them. When data arrives at a previously empty buffer of a

suspended module's port, the operating system evaluates the trigger to determine if activation criteria has been met. All messages are held in the buffer until the receiving module issues a "READ" instruction for that port. The "READ" instruction causes data to be transferred into the module, which releases buffer space. The port name is specified in the instruction.

Data transfer is performed by the operating system with control returning to the executing module upon completion of the task. Data transfer between paths and modules is similar to data transfer between disk and executing programs in current computing systems. Data from multiple suppliers, in the case of fan-in, is interleaved in the buffer as it arrives. There is no priority associated with the data arriving at the buffer. Fan-in is supported by the system and is transparent to the receiving module.

Data supply occurs when a module executes a "WRITE" instruction. A port name specification is part of the instruction. The instruction causes data to be moved from the supply port of the sending module to the receiving module's port. The data at the supply port is placed in the buffer of the receiving module's port as indicated by the path connecting the modules. Buffers are not required at supply ports of the source module. When paths fan-out from a supply port, the messages are replicated and placed

in the buffers of all receiving module's ports by the operating system.

3. Data Request. Data supply ports which respond to data requests have an operating system supported queue associated with the port. The queue holds the requests until the receiving module reads the request. Each request is processed in the order received. A request can, for example, indicate the need for a named data quantity. Each request is queued at the data supply port which is to respond to the request. When the requested module is activated, each request is satisfied one at a time in the order of receipt. The source of the request is held by the operating system. When the request has been satisfied, the operating system routes the data to only the requesting module. The supply module is not given and does not know the source of the request. All routing is performed by the operating system.

Figure 20 shows two modules requesting data from a third module. Module 1 is an order-receiving module which requests data from the inventory module, module 3. The data requests if module 1 can fill the order by determining if the item ordered is in stock. Module 2 checks inventory to determine if a particular item is a standard stock item or if it must be specially ordered. Each module requests data from the inventory module, module 3. When a request arrives at port C, the operating system retains the source of the request. For demonstration

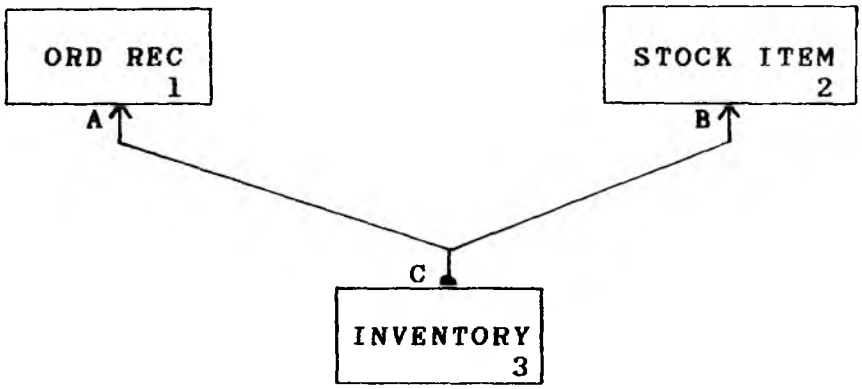

FIGURE 20
DATA REQUEST

purposes, assume the first request originates at module 1. While module 3 is processing the request, a request from module 2 arrives at port C. The later request is held at the port until the first request is satisfied. When module 3 writes the requested data to port C, the operating system routes the data to port A of module one. The second request is now ready for processing by module 3. Since module 3 processes one request at a time, the operating system always knows the destination of the response. Pending requests are queued at port C. Each request is processed individually allowing appropriate routing of the response.

4. Directory. Messages are routed as described by the directed graph of paths designed by the programmer. The operating system maintains a directory of sources and corresponding destinations. Each pair consists of a module's supply port name and a list of module's receiving port names. A list is required to represent the fanning-out of paths from a single port. When data appears at a source, the corresponding destinations are found from the directory and the data message placed in appropriate buffers for receiving ports.

The source-destination relation maintained in the operating system is static. The structure is not modified during execution except during module replication by the operating system. The user causes the structure to be modified when the addition or deletion of modules occurs.

When new modules are added to the program, the directory is updated to reflect the new data paths that are added. Similarly, when modules are deleted from the program, the directory is updated to show the elimination of data paths. Modules are added, deleted and updated as program development occurs. Updating is required when modules are added (to perform new functions) or when modules are deleted (found to be in error, of inferior design, or no longer required).

5. Message Composition. Messages are composed of values written to a port as a group. The supplying module separates data value groups by an "end of record" marker (EOR) and may terminate a set of messages by an "end of file" marker (EOF). This separation is done in the same manner as records are determined in traditional languages. A test for the end of file marker may be used in the expression of predicates as discussed in chapter 2. The size of the message is not fixed and may vary from one message to another.

C. MODULE SUSPENSION AND WAKEUP.

1. Requested Suspension. Modules are suspended at their request. When current data is exhausted, the module issues the "SLEEP" instruction which is a request to the operating system for suspension. When a module is suspended, the operating system saves the environment internal to the module. The value of variables can be either reinitialized or their current value retained for module reactivation. Value retention is the default. When the module is suspended, resources, processors and other support, are released for use by other modules. The buffers are retained for destination ports since data received at an inactive module must be saved. Request queues are maintained for data supply ports capable of responding to requests. Messages in the buffers and requests in queues are used to determine satisfaction of triggers. A trigger modification may be passed to the operating system when a module is suspended, (the dynamic triggers allow varying message presence conditions to determine module activation). The operating system rewrites the list of predicates that comprise the trigger when module deactivation occurs. If sufficient real memory does not exist, the module's space may be released by traditional roll-out to secondary storage.

2. Suspension Without Request. Suspension can occur if the module is doing little processing. In this case, no request by the module for suspension exists. Conditions are evaluated by the system to determine the amount of data an active module is processing. When the presence of data is such that the module is doing very little processing, the operating system can suspend the module. During the suspension, data is allowed to collect in the buffers of the module's ports. When data in sufficient quantity exist, the module is restarted. No request for suspension or writing of new triggers occurs. The module performs exactly as if it were continually active. Resources allocated to the module are freed for use by other modules during this suspension. Performance of the system is improved since module support is not continuously required.

3. Wakeup. Wakeup or activation of a module after a requested suspension occurs when one of the predicates of its trigger is satisfied. Initialization as required by the module, allocation of resources and control passed to the address associated with the predicate completes a module's wakeup procedure. The module now executes independently of any other module in the system.

Wakeup after a non-requested suspension requires allocation of resources to support the module. Execution begins at the instruction following the last executed instruction. Triggers are not involved. Execution

proceeds as if the module had never been suspended.

This type of module suspension is required if limited resources exist. Suspension is a mechanism of releasing resources. The suspended modules remain in the system, becoming active when data presence in sufficient quantity indicates the need for activation. Conceptually, all modules remain active in the system at all times.

D. MODULE REPLICATION BY THE OPERATING SYSTEM.

1. General. The operating system can duplicate a module in order to improve computing efficiency. Duplication consists of creating an identical module, adjusting data paths, allocating resources and activating the new module. The purpose of duplication is to reduce bottlenecks created by high data flow to a module. Data is supplied to the module with the least amount of data waiting to be processed as determined by the quantity of data held in the buffers. Data accumulation at a module's port is one criteria used in determining a need for duplication. When buffer size becomes very large, the receiving module's processor is probably degrading system performance. Assuming the module is active, a large buffer indicates the need for module replication. Modules are replicated in the following manner:

1) Copy the module's instructions.

2) Copy the module's trigger.

3) Allocate resources to the copied module.

4) Branch-out the data paths supplying the original and duplicated module.

5) Fan-in data paths supplied by the original and duplicated modules.

6) Activate the module.

In a branched data path the messages are supplied to the module whose buffer contains the least number of messages. Figures 21 and 22 show how branches are constructed. Figure 21 shows how a module looks in a program segment. Figure 22 is the same program segment with the module replicated by the operating system. The path is designated with a circle at the branch point to indicated data distribution for each path. The data is placed in one of the branch paths, that path with the smallest amount of data waiting for processing.

2. User Allowed Replication. The programmer must allow duplication by marking a module. Only modules marked by the programmer can be replicated by the operating system. Not all modules can be replicated by the operating system since nondeterminacy can be introduced. If a module can be replicated, the programmer must include an instruction of the form "BREAK" which allows the operating system to temporarily halt the module's execution, perform the replication, and restart the module and its replication.

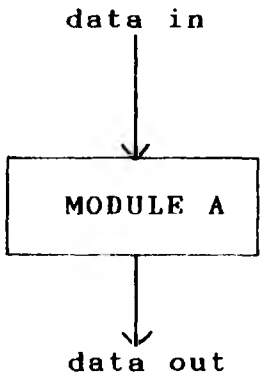

FIGURE 21
ORIGINAL

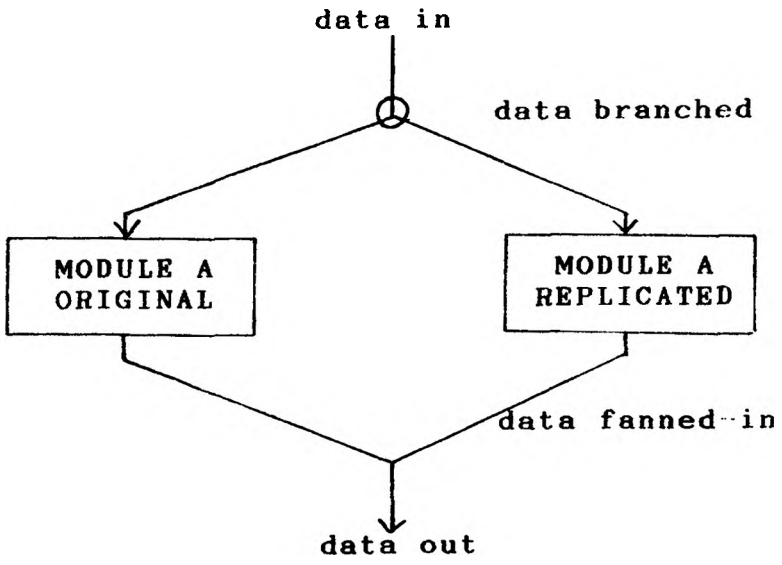

FIGURE 22
MODULE REPLICATED

Programmer-marked modules are candidates for replication by the operating system. Modules which store data warrant special consideration. If such modules are replicated, caution must be exercised to ensure data integrity.

The sequential order of data supplied by a module and its replication may not be the same as if replication did not occur. Caution must be exercised when replicating modules that supply data whose sequential order must be maintained. A replicated module's processor may execute slower or faster than the original module's processor. This causes data to appear in the data supply path relatively sooner or later than one module would have supplied. Supplied data from replicated modules is multiplexed at the receiving port just as any other fanned-in data. No order preserving facility exists for data receipt.

3. Path Modification. To replicate a module, the operating system causes data receiving paths to branch-out to the module and its replication. The operating system distributes the incoming data so that each module and its replication shares in the computing load. Figure 23 shows a program example with modules that receive orders, check inventory, and retain back orders. The data receiving paths labelled "new orders" and "back orders", paths 1 and 2 of Figure 24, show the required paths due to module

replication. These paths are branched-out to supply data to the original module and its replication.

The example in Figure 23 shows a simple program segment. Module 1 writes data to module 2 which in turn writes data to module 1 again. Module 1 requests data from module 3 and in turn writes data to module 3. The program is a segment of a mail-order program shown in its entirety in chapter 4. Module 1 is an order receiving module, module 2 processes back orders, and module 3 stores the inventory of a mail-order house. Since module 1 manipulates, receives, and supplies data with all other modules, it is a potential source of a data bottleneck. Suppose the operating system determines that replication of module 1 is required. Figure 24 shows the program segment after replication.

Data supply paths to the module and its replication are fanned-in at all receiving modules. Paths labelled "back ord." and "inv. update", paths 3 and 5 of Figure 24, show the required fan-in. The replication of a module by the operating system is transparent to any other module. Module replication is contingent on: 1) the module being active, 2) sufficient resources available to support a replicated module, and 3) programmer marked as replicatable. A processor must be available to activate the replicated module.

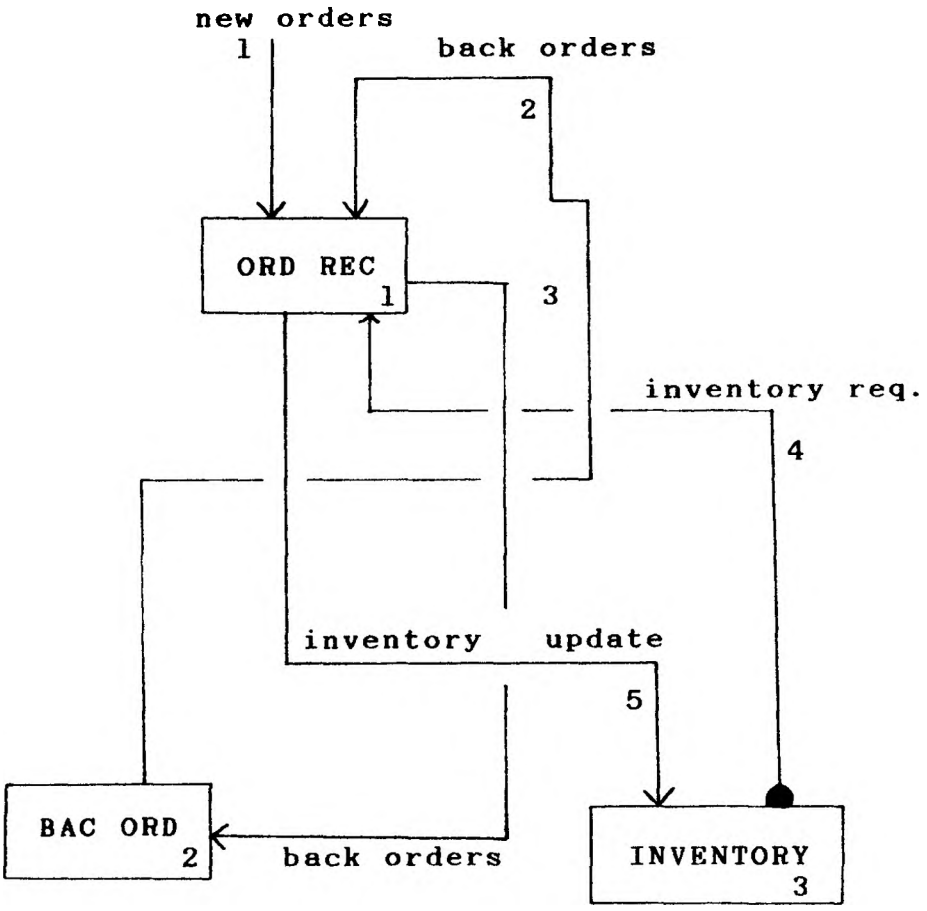

FIGURE 23
PROGRAM SEGMENT

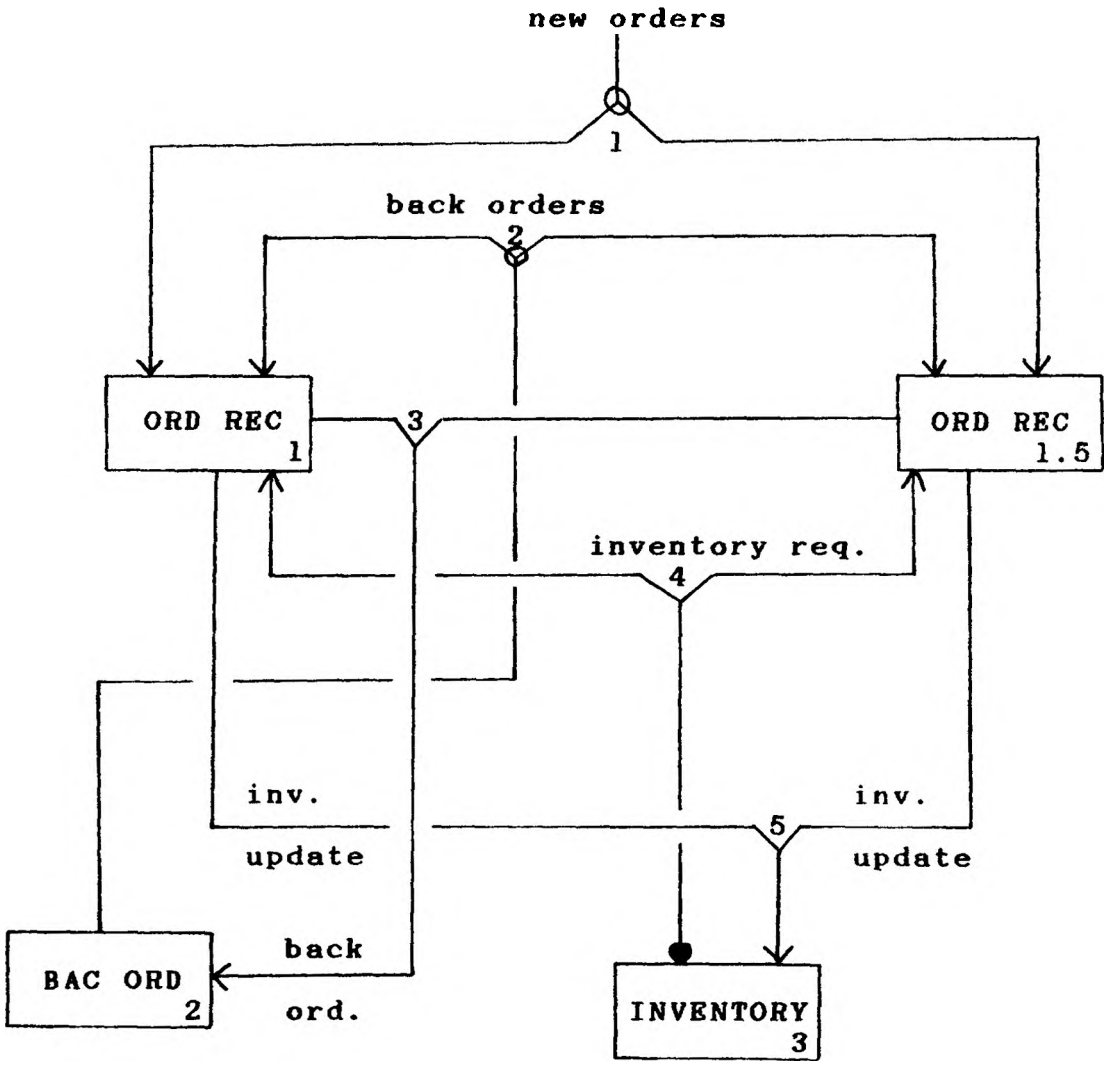

FIGURE 24
MODULE REPLICATION

The paths are shown as required for proper data transfer. Paths 1 and 2 of Figure 24 show data distributed to the original module and its replication. Paths 1 and 2 use the branch-out of data since a copy of the data is not placed in each path as previously described for paths that fan-out; instead, the data is supplied to the module with the least amount of data waiting for processing as indicated by its buffer size. The operating system is able to distinguish between this branch-out and programmer designed fan-out, discussed in chapters 2 and 4, since a record of path branch-out due to replication is retained in the operating system. Path 3 of Figure 24 shows data being merged as a result of the replicated modules. The data is actually merged at the port by the operating system. However, for simplicity, the paths are shown merged near the data supply port. Messages from multiple sources might be interleaved at the receiving port. All data is received at the back-order module, module 2, just as though it were supplied from one module. But, the orders received from the two modules may not be in the same sequence as would have occurred if supplied from one module. The programmer determines if this degree of indeterminancy is crucial to the execution of the program. If so, module 1 must be marked as non-replicatable.

Multiple request paths emanating at a port are shown at path 4. The satisfied request will be routed to the

appropriate module. The operating system retains the name of the requesting module and routes the response to that module. When replication takes place, the operating system assigns a unique name to the replicated module, thus proper routing can be accomplished by using the module's name. Data is merged at path 5 in the same way data is merged at path 3.

4. System Support of Replication. Reconsider the example shown by Figure 17 and reproduced in Figure 25. Module "DISTRIBUTE" works by reading the splitting value from port S and reading the data from port A. When present, a new splitting value is read and then port A again, and so on. If the module supplying port A is replicated, a potential problem exists. Each module and its replication are comprised of the same set of instructions. If a module supplies an EOF marker, its replication will also supply the same marker. Any receiving module which uses the EOF marker in its trigger is not aware of the replication. The operating system ensures proper execution by stripping off the extra marker. The receiving module receives the same set of data after the supplying module has been replicated. Thus, if there are two modules, an original and its replication, the first EOF marker is stripped from the data stream. When a module is replicated more than once, resulting in N modules performing the same function, N - 1 EOF markers are stripped from the data. The system

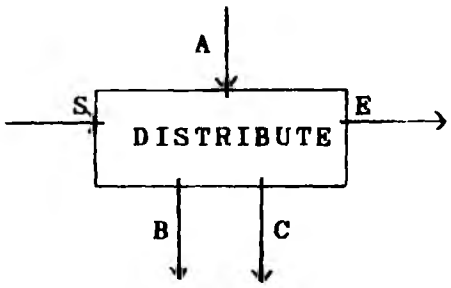

FIGURE 25
ENHANCED MODULE DISTRIBUTE

is responsible for keeping track of replicated modules and ensuring proper execution when replication is performed.

5. Module Reduction. The operating system is responsible for replicating modules when needed. The recognition that replicated modules are no longer needed is also an operating system function. By keeping statistics on the amount of data in the buffers associated with the ports and the rate of data flow through the paths, the operating system can asses the need for replicated modules. If a module is no longer needed, the system can dispense with the module by reversing the steps for replication. Branched paths added during replication are merged and the program returns to the prereplication state.

6. Linear Growth. When modules are replicated or deleted, one module is added or subtracted from the system at a time. Suppose a module (1) is replicated so that modules (2), (3) and (4) exist where (2), (3) and (4) are the replications. If the data load indicates need for replication of all of these modules, the operating system creates only a fifth module, not four more modules. Since the original module is known and each of the replications marked as replications, the operating system creates one new module rather than a number of new modules. After replication, the system tests the data load to determine if sufficient performance is achieved. The system grows

or shrinks linearly rather than exponentially. Generally, a linear growth is more stable in responding to changing computing demands.

E. TERMINAL INTERFACE.

1. General. Figure 26 shows a complete programming environment. The terminals are interactive and correspond with programs through the terminal interface. The environment consists of two programs, in this example, each consisting of modules and each performing different tasks. Each program produces printed output and is connected to an operating system module labelled PRINTER. This module reflects the existence of a real device. Each program writes data to a port, P in this example, which represents a printer. The operating system controls printing in the same way current operating systems handle virtual printers. Similarly, program 2 supplies plotter commands through port C. The system module spools the data and routes it to the appropriate plotter. Modules supply data to ports whose paths connect to operating system modules representing the desired devices. Communication with real devices is thus accomplished through path connections to operating system ports.

Each program communicates with a user terminal through the terminal interface. This port is designated as A in both programs of Figure 26. Unique port names are unnecessary since the terminal interface contains a

directory of program ports. The path, if connected between a module and a port at the program boundary, as in port A of Figure 26, transfers data in the same fashion as if the path were connected to the ports of interior modules. Buffers exist between the terminal interface and the program ports. Data transfer between modules and the terminal interface is accomplished in the same manner as data transfer between modules within a program. When a program module writes data to a port connected to the terminal interface, the data is held in a buffer at the interface until it can be processed. The concept is similar to data transfer between modules. The data written to a program from the terminal interface is treated like the data which comes from another module. The program boundary is not considered in this data transfer.

The terminal interface creates a desired environment for the user. The control modules existing in the terminal interface can be modified to become customized versions. The user can, by writing his own control modules, create a personal environment.

2. Private, Semi-Private, Public Messages. When the user establishes communication with the computer, he provides a data supply name. This name determines the port name that will receive the messages originating at this terminal. The user can have one supply name active

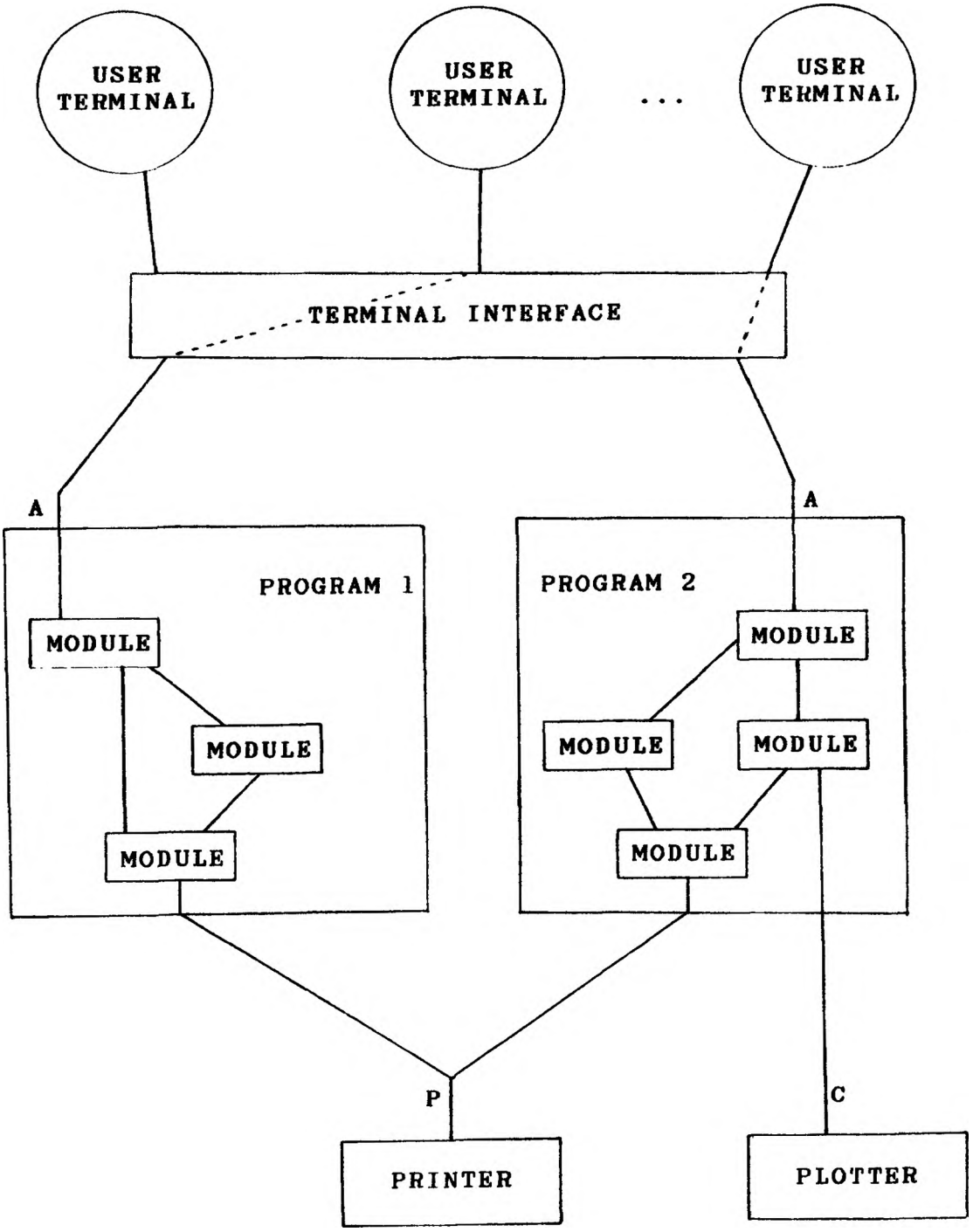

FIGURE 26
COMPLETE SYSTEM ENVIRONMENT

at his terminal at a time, but can change the name as necessary to establish communication with other modules. Only module receiving ports whose names correspond to the supply name of the terminal will receive the terminal's messages. The supply name determines which modules will receive the terminal's messages.

The user also provides a data receiving name. This name determines which messages will be directed to the user's terminal. The terminal interface intercepts all messages to determine proper routing indicated by the terminal's data supply and data receiving names. Messages received by a terminal may be of three types: priviate, semi-private and public. When private messages are being received, the terminal interface routes the message to only one terminal. A private dialogue is taking place between a program and the terminal. The operating system, at the terminal interface, appends a user name to all messages originating at a terminal. If the program replies to a particular message, it includes the name with the message. When the name is present, the terminal interface routes the message to the appropriate user. If a user has left the system, the message is queued for delivery upon his return. Errors in data entry, for example, are returned to the original source through this facility.

Semi-private messages occur when messages of a particular nature are transmitted. A special designation is included with the message to cause routing to all terminals in a common mode. For example, the notification of depleted inventory is sent to all terminals placing orders as identified by their data supply and receiving names. Public messages are sent to all active terminals. A public message contains no name. A message of "system going down" is an example of a public message.

The user, when establishing communication with the computer, supplies the data supply name and the data receiving name. These two names categorize the user as a particular type. When semi-private messages occur, the type is checked to determine which users will receive the message.

All three message types originate in program modules. The message type indicates which terminal or group of terminals receive the message. All messages originating at a terminal are routed according to the data supply name, thus, one or more modules receive the message.

F. DEVICES.

Data transfer to devices that are common to all programs is supported. These devices are characterized by the fact that they do not carry on a dialogue as do the terminals since communication is generally in one direction. Printers, plotters, and card readers are examples of these devices. Since a dialogue is not required, a complex interface is unnecessary. The path of a program's port is connected to special operating system ports that represent these devices. Port P in Figure 26 is an example of this port. Each program in the system writes data to be printed to a port that has been connected to an operating system module. Data to be printed is routed from the module to a port on the program boundary. The operating system routes the data at port P to the printer, retaining proper order of the data. Data is buffered on each path connected to port P until a special marker appears. At that time the data is given to the printer. No data interleaving takes place on these paths. The order of the data printed is retained. This concept is similar to conventional spooling which takes place in current operating systems. This communication presents no new problems for the user. A user program communicates with a port and the operating system provides support for the desired facility.

G. OPERATING SYSTEM LEVELS.

A high-level component of the operating system is characterized by providing general support of all programs in the system and a lower level component of the operating system is characterized by providing local support of modules within a program. One instance of the high-level operating system exists and several instances of the low-level system, one for each module, exist. The higher level of support includes communication with the terminal interface and external devices. The replication of modules is included in the higher level of support. The instance at the higher level monitors the operation of the complete system. Low-level support is characterized by the operations that take place in support of each individual module. The evaluation of predicates upon data arrival and data transfer from a port's buffer to the module are examples of this. Conceptually, this portion of the operating system is distributed to each module.

The higher level of the system supports devices outside of the program environment. Data transfer support is an example as shown for printer and plotter modules in Figure 26. This support is at the higher level of the operating system since it is used by all executing programs. The support retains data order required at the printer or plotter receiving modules.

H. PROCESSOR ASSIGNMENT.

The implementation of the programming environment is not dependent on the number of processors available. If one processor exists, the program can be executed by assigning the processor in turn to each module that has its trigger satisfied. If two processors exist, each is assigned to modules as triggers are satisfied. Only the amount of concurrent program execution is dependent on the number of processors available. Algorithm development can proceed independently of the actual hardware being used to implement the system. Consideration is given for processor assignment to modules by the following criteria: [24]

1) Intermodule Communication - two modules known to communicate heavily should be assigned processors concurrently to increase system performance.
2) Accumulative Execution Time - a module which commonly executes for a longer time than average module execution time should have a higher priority. An avoidance of bottlenecks is possible by assigning processors to high execution time modules.
3) Precedence Relationship - certain paths in a program network can have a priority assigned to them. Modules connected to the higher priority

paths have processors assigned before modules with lower priority paths.

I. SUMMARY.

Messages passed between modules perform the task of: 1) satisfying activation criteria of the receiving module and 2) carrying data so that an instance of a problem can be solved by the program. The operating system provides transparent support to the module. This support consists of message switching, module suspension and activation, module replication (with permission), terminal support through the terminal interface, support for external devices, and resource allocation to modules. Most of the support remains transparent to the user. Modules are self-contained and require no status information of the surrounding environment for proper execution.

The operating system support is performed at two distinct levels. A higher level of support performs a supervisory role over the complete computing system. Data transfer between the terminal interface and modules as well as between the modules and devices is a service of the higher level of support. One instance of the operating system exists at this level. At a lower level, the system provides support for individual modules. With assistance of the higher level, the lower level performs data transfer between modules' ports. The evaluation and support of triggers is performed at this level.

Suspension of modules is performed by the operating system at the request of the module or by the operating system without a request if preemption is warranted. Demand is determined by the quantity of data messages waiting to be processed by a module. When the module finds that it has processed all of the available incoming data, it writes new predicates to the operating system and issues a "SLEEP" request which allows the module to be suspended. The module does not leave the system, but allows resources to be reallocated. The operating system, when it determines a module is doing little processing, can cause a module to be suspended. The module does not issue a "SLEEP" request, nor are new triggers given. The complete module environment is saved. Processor reassignment is performed when message accumulation is sufficient. This concept is similar to the multiprogramming environments of today's computers, but depends on data presence rather than I/O waits and timer interrupts.

With permission, the operating system is capable of replicating a module to enhance performance. The surrounding modules are not affected by the replication. The operating system is capable of evaluating the load placed on modules to determine the need for module replication or deletion. The number of modules grows or shrinks linearly as required by computing demands and as resources permit.

The user can modify the terminal interface to create a desired terminal environment. User modification of the terminal interface allows the creation of an environment suitable for each application. The computing environment is flexible to meet the needs of a variety of users. All messages between the user and program modules pass through the terminal interface. Private, semi-private and public messages are supported.

Printers, plotters and other devices are supported by the system. The transfer of data to these devices is convenient since the programmer includes a path to a system port on the boundary of the program designated as the desired device. The program boundary is necessary for this designation, since the operating system recognizes the port names on the program boundary as device names.

Determination of processor assignment can be performed according to a variety of schemes, however, data message presence conditions expressed by triggers of suspended modules are the primary determinant. Priority assigned to modules or paths can be used to aid in determining the binding of processors to modules.

IV. PROGRAMMING IMPLICATIONS

A. INTRODUCTION.

A new concept in parallel algorithm development is being described. To reinforce the concept, a number of programming considerations and examples are presented in this chapter. The examples demonstrate capabilities of the system while exemplifying the programming vehicle design. Such features as module nesting, module replication by the programmer, the representation of files, module testing and verification and common module examples are used to demonstrate programming implications. The use of common modules shows a higher level of programming, a level seldom afforded in data flow languages. As users become more familiar with the vehicle, more common modules will evolve. Nesting and replication aid the application programmer during program development by allowing modules to contain other modules and by allowing the same module to be used multiple times. Each feature enhances program readability and reduces program development time.

B. REPLICATION BY THE PROGRAMMER.

The vehicle allows module replication in two different ways. First, replication is performed by the operating system as described in Chapter 3, Section D.

Module replication by the operating system improves system performance as required by data presence. Second, a programmer can explicitly include many instances of a module.

Several reasons for desiring replication exist. 1) Identical modules can perform similar functions on distinct data items. 2) Modules are replicated to build redundancy into the program. 3) Improved system performance is realized as a result of replication. Replication is accomplished by the inclusion of desired modules in the program, connecting data supply paths and data consuming paths in the same manner as nonreplicated modules.

1. Identical Function Replication. Figure 27 shows module A replicated by the programmer. The program is a segment of the sample program shown in section F on household expenses. For example, path 1 provides expense data for house maintenance and path 2 provides expense data for car maintenance. The processing required is identical in both cases so that the same module is used. Module B performs total expense summary, hence data from both modules is routed to module B. Only one module is developed to perform the maintenance computations. By including the module many times, program efficiency as well as programmer efficiency is increased.

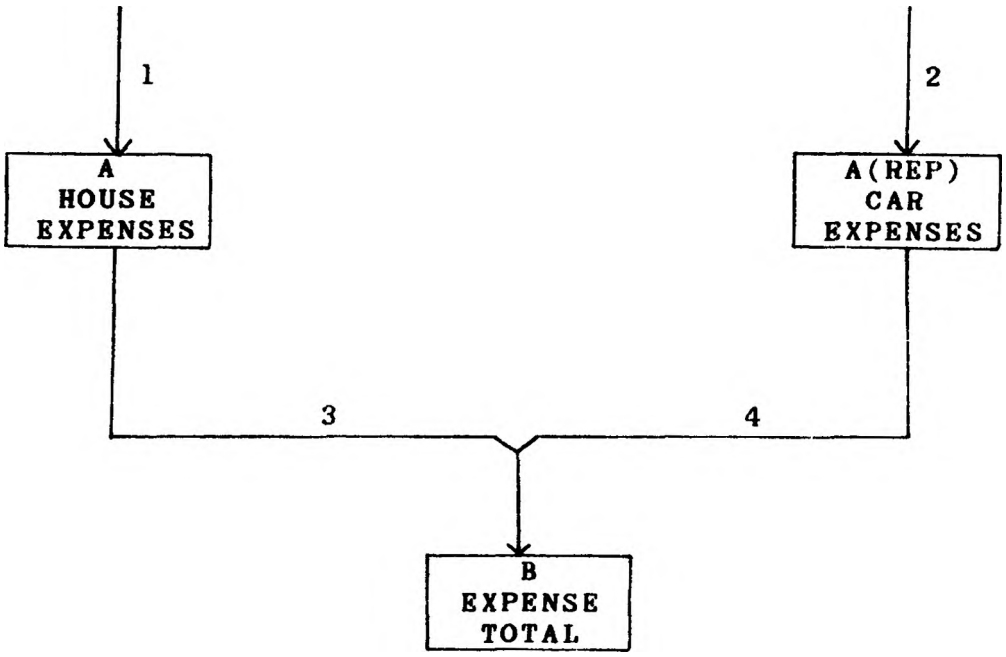

FIGURE 27
MODULE REPLICATION

2. Ordered Data Supply. In the example provided by Figure 27, data is consumed by module B in the order of arrival. Since module B is an accumulator, no problems are encountered. A potential problem exists when the arriving data cannot be mixed, when replication of module A requires replication of module B. Data streams that cannot be so mixed are called order-sensitive.

Figure 28 shows the effect of replication when an order-sensitive data flow is involved. Module B (REP), which is a replication of module B, is included as a result of module A's replication. Data order is retained since each module A supplies data to a distinct module B. The operating system does not determine necessary replications (module B) that result from replications (module A) occurring in a program network. The programmer must recognize the need to replicate receiving modules upon replication of supply modules. Order-sensitivity must be determined by the program designer.

3. Redundancy Replication. Figure 29 shows an example of module A being replicated to module A(REP). Modules A and A(REP) are identical and manipulate the same data. Redundancy provides reliability and reduces the probability of erroneous data entering computations. Data on path 1 and path 2 is identical and fanned-out from the supplier.

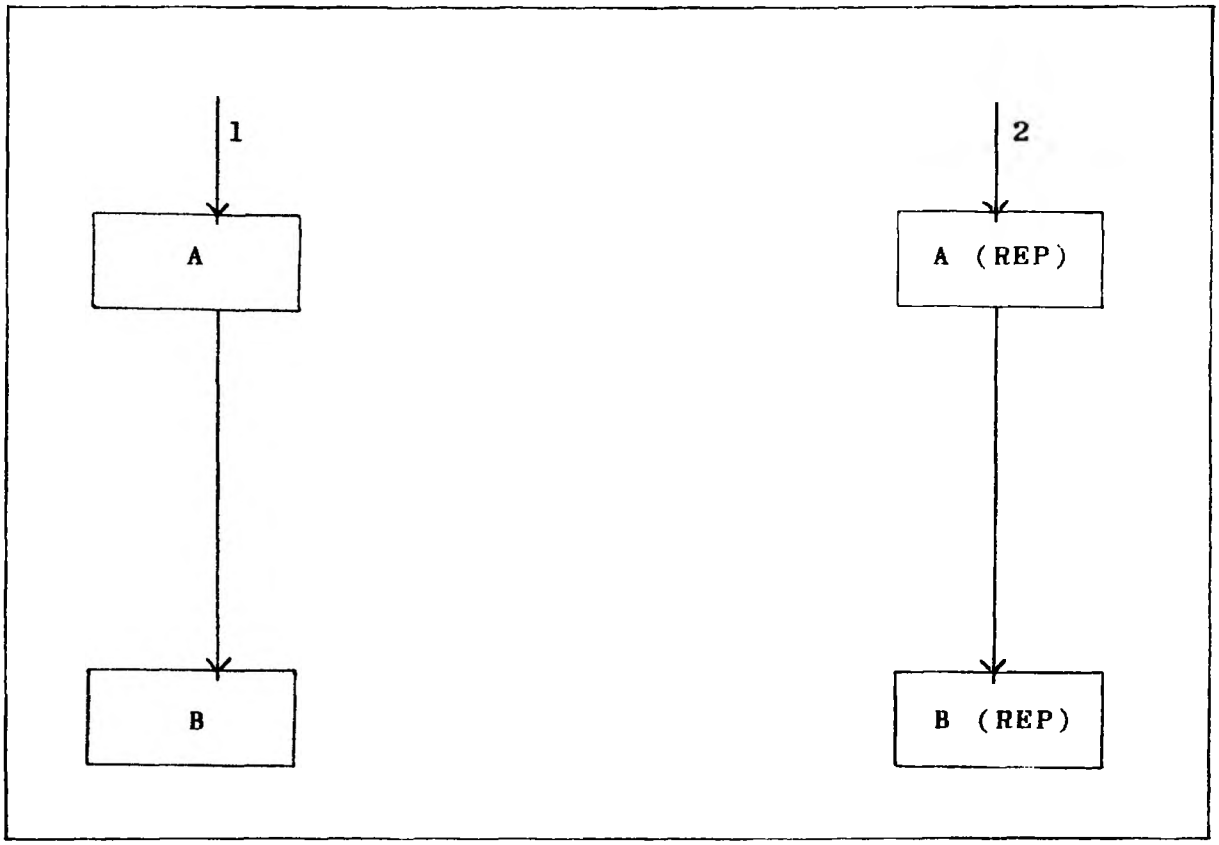

FIGURE 28
MODULE DESTINATION REPLICATION

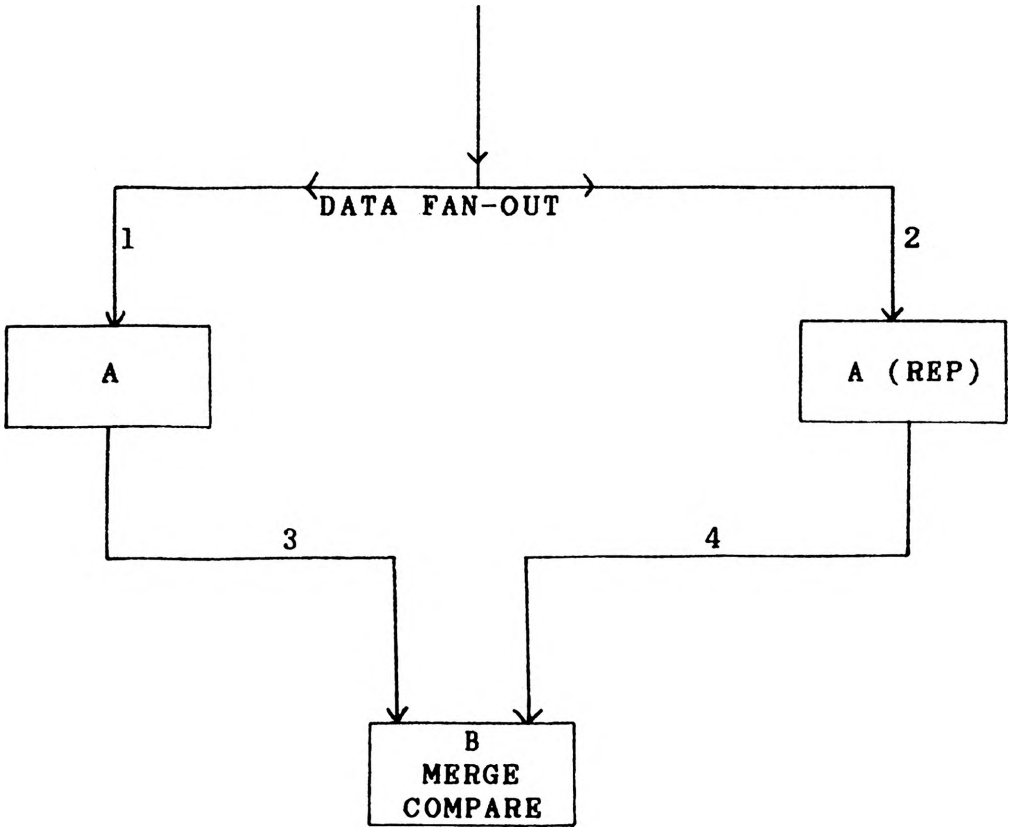

FIGURE 29
DATA REDUNDANCY

Data supplied by these two modules is merged and compared by module B. Data reliability is enhanced by this module replication. If data inconsistencies are recognized by module B, corrective measures are taken as actions of that module.

Redundancy is a programmer-designed feature included in this application, so data synchronizing is performed within the modules, for example, module B in Figure 29. Redundancy is a natural extension to the vehicle design and allows duplicative processing of critical data. By including redundancy, not only is data integrity enhanced, but reliability is also improved. If a processor associated with module A should fail, the system halts with a message indicating a discrepancy. Data checking is lost since only module A(rep) supplies data to module B until module A is reactivated.

4. Module Testing. As shown in Figure 29, modules are easily replicated to enhance reliability. Redundancy, to any level desired, can be included through replication. As with redundancy, testing of modules can also be included naturally within the vehicle. A new module may be placed alongside an existing module with supply lines fanned to each module. The corresponding data produced by the new module can be tested without interrupting original data processing.

Figure 30 shows an example of a new module B included in the program. The data is copied into the new path for

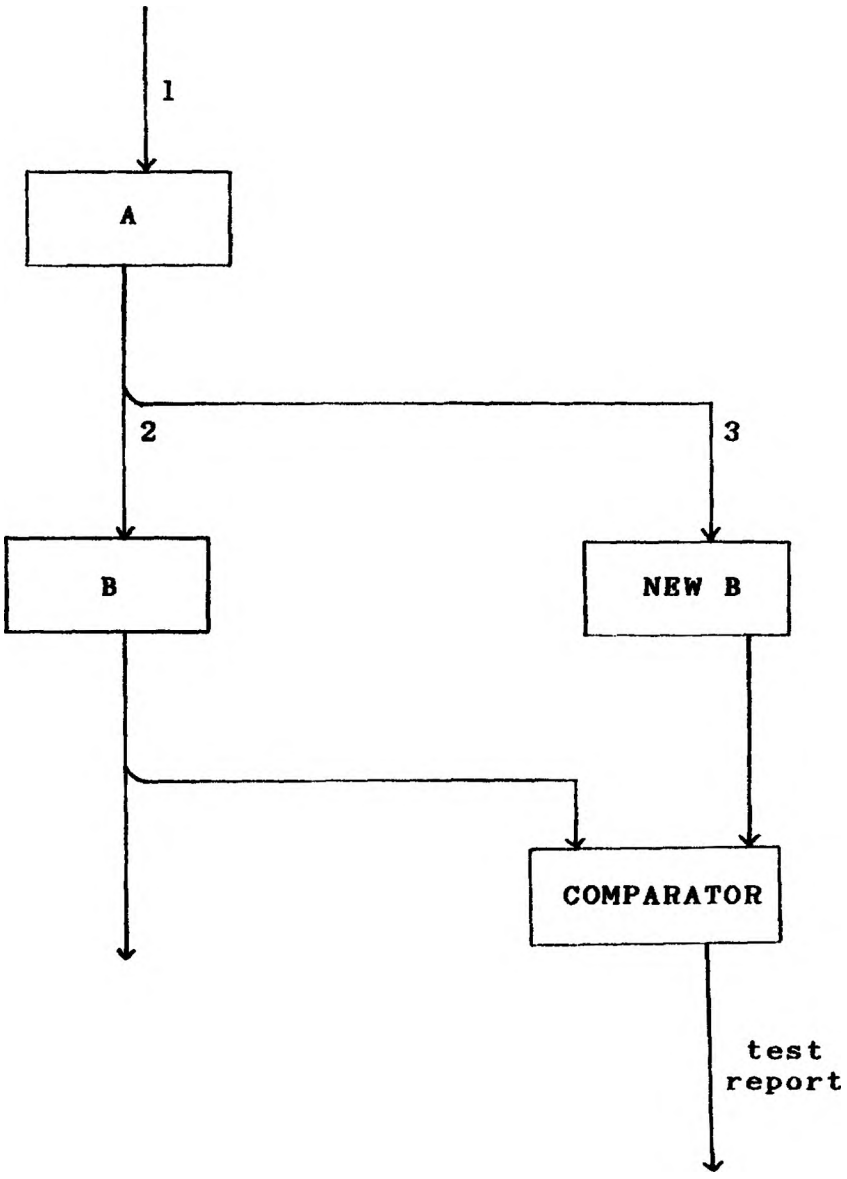

FIGURE 30
MODULE TESTING

test module B by the operating system function of fan-out already in place. Since the original module B is undisturbed, the new test module B can be studied without interrupting processing. The comparator module is used to check the results of the original module and the new module producing a report of data accuracy.

C. NESTING.

1. General. A module has been described as a unit expressed in terms of conventional programming instructions. A module may also be comprised of other modules. Modules placed within a module are said to be nested. Nesting of modules is consistent with top-down structured program design. The vehicle developed allows the top-down design to be included in the final program specification. Readability and reduced development time are enhanced by this feature.

2. Specification Of Nesting. Nested modules enter the system as shown in Figure 31. Modules X and Y are previously defined modules. The data descriptions associated with module XX describe the data requirements of ports A, B and C. These data requirements correspond to ports D, E and H, respectively. Modules X and Y are known to the system and are low level modules or modules consisting of other modules.

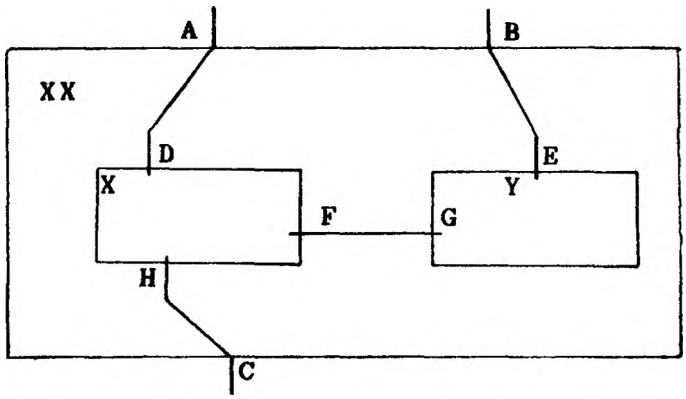

FIGURE 31
NESTED MODULES

Module XX is included in program graphs as shown in Figure 32. Associated with the module is the description of its function, a description of data supplied and data consumed. The descriptions are similar to those for non-nested modules.

The outermost module is specified in the program. The system, through the module description, includes the nested modules provided that the nested modules have previously been entered in the system. The outermost level module is included in the program graph. When the nested modules are new, the programmer must specify the complete make-up of the module network. In general, a module is included in a program graph by specifying the module name and connecting paths to ports. The module which is comprised of other modules is included in a program similarly. The composition of the module does not change the inclusion procedure.

Nested modules are included by the operating system through the description of the outer module. A module is described by supplying four components: a description of the function performed by the module, a description of the data supplied, a description of data consumed, and the instructions of the module. The descriptions assist the programmer in determining which previously developed modules to include in the program. The instructions direct the computer to perform the desired computations. Each module consisting of nested modules is given by the

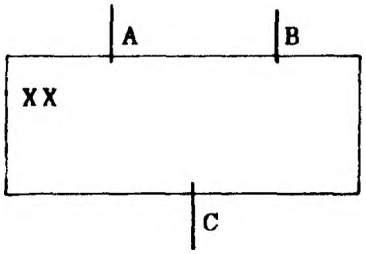

FIGURE 32
USAGE OF MODULE

components: a description of the function of the module, a description of the data supplied at each port, a description of data consumed at each port, and a specification of the internal structure of the module in terms of modules, paths and ports. A list of instructions is necessary only at the lowest level module.

The replication factor is used when a module is repeated in a program. Modules replicated by the programmer are used to perform the same function on different data. The name and replication number uniquely identify each module in the system. In Figure 33, the module AAA is used twice. Both modules perform the same function on different data. AAA(1) and AAA(2) uniquely name each module so that the appropriate internal port to external port mapping can take place. The module name XXX is used in program networks as if it were a low level module as in Figure 32. The description of module XXX consists of the data consumed (ports A, B, C, and D), data supplied (port E) the functional description, and the structure Figure 33 depicts.

3. Activation And Replication Of Nested Modules. Modules containing other modules are not directly considered for activation. There is no trigger associated with module XXX in Figure 33. The internal modules become active due to data presence condition that satisfy their triggers. Activation is only considered for the lowest level modules containing program instructions.

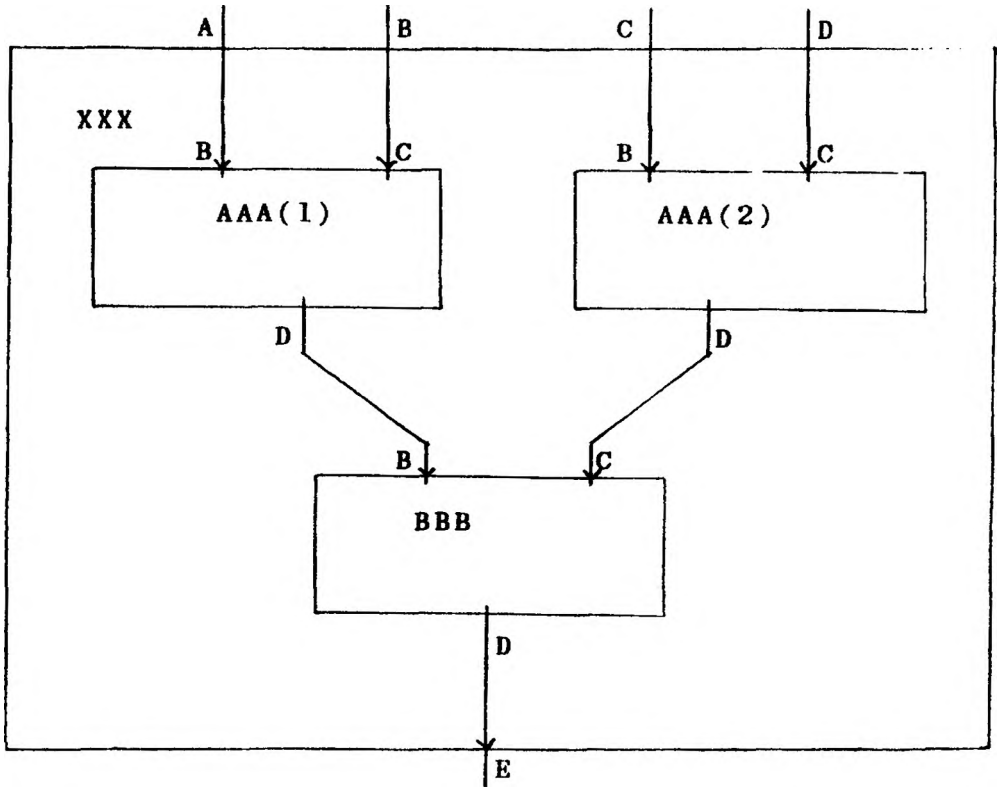

FIGURE 33
NESTED MODULES

All modules marked by the programmer are replicatable by the operating system. Nested modules, originally nonreplicatable, may be replicated when nested. Suppose BBB is a nonreplicatable module in Figure 33. Data consumption and supply may allow XXX to be replicated. The replication of XXX indirectly replicates BBB. The replication of BBB in this context is permissible even if BBB is marked nonreplicatable. The programmer considers each module individually within its environment to determine replication permission. A module at one level that is not replicatable may become indirectly replicatable when nested.

The nesting of modules makes programs more readable and is consistent with top-down structured program design. By allowing modules to contain other modules, the specification of a program graph is simplified. Program development time is reduced when previously developed modules are used.

D. FILES.

Data files can exist in the programming environment in three forms. A file can be inside a module, supported by the operating system and connected to a module, or exist outside of the program.

1. Modules As Files. A file that is inside a module appears and performs like any other module. A module, acting like a file, generally accepts messages calling for alteration of the file's content and also responds to inquiries. The use of previous modules described and a module acting like a file is only a small concept change in module usage. The role of traditional modules is to manipulate data. Data storage exists only in support of the manipulations. Modules acting as files primarily hold data. Realizing that technically there is no difference between data and program, the point is one of the role a module plays in the program graph as used by the program designer. Data security is high for these modules since they exist within a program. The only access to the data is through the paths and ports associated with the module. File maintenance is difficult with this type file since all updates are passed through ports.

2. Module Associated Files. Figure 34 shows a file connected to a module. The file is fully supported by the operating system; that is, the module supplies data to port A and consumes data from port B. Data is read and written by the same module. The operating system maintains the path in the same manner as paths which exist between modules. The module can "page" through the data by reading from port B and writing data to be "filed" at port A. Files stored in this way are processed

sequentially. An EOF marker is written into the path marking the end of the file. The marker is used to determine when the entire file has been examined. The security of this file is high since the only module capable of accessing the data is the module to which the file is attached. Other modules must request or receive data through other ports associated with the module, not from the file itself. Information hiding and request validation are performed by the module associated with the file. File transactions must be routed through the associated module since the file is physically stored as a queue of messages on a path. The natural use of this mechanism is for small or temporary files that are processed sequentially.

3. External Files. Files may also exist outside of a program environment. In an external file, the file exists outside of the PROG 1 environment. Accesses to the file are through port E as shown in Figure 35. Traditional file support is provided by the operating system for files of this type. Direct access, indexed sequential, or any other access method can be supported for this file. Modules access the file by communicating with a specified port, port E in Figure 35. A path has been specified which links port E with the globally-named system supported file. Many programs may have access to the file, thereby reducing the level of security.

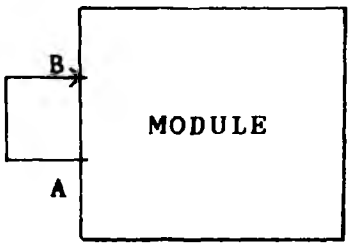

FIGURE 34
MODULE ASSOCIATED FILE

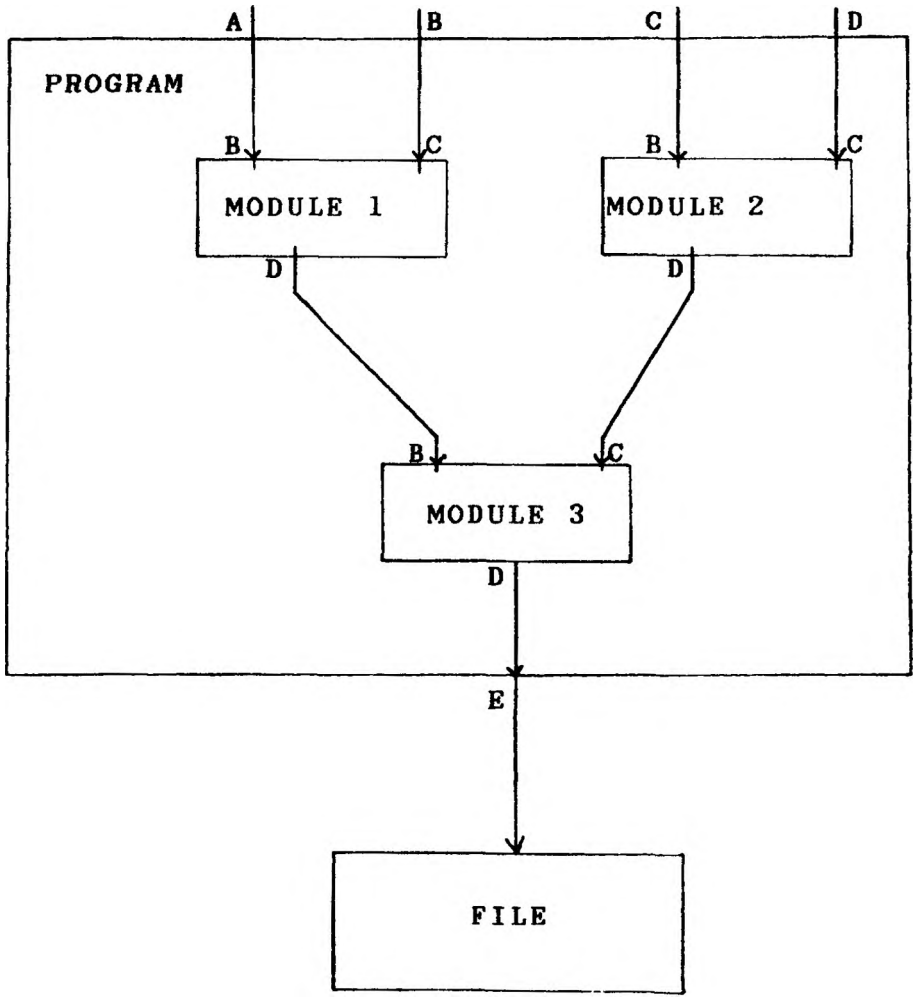

FIGURE 35
EXTERNAL FILE

Figure 35 shows one program accessing the external file. In common application, many programs will be allowed file access. The operating system must resolve common problems such as simultaneous updating and accessing of data. Many traditional problems exist when files of this type are used. Advantages are realized with this file organization since many programs can access the data. Data important to a variety of applications can be held in the external file. Each program has access to the file directly provided by the operating system.

Programs communicate with external files through dedicated system ports. Each program wishing to communicate with the file will contain paths to a port similar to port E in Figure 35. The program supplies data to or requests data from a known port. The operating system controls the flow of data to and from the file. Program communication with the external file exists in the same way communication with any internal port in the program environment takes place. Since the operating system is the only facility communicating directly with the file, traditional problems of file accessing are solved in a similar fashion as solutions in current computing systems.

E. COMMON MODULES.

From the experience of programming in the described environment, some common modules are emerging. The common modules are generic forms that are developed for use within many programs.

1. Keep Sorted. GENERAL DESCRIPTION: The module is called KEEP SORTED. Its function is to maintain a sorted file. Data received is inserted in the ordered position within the file.

DATA DESCRIPTION: Data records are received at port A. New records are written at port B and record retrieval occurs at port E. Upon request, records of the entire file are supplied through port D.

Notice, in Figure 37, that no new triggers are written when the module's sleep request is executed. The initial triggers provided at module entry are used throughout the module's existence. The module is a candidate for activation when its trigger is satisfied. During each activation at ENTRY_ONE, the file is updated. The parallel nature of the computing environment along with the continual presence of the module allows the file to remain current. Thus, each request for data will yield all current records. Naturally, a more efficient but far more complex version of this module can be crafted.

The concept presented in this dissertation of a new programming vehicle for expression of parallel algorithms is typified by the KEEP SORTED module shown in Figure 36 and described above. File updates are typical of many computer applications. Traditional updating routines collect data in a file in random order until a program is released to perform a sort as typified by Figure 38. The sort may require significant time during which the user waits for results. In the new programming environment, the data is supplied to the module through port A (Figure 36). Data is inserted in its proper position by reading from port E and writing to port B. When more data appears at port A, it is inserted in the correct place. Thus, the KEEP SORTED module is always present and potentially active. The data is sorted as it is collected. When a request occurs at port D, all available sorted data is written. Very little wait time will occur since the data had been sorted as it arrived. The KEEP SORTED module can be activated during slack computing time so that better utilization of resources is possible. If it is critical that the records be kept current, the module can be active at all times.

The KEEP SORTED module, as applied to a file, contrasts with sequential file updating in Figure 38. Processing takes place as a result of an operator releasing the programs for execution. Problems exist when access requests for records arrive during the updating

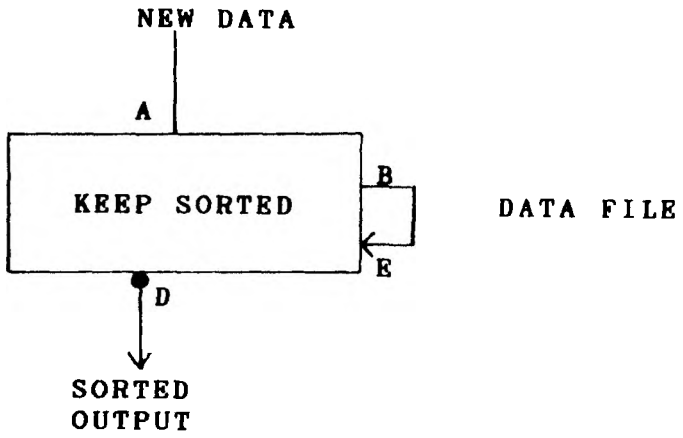

FIGURE 36
KEEP SORTED

```
     MODULE KEEP SORTED:

1    MODULE TRIGGER:  TRIG ONE: A, ENTRY_ONE;
2                          TWO: D AND NOT A, ENTRY_TWO;

3    MODULE PSEUDO-CODE:

4         ENTRY_ONE:
5              READ FROM PORT A TO VARIABLE NEW_DATA.
6              IF NO DATA AT PORT E WRITE EOF TO PORT B.
7              READ FROM PORT E AND WRITE TO PORT B UNTIL
                    POSITION FOR NEW_DATA IS FOUND.
8              WRITE NEW_DATA TO PORT B.
9              WRITE DATA FROM PORT E TO PORT B UNTIL EOF
                    IS WRITTEN.
10             SLEEP.
11        END ENTRY_ONE.

12        ENTRY-TWO:
13             READ FROM PORT D.  (* REMOVE THE REQUEST *)
14             READ FROM PORT E AND WRITE TO PORT D UNTIL
                    THE FILE IS EXHAUSTED.
15             WRITE EOF TO B.
16             SLEEP.
17        END ENTRY_TWO.

18   END MODULE KEEP SORTED.
```

FIGURE 37
KEEP SORTED PSEUDO-CODE

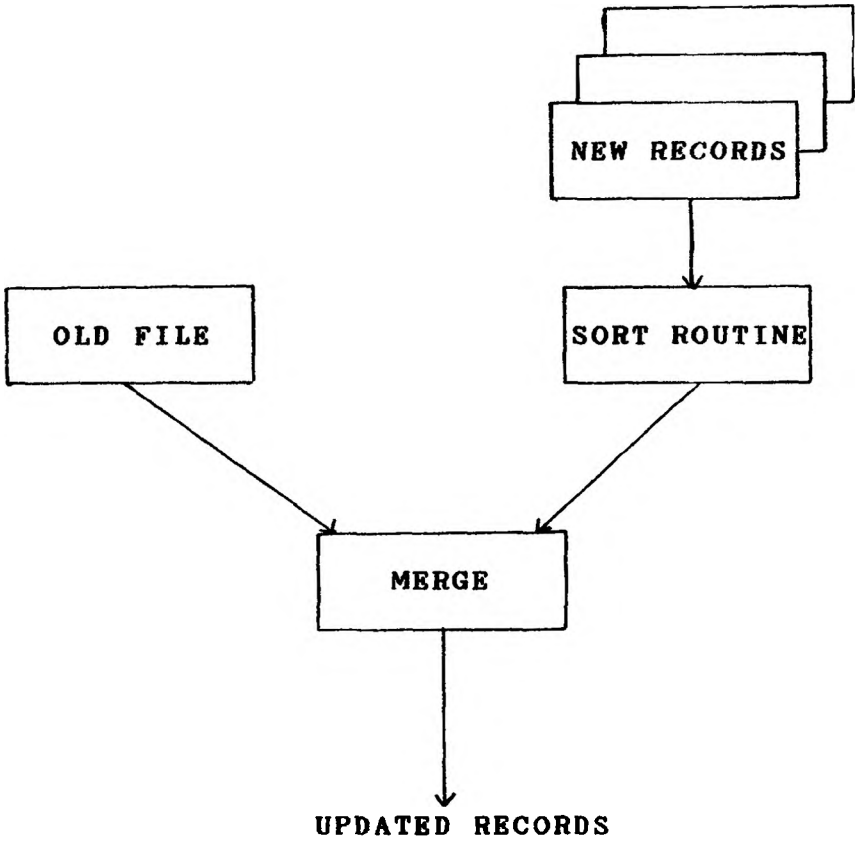

FIGURE 38
TRADITIONAL SEQUENTIAL FILE UPDATE

process. Generally, the file is inaccessible during this procedure. New records are queued, waiting for the update process to occur. Access to new records is difficult preceding the updating process.

The keep sorted concept overcomes the above problems. New records are processed immediately and thus are available when needed. By keeping the update facility active, the version of the file is always current.

A MERGE module is also common. Figure 39 shows a use of the merge. Ordered data streams are supplied to two ports, A and B. Depending on a specified criteria (numerical or lexicographical ordering, for example), the data is merged into one ordered data stream and supplied through port C. The module is always resident and begins execution as indicated by data presence.

2. Merge. GENERAL DESCRIPTION: The module is called "MERGE". Its function is to combine two ordered data streams into a single ordered stream.

DATA DESCRIPTION: The module is activated when data or the EOF marker is present in the paths connected to port A and port B. When data is present in each path, the module is a candidate for activation and a data record can be provided to port C as determined by the order criteria in the module. Data is consumed, one record at a time, from the appropriate port. If the record from port A is written to port C, the data supply port, then port A is read next. Port B is read similarly when data from port B

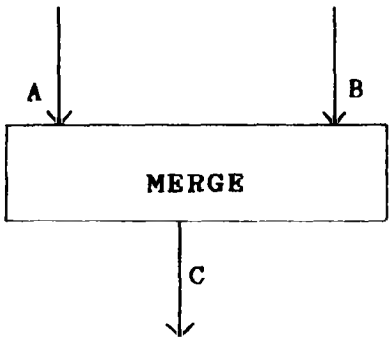

FIGURE 39
MERGE

is written to port C. When a path is exhausted, the data at the other port is passed to port C. Port C is the data supply port where all output is written. The pseudo-code is shown in Figure 40.

The module performs a common merging of two data streams. The EOF marker is removed from the paths by the operating sytem when the module is triggered on the EOF; that is, the EOF marker is implicitly removed when a predicate like EOF A is true and the corresponding entry is taken. The modules supplying the data streams are required to write the EOF marker in each path to separate instances of data streams. Merge writes EOF to port C after EOF is found at port A and port B.

The merge module captures the flavor of programming in this environment. Merge can be activated when there is data ready to be processed rather than wait for a complete file to be available. By using a variety of trigger configurations, many data presence conditions allow module activation. Figure 41 shows the conditions present that allow control to be passed to a particular entry point. The lines indicate which entry point can receive control if the condition on the line is satisfied. The trigger determines which of the conditions are true. Rectangles name the entry point where control is passed depending on data presence. The comment in the rectangle tells the action performed in the module by the entry. The diagram

```
     MODULE MERGE:

1    MODULE TRIGGER:  TRIG ONE: A AND B, ENTRY_ONE;
2                     TWO: A AND EOF B, ENTRY_TWO;
3                     THREE: EOF A AND B, ENTRY_THREE;
4                     FOUR: EOF A AND EOF B, ENTRY_FOUR;

5    MODULE PSEUDO-CODE:

6         ENTRY_ONE:     {Ports A and B have data}
7              READ PORT A INTO DATA 1
8              READ PORT B INTO DATA 2
9              GO TO NINE
10        END ENTRY_ONE.

11        ENTRY_TWO:     {Data at port A, EOF at port B}
12             READ PORT A INTO DATA 1
13             WRITE DATA 1 TO PORT C
14             SLEEP: NEW TRIGGER:
15                  TRIG ONE: A, ENTRY_TWO;
16                       TWO: EOF A, ENTRY_FOUR;
17        END ENTRY_TWO.

18        ENTRY_THREE:   {Data port B EOF at port A}
19             READ PORT B INTO DATA 2
20             WRITE DATA 2 TO PORT C
21             SLEEP: NEW TRIGGER:
22                  TRIG ONE: B, ENTRY_THREE;
23                       TWO: EOF B, ENTRY_FOUR;
24        END ENTRY_THREE.

25        ENTRY_FOUR:   { EOF at port A and EOF at port B}
26             WRITE EOF TO PORT C
27             SLEEP: NEW TRIGGER:
28                 TRIG ONE: A AND B, ENTRY_ONE:
29                      TWO: A AND EOF B, ENTRY_TWO;
30                      THREE: EOF A AND B, ENTRY_THREE;
31                      FOUR: EOF A AND EOF B, ENTRY_FOUR;
32        END ENTRY_FOUR

33        ENTRY_FIVE:   {Read data at port A, data in B}
34             READ PORT A INTO DATA 1
35             GO TO NINE
36        END ENTRY_FIVE
```

FIGURE 40
MERGE PSEUDO-CODE

```
37        ENTRY_SIX:  {EOF at port A, no data at A,
                       data in data 2}
38              WRITE DATA 2 TO PORT C
39              SLEEP: NEW TRIGGER:
40                    TRIG ONE: B, ENTRY_THREE;
41                         TWO: EOF B, ENTRY_FOUR;
42        END ENTRY_SIX

43        ENTRY_SEVEN:  {Data at port B, data in data 1}
44              READ PORT B INTO DATA 2
45              GO TO NINE
46        END ENTRY_SEVEN

47        ENTRY_EIGHT:  {EOF at port B, no data in data 2
                         data at A}
48              WRITE DATA 1 TO PORT C
49              SLEEP: NEW TRIGGER:
50                    TRIG ONE: A, ENTRY_TWO;
51                         TWO: EOF A, ENTRY_FOUR;
52        END ENTRY_EIGHT

53        NINE:  {do the comparison}
54              IF DATA 1 < DATA 2 THEN BEGIN
55                    WRITE DATA 1 TO PORT C
56                    SLEEP: NEW TRIGGER:
57                          TRIG ONE: A, ENTRY_FIVE;
58                               TWO: EOF A, ENTRY_SIX;
59                    END BEGIN
60              ELSE BEGIN
61                    WRITE DATA 2 TO PORT C
62                    SLEEP: NEW TRIGGER:
63                          TRIG ONE: B, ENTRY_SEVEN;
64                               TWO: EOF B, ENTRY_EIGHT;
65              END BEGIN
66        END NINE

67    END MERGE
```

FIGURE 40 (continued)
MERGE PSEUDO-CODE

ENTRY CONDITIONS:

ENTRY_ONE: DATA AT A AND B
ENTRY_TWO: DATA AT A, EOF AT B
ENTRY_THREE: EOF AT A, DATA AT B
ENTRY_FOUR: EOF AT A AND B
ENTRY_FIVE: DATA AT A
ENTRY_SIX: EOF AT A
ENTRY_SEVEN: DATA AT B
ENTRY_EIGHT: EOF AT B

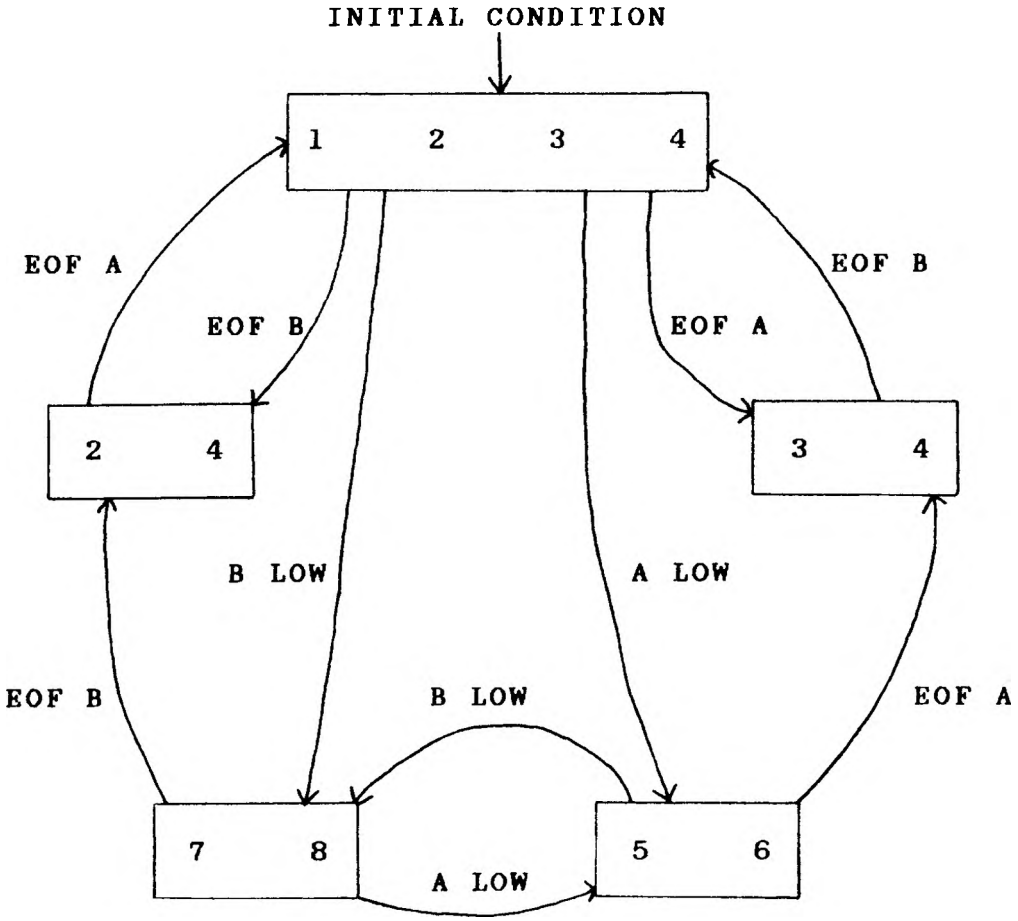

FIGURE 41
MERGE LOGIC DIAGRAM

shows the use of triggers to determine data conditions which activate and control data processing by the entry point within the module. Through the use of triggers a wide variety of data conditions can exist. The dynamic triggers, module presence, and activation determined by data presence is shown by the Merge module and exemplifies the basis of this style of programming.

F. PROGRAM EXAMPLE - ORDER PROCESSING PROGRAM.

This section provides an example that can be followed to create other program systems. The program processes data associated with a wholesale supplier. Orders are received, payments received, inventory levels maintained and reports generated. Programming features such as path fan-out, path fan-in, common module usage, duplicate modules, multiple entry points to a module, parallelism, triggers and request queuing are demonstrated in this example.

GENERAL PROGRAM DESCRIPTION: The program is called Order Processing. Merchandise orders are received, customer accounts updated, inventory updated and reports generated through the program network of Figure 42.

1. Customer Verification. GENERAL DESCRIPTION: The module is called Customer Verification. Its function is to receive new orders, check the customers standing with the Customer Master File module and remove the rejected orders from the system. A qualified customer

order is passed to other modules upon satisfaction of the check. New customer orders are returned to the account maintenance portion of the program.

DATA DESCRIPTION: The module consists of data receiving ports, A, C, F and data supply ports, B, G, D, E. Port A receives new orders. A customer order whose account is not present in the Customer Master File module is passed to Account Maintenance through port B. Port C requests the customer master record to determine account status. Port D writes orders successful in the previous checks to other modules. Ports E and F are used to hold orders while customer verification is obtained. Port G writes customer orders whose account status causes the order not to be filled. The pseudo-code is shown in Figure 43.

2. Customer Master File. GENERAL DESCRIPTION: The module is called Customer Master File. Its purpose is to maintain customer records and respond to requests. Each customer associated with the program has a record in the module. All relevant information related to a customer's account is retained in the module.

DATA DESCRIPTION: Ports A and B communicate with customer maintenance. They receive new customers and satisfy customer inquiries, respectively. The current state of an account is maintained in the module. Payments and credits are credited to an account from data received

through port C. Data received through port D represents charges against an account. Requests for account status are received and supplied through port B. The pseudo-code is shown in Figure 44.

3. Distribute. GENERAL DESCRIPTION: The module is a common module used for receiving data of a particular type, translating data and passing it to the receiving modules. The module is called Distribute.

DATA DESCRIPTION: Port A is the data receiving port and receives payments or credits. If the data is a payment, the credit is written to port B (to pass the payment to the bank), to port C (accumulation of payments received by day, month, quarter and year), and to port E (credit the appropriate account). If the data is "returned merchandise", the inventory item is written to port D (to update inventory), and to port E (to credit the account for the returned merchandise). The credit is not reflected in daily payments or to the bank since only cash flow occurs in these paths. The module pseudo-code is in Figure 45.

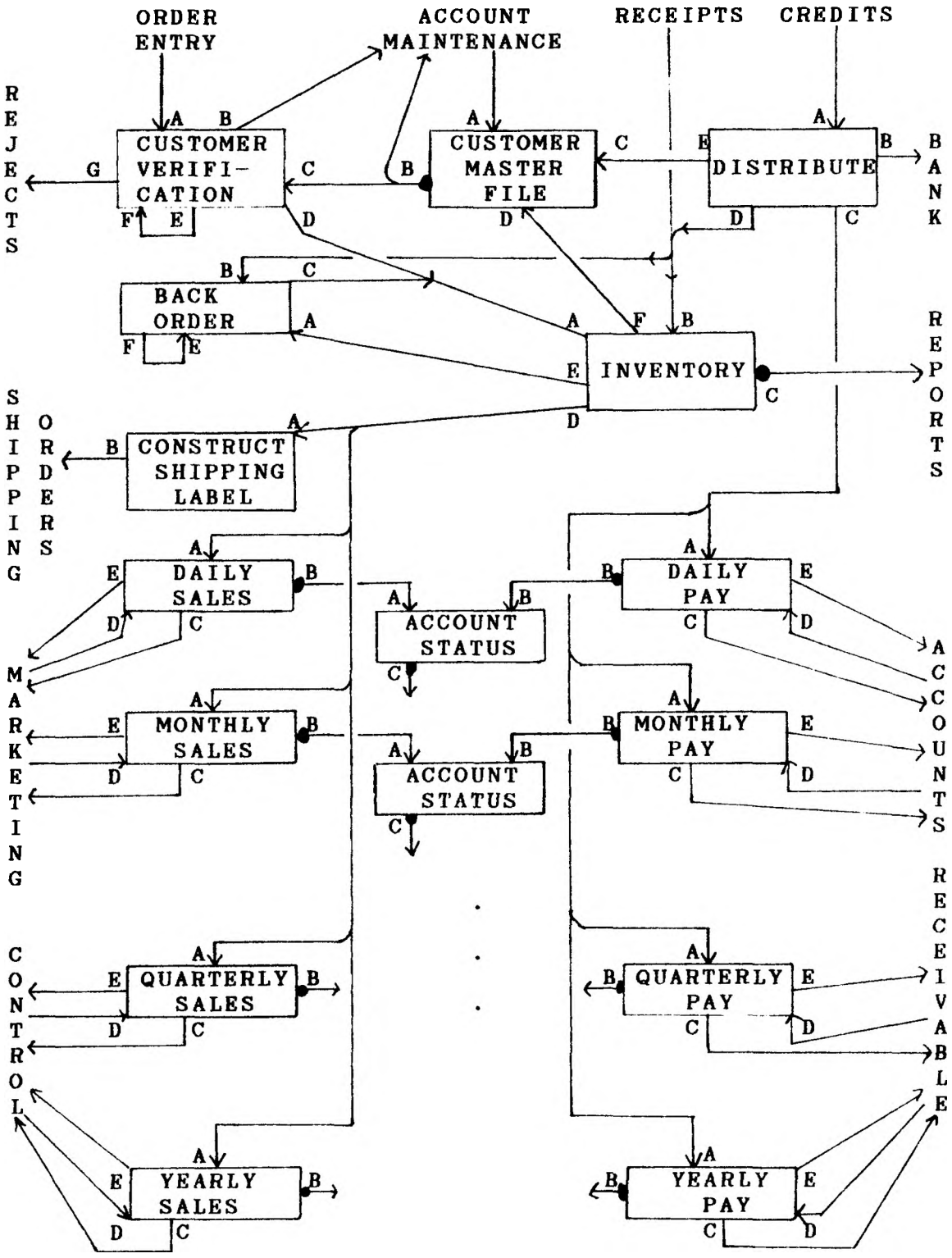

FIGURE 42
ORDER PROCESSING PROGRAM

```
1     MODULE CUSTOMER VERIFICATION:

2     MODULE TRIGGER:  TRIG ONE:   A,   ENTRY_ONE;
                            TWO:   C,   ENTRY_TWO;

3     MODULE PSEUDO-CODE:

4           ENTRY_ONE:
5                 READ FROM PORT A TO VARIABLE NEW_ORDER.
6                 WRITE CUSTOMER REQUEST TO PORT C.
7                 WRITE NEW_ORDER TO PORT E.
8                 SLEEP.
9           END ENTRY_ONE.

10          ENTRY_TWO:  {match order status & chk invent.}
11                READ PORT C INTO ORDER_STATUS.
12                READ PORT F TO NEW_ORDER AND WRITE PORT E
13                      WITH NEW_ORDER UNTIL ORDER AGREES
14                      WITH ORDER_STATUS.
15                IF ORDER_STATUS IS OK THEN WRITE NEW_ORDER
16                      TO PORT D.
17                ELSE WRITE ORDER TO PORT G.
18                IF ORDER_STATUS IS "NOT PRESENT" WRITE
19                      NEW_ORDER TO PORT B.
20                SLEEP.
21          END ENTRY_TWO.


22    END MODULE CUSTOMER VERIFICATION.
```

FIGURE 43
CUSTOMER VERIFICATION MODULE

```
1     MODULE CUSTOMER MASTER FILE:

2     MODULE TRIGGER:   TRIG ONE: A, ENTRY_ONE;
                             TWO: C, ENTRY_TWO;
                             THREE: D, ENTRY_THREE;
3                            FOUR: B, ENTRY_FOUR;

4     MODULE PSEUDO-CODE:

5           ENTRY_ONE:
6                 READ PORT
7                 ADD OR DELETE A RECORD
9                 SLEEP
            END ENTRY_ONE

10          ENTRY_TWO:
11                READ PORT
12                CREDIT THE INDICATED ACCOUNT.
13                SLEEP
14          END ENTRY_TWO

15          ENTRY_THREE:
16                READ PORT
17                DEBIT THE INDICATED ACCOUNT.
18                SLEEP
19          END ENTRY_THREE

20          ENTRY_FOUR:
21                READ PORT B.
22                SUPPLY PORT B WITH THE REQUESTED RECORD.
23                SLEEP
24          EMD ENTRY_FOUR.

25    END CUSTOMER MASTER FILE
```

FIGURE 44
CUSTOMER MASTER FILE MODULE

4. Back Order Module. GENERAL DESCRIPTION: The module, called Back Order, holds orders when insufficient inventory exists. Back orders are retained within the module to be filled when new inventory arrives.

DATA DESCRIPTION: Port A receives orders that cannot be filled. Ports F and E act as a sequential file to hold the back orders. The back orders are written to port C for further processing upon receipt of a message at port B. The module pseudo-code is in Figure 46.

5. Inventory. GENERAL DESCRIPTION: The module stores the quantity of merchandise on hand. Orders are received, inventory adjusted and requests satisfied by the module. Parameters to determine reorder point and other inventory details are provided by the reports environment.

DATA DESCRIPTION: Port A receives orders for merchandise. Port B receives data added to inventory when new orders arrive. When sufficient inventory exists, the order is filled by writing it to port D and adjusting the inventory level for that item. Port C responds to requests regarding inventory status. When insufficient inventory exists to fill an order, the order is written to port E. As orders are filled, the amount of the order is written to port F. The pseudo-code for the module is given in Figure 47.

```
1     MODULE DISTRIBUTE:

2     MODULE TRIGGER:  TRIG ONE: A,  ENTRY_ONE;
3                            TWO: EOF A, ENTRY_TWO;

4     MODULE PSEUDO-CODE:

5           ENTRY_ONE:
6                READ PORT A.
7                IF DATA IS CASH_CREDIT THEN DO.
8                     WRITE CASH AMOUNT TO PORT B.
9                     WRITE CUSTOMER NUMBER AND CASH TO
10                         PORT E.
11                    WRITE CASH AMOUNT TO PORT C.
12                    END
13               ELSE DO. { process returns }
14                    WRITE INVENTORY DATA TO PORT D.
15                    WRITE CREDIT AMOUNT AND CUSTOMER
16                         NUMBER TO PORT E.
17                    END
18               SLEEP.
19          END ENTRY_ONE.

20          ENTRY_TWO:
21               WRITE EOF TO PORT B
22               WRITE EOF TO PORT C
23               SLEEP
24          END ENTRY_TWO

25    END MODULE DISTRIBUTE
```

FIGURE 45
DISTRIBUTE MODULE

```
1     MODULE BACK ORDER:

2     MODULE TRIGGER:  TRIG ONE: A,  ENTRY_ONE;
3                           TWO: B,  ENTRY_TWO;

4     MODULE PSEUDO-CODE:

5           ENTRY_ONE:
6                READ PORT A INTO DATA.
7                WRITE DATA TO PORT F.
8                SLEEP
9           END ENTRY_ONE

10          ENTRY_TWO:
11               READ PORT B
12               WRITE AN EOF MARK TO PORT F
13               REPEAT UNTIL EOF OCCURS
14                    READ PORT E INTO DATA
15                    WRITE DATA TO PORT C
16               END REPEAT
17               SLEEP
18          END ENTRY_TWO

19    END MODULE BACK ORDER
```

FIGURE 46
BACK ORDER MODULE

6. Construct Shipping Label. GENERAL DESCRIPTION: The Construct Shipping Label module receives orders that can be filled. The information received at port A is formatted and passed to the division of shipping orders.

DATA DESCRIPTION: Port A receives orders as generated. Format specifications are built into the module. The order is written to port B. Pseudo-code for the module is in Figure 48.

7. Daily Sales And Daily Pay Modules. GENERAL DESCRIPTION: The modules are repeated and used to accumulate the sales or payments by a customer on a daily basis. The module receives orders from which the customer and amount is extracted. A total is retained for each customer. The item and quantity is used to maintain a total number sold for each item. The module interacts with marketing control to produce reports as desired. The generic Keep Sorted module and Keep Totaled modules make up the daily sales module as shown in Figure 50. The nesting modules feature allows modules to be comprised of modules, ports, and paths and to promote a heirarchy of complexity.

DATA DESCRIPTION: Port A receives the order data which is accumulated by customer name. Port B provides customer records on request. When the end of the period occurs, the records are written to port C. Ports D and E communicate with the marketing control division. Port D

```
1     MODULE INVENTORY:

2     MODULE TRIGGER:   TRIG ONE: A, ENTRY_ONE;
3                             TWO: B, ENTRY_TWO;
4                             THREE: C, ENTRY_THREE;

5     MODULE PSEUDO-CODE:

6           ENTRY_ONE:
7                 READ PORT A INTO DATA
8                 DETERMINE FROM THE ORDER THE REQUIRED
9                       INVENTORY
10                IF CURRENT INVENTORY LEVEL FOR THE ITEM >
11                      REQUIRED INVENTORY THEN DO:
12                      WRITE THE ORDER TO PORT D
13                      WRITE THE DOLLAR AMOUNT TO PORT F
14                      END
15                ELSE WRITE THE ORDER TO PORT E
16                SLEEP
17          END ENTRY_ONE

18          ENTRY_TWO:
19                READ PORT B INTO DATA
20                ADJUST ITEM INVENTORY LEVEL TO REFLECT NEW
21                      INVENTORY RECEIVED
22                SLEEP
23          END ENTRY_TWO

24          ENTRY_THREE:
25                READ PORT C INTO REQUEST
26                WRITE REQUESTED DATA TO PORT C
27                SLEEP
28          END ENTRY_THREE

29    END INVENTORY MODULE
```

FIGURE 47
INVENTORY MODULE

```
1     MODULE CONSTRUCT SHIPPING LABEL:

2     MODULE TRIGGER:  TRIG ONE: A,  ENTRY_ONE;

3     MODULE PSEUDO-CODE:

4          ENTRY_ONE:
5               READ PORT A INTO ORDER
6               USING SHIPPING DESIGN FORMS CONVERT ORDER
7                    TO APPROPRIATE SHIPPING LABEL.
8               WRITE SHIPPING LABEL TO PORT B.
9               SLEEP
10         END ENTRY_ONE.

11   END CONSTRUCT SHIPPING LABEL
```

FIGURE 48
CONSTRUCT SHIPPING LABEL MODULE

receives a prod causing records to be written to port C. Records are also written to port E as requested.

MODULE DAILY SALES: The pseudo-code and triggers for a module comprised of other modules are given by the lower level modules. Therefore, no pseudo-code is expressed at this point. (See Figures 50, 51, 52, 53, 54 and 55 for the modules and pseudo-code that comprise the pay and sales modules.)

8. Account Status. GENERAL DESCRIPTION: The module, Account Status, creates reports of a customer's status by merging data from ports A and B. Data supplied has been sorted so that data is merged directly and reports formatted within the module.

DATA DESCRIPTION: Port A receives data related to a customer's sales while Port B receives data related to a customer's payments. The two items are merged into one report and supplied upon request to port C. The module pseudo-code is shown in Figure 49.

9. Remaining Modules. The remaining modules in the program are replications of modules already described. Monthly sales, quarterly sales and yearly sales are identical to daily sales. Marketing control provides information to indicate when the module will emit its information. Upon providing the information, the module zeroes all totals. Each module performs an identical task, passing its data to port C upon receipt of the prod

from marketing control. Marketing determines the time of year for the module distinguishing a monthly sales module from a yearly sales module, for example. A similar discussion exists for the "pay" modules.

There exists an account status module for each period. Upon request, data is extracted from the sales and pay modules corresponding to the account status requested. Current reports can be produced for each account since data is always "up-to-date" in each of the supplying modules. Individual reports by customer or comprehensive reports can be produced as desired.

10. Sales And Pay - Nested Modules. The program segment shown in Figure 50 exemplifies a number of the programming features previously described. Nesting is shown by modules which make up the Pay and Sales modules. New triggers are shown in the Keep Sorted On Key module and program execution considerations are shown by resetting sums and initialization considerations of Keep Sorted and Keep Totaled. Each of these features are demonstrated within the following examples. Data request queueing is exemplified by the Keep Sorted module. The operating system queues the request allowing the module to satisfy one request at a time. The system routes the request to the appropriate path. Requests are satisfied in order. The system queues requests and routes responses accordingly.

```
1     MODULE ACCOUNT STATUS:

2     MODULE TRIGGER:  TRIG  ONE:  A AND B,  ENTRY_ONE;
3                      TWO: C,  ENTRY_TWO;
4                      THREE:  A AND EOF B,  ENTRY_THREE;
5                      FOUR:  EOF A AND B,   ENTRY_FOUR;
6                      FIVE:  EOF A AND EOF B,  ENTRY_FIVE;

7     MODULE PSEUDO-CODE:

8           ENTRY_ONE:
9                READ PORT A TO SALE
10               READ PORT B TO PAY
11               MERGE SALE AND PAY TO CREATE A REPORT
12               WRITE REPORT TO PORT C
13               SLEEP
14          END ENTRY_ONE

15          ENTRY_TWO:
16               READ PORT C INTO REQUEST
17               WRITE REQUEST TO PORT B AND TO PORT A
18               SLEEP
19          END ENTRY_TWO

20          ENTRY_THREE:
21               READ PORT A TO SALE
22               READ PORT B
23               WRITE SALE TO PORT C WITH MESSAGE "NO
24                    CUSTOMER IN PAYMENT FILE FOR THIS
25                    SALE".
26               SLEEP
27          END ENTRY_THREE

28          ENTRY_FOUR:
29               READ PORT A
30               READ PORT B TO PAY
31               WRITE PAY TO PORT C WITH MESSAGE "NO
32                    CUSTOMER IN SALE FILE FOR THIS
33                    PAYMENT".
34               SLEEP
35          END ENTRY_FOUR:

36          ENTRY_FIVE:
37               READ PORT A
38               READ PORT B
39               WRITE MESSAGE "NO CUSTOMER IN EITHER FILE
40                    TO SATISFY REQUEST"
41               SLEEP
42          END ENTRY_FIVE

43    END MODULE ACCOUNT STATUS
```

FIGURE 49
ACCOUNT STATUS MODULE

a. Keep Totaled By Key. GENERAL DESCRIPTION: Two Keep Totaled by key modules are shown in Figure 50. They are called Keep Totaled By Item and Keep Totaled By Customer. Each performs a similar function on the data it receives and represents the duplicate module concept. One module totals the number of each item sold while the other module totals the sales from each customer. Each total is accumulated as data is received, allowing immediate satisfaction of requests.

DATA DESCRIPTION: One data description is required for both identical modules. Port A receives the data to be totaled: Item and Quantity or Name and Dollar Amount. Upon receipt of a prod at port C, all totals are written to port B and the module total reset to zero. Port D receives requests for individual sums by name or item as required by Marketing Control.

b. Keep Sorted. GENERAL DESCRIPTION: The module retains sorted records by name. Data received by the module is Name, Item, Quantity and Amount. The module inserts each record received in its correct sorted order by name. All sorting is done immediately upon receipt of the data, keeping the file current. Requests for records are satisfied without delay since the file remains sorted.

DATA DESCRIPTION: Port A receives new records as described. Requests are queued by the system at port B. Each request is satisfied and routed to the appropriate requestor by the system. The requests are for a named

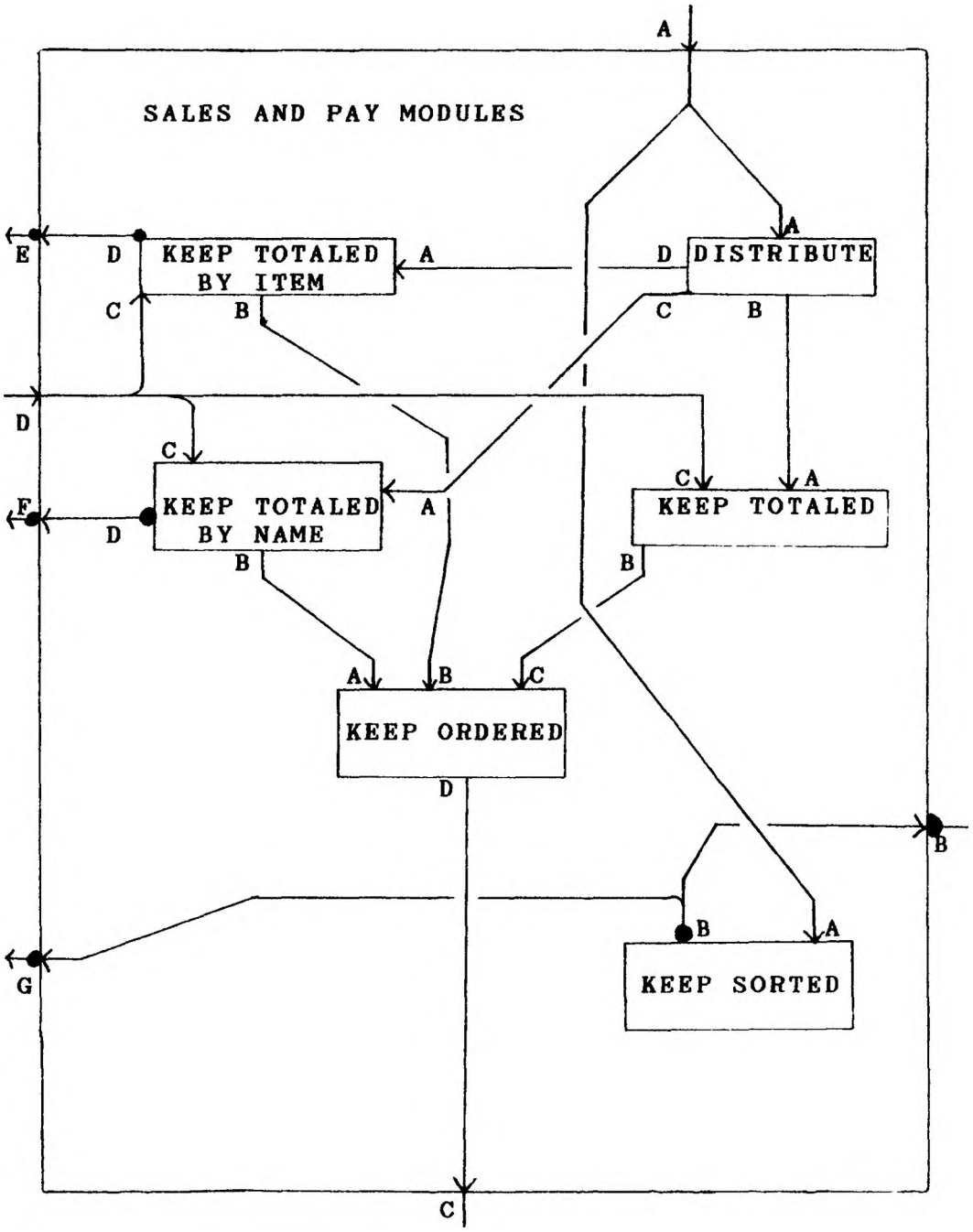

FIGURE 50
SALES AND PAY MODULE

record or a complete file. The module will satisfy either request and when all records are written, the file is reset. The pseudo-code for Keep Sorted is in Figure 55.

c. Keep Totaled. GENERAL DESCRIPTION: The module receives data representing an amount. Each amount is added to produce an overall sum for the period. A running total of sales or payments is maintained by the module.

DATA DESCRIPTION: Port B receives data in the form of an amount. The amount is added to the total. Upon receipt of a prod from port A, the total is written to port C and the total reset to zero.

d. Keep Ordered. GENERAL DESCRIPTION: The module receives records through three ports. Data received cannot be intermixed which requires all data to be processed from one port before processing any data from the another port. Data received at ports A, B or C is passed to port D in order.

DATA DESCRIPTION: The port receiving the first data message causes module activation. Variables and triggers ensure all processing at a port is complete before processing the next port and all three ports have been processed before starting over. The variables DONE_A, DONE_B and DONE_C determine which ports have already been processed. Triggers determined by the variables ensure correct processing of the ports.

e. Distribute. The distribute module nested in the Sales Or Pay module is identical to the Distribute module defined in Figure 42. The data received is different and the distribution modified, but the function is conceptually identical. A data record is received and specific fields written to ports as required.

DATA DESCRIPTION: Port A receives records of the form: Name, Item, Quantity and Amount. The Name and Amount are written to port C, Item and Quantity are written to port D and the Amount is written to port B. Port E as previously described in Figure 42 is not used in this application and will not be included in the trigger.

f. Keep Totaled. GENERAL DESCRIPTION: The module keeps a current total of all sales or payments as determined by the module's position within the program. That is, if the module is in the sales section, it keeps a total of all sales. If it is in the payments section, it keeps a total of all payments. The total is provided upon receipt of a prod at port C which also allows the module to reset the total to zero.

DATA DESCRIPTION: Port A receives an Amount. The Amount is entered into the sum. Upon receipt of a prod at port C, the total is written to port B and is reset to zero.

```
1     MODULE KEEP TOTALED ON KEY

2     MODULE TRIGGER:  TRIG ONE:  A,  ENTRY_ONE;

3     MODULE PSEUDO-CODE:

4     ENTRY_ONE:
5          READ PORT A
6               FIND ITEM SUM
7               ADD AMOUNT TO ITEM SUM
8          SLEEP: NEW TRIGGER:
9               TRIG ONE:  A,  ENTRY_ONE;
10                    TWO:  C,  ENTRY_TWO;
11                    THREE: D, ENTRY_THREE;
      {this trigger organization allows data receipt before
       purge or request }
12    END ENTRY_ONE

13    ENTRY_TWO:
14         READ PORT C
15         WRITE ALL RECORDS TO PORT B
16         RESET MODULE TO INDICATE ABSENCE OF RECORDS
17         SLEEP:  NEW TRIGGER:
18              TRIG.  ONE:  A,  ENTRY_ONE;
19    END ENTRY_TWO

20    ENTRY_THREE:
21         READ PORT D INTO REQUEST
22         WRITE REQUESTED RECORD TO PORT D.
23         SLEEP
24    END ENTRY_THREE

25    END KEEP TOTALED ON KEY
```

FIGURE 51
KEEP TOTALED ON KEY

```
1     MODULE DISTRIBUTE (NESTED)

2     MODULE TRIGGER:  TRIG ONE:   A, ENTRY_ONE;
3                            TWO: EOF A, ENTRY_TWO;

4     MODULE PSEUDO-CODE:

5     ENTRY_ONE:
6          READ PORT A INTO DATA
7          EXTRACT NAME AND AMOUNT AND WRITE TO PORT C
8          EXTRACT ITEM AND QUANTITY AND WRITE TO PORT D
9          EXTRACT AMOUNT AND WRITE TO PORT B
10         SLEEP
11    END ENTRY_ONE

12    ENTRY_TWO:
13         WRITE EOF TO PORT C
14         WRITE EOF TO PORT D
15         WRITE EOF TO PORT B
16         SLEEP
17    END ENTRY_TWO

18    END DISTRIBUTE MODULE (NESTED)
```

FIGURE 52
MODULE DISTRIBUTE (NESTED)

```
1     MODULE KEEP ORDERED

2     MODULE TRIGGER:   TRIG  ONE: A, ENTRY_ONE;
3                             TWO: B, ENTRY_TWO;
4                             THREE: C, ENTRY_THREE;

5     MODULE PSEUDO-CODE:

6           ENTRY_ONE:
7                 DONE_A = TRUE
8                 READ PORT A AND WRITE PORT D
9                 SLEEP: NEW TRIGGER:
10                      TRIG ONE: A, ENTRY_ONE;
11                           TWO: EOF A, ENTRY_FOUR;
12          END ENTRY_ONE

13          ENTRY_TWO:
14                DONE_B = TRUE
15                READ PORT B AND WRITE PORT D
16                SLEEP: NEW TRIGGER:
17                      TRIG ONE: B, ENTRY_TWO;
18                           TWO: EOF B, ENTRY_FOUR;
19          END ENTRY_TWO

20          ENTRY_THREE:
21                DONE_C = TRUE
22                READ PORT C AND WRITE PORT D
23                SLEEP: NEW TRIGGER:
24                      TRIG ONE: C, ENTRY_THREE;
25                           TWO: EOF C, ENTRY_FOUR;
26          END ENTRY_THREE;

27          ENTRY_FOUR:  {one or more ports finished}
28                IF NOT DONE_A THEN
29                      IF NOT DONE_B THEN {c is done}
30                            SLEEP: NEW TRIGGER:
31                                  TRIG ONE: A, ENTRY_ONE;
32                                       TWO: B, ENTRY_TWO;

33                      ELSE IF NOT DONE_C THEN {b is done}
34                            SLEEP: NEW TRIGGER:
35                                  TRIG ONE: A, ENTRY_ONE;
36                                       TWO: C, ENTRY_THREE;
37                      ELSE SLEEP: NEW TRIGGER: {b, c done}
38                                  TRIG ONE: A, ENTRY_ONE;
```

FIGURE 53
KEEP ORDERED MODULE

```
39              ELSE IF NOT DONE_B THEN {a is done}
40                   IF NOT DONE_C THEN
41                        SLEEP: NEW TRIGGER:
42                             TRIG ONE: B, ENTRY_TWO;
43                                  TWO: C, ETNRY_THREE;
44                   ELSE SLEEP: NEW TRIGGER: {a & c done}
45                             TRIG ONE: B, ENTRY_TWO;

46              ELSE IF NOT DONE_C THEN {a & b done}
47                   SLEEP: NEW TRIGGER:
48                        TRIG ONE: C, ENTRY_THREE;

49              ELSE  {a, b & c done}
50                   DONE_A = FALSE
51                   DONE_B = FALSE
52                   DONE_C = FALSE
53                   SLEEP: NEW TRIGGER:
54                        TRIG ONE: A, ENTRY_ONE;
55                             TWO: B, ENTRY_TWO;
56                             THREE: C, ENTRY_THREE;
57                   WRITE EOF TO PORT D
58         END ENTRY_FOUR

59    END KEEP ORDERED MODULE
```

FIGURE 53 (continued)
KEEP ORDERED MODULE

```
1     MODULE KEEP TOTALED:

2     MODULE TRIGGER:  TRIG ONE: A, ENTRY_ONE;
3                           TWO: C, ENTRY_TWO;

4     MODULE PSEUDO-CODE:

5           ENTRY_ONE:
6                 READ PORT A INTO DATA
7                 ADD AMOUNT FROM DATA RECORD TO TOTAL
8                 SLEEP
9           END ENTRY_ONE

10          ENTRY_TWO:
11                READ PORT C
12                WRITE ALL RECORDS TO PORT B PURGING EACH
13                      RECORD AFTER WRITING
14                WHEN FINISHED WRITE EOF TO PORT B
15                SLEEP
16          END ENTRY_TWO

17    END KEEP TOTALED
```

FIGURE 54
KEEP TOTALED

```
1     MODULE KEEP SORTED:

2     MODULE TRIGGER:   TRIG ONE: A, ENTRY_ONE;
3                             TWO: B, ENTRY_TWO;

4     MODULE PSEUDO-CODE:

5           ENTRY_ONE:
6                 READ PORT A INTO NEW_DATA.
7                 INSERT NEW_DATA INTO FILE IN SORTED ORDER
8                 SLEEP
9           END ENTRY_ONE

10          ENTRY_TWO:
11                READ PORT B INTO REQUEST.
12                WRITE SATISFIED REQUEST TO PORT B.
13                RESET THE MODULE IF ALL RECORDS ARE
14                      REQUESTED AND WRITE EOF.
15                SLEEP
16          END ENTRY_TWO
```

FIGURE 55
KEEP SORTED

G. PROGRAM EXAMPLE - SPELLING CHECKING PROGRAM.

Many vehicle features are shown by the following example. The program demonstrates how a simple, primarily sequential, algorithm can be written within the model and yield a parallel program. Parallelism is exploited at the module level by duplicating common modules and distributing tasks as shown in Figure 56.

GENERAL PROGRAM DESCRIPTION: The program receives a data file consisting of the document to be checked for spelling errors. The results of the program are a list of misspelled words and the original document with each misspelling marked. The data dictionary modules are capable of receiving new words as required. These modules perform a search to match the words in the document with the words in the dictionary. If a match is found, the word is assumed correctly spelled. The program distributes the document among many search modules. To reconstruct the document, the program associates a successive integer with each word. The document is reconstructed by the merge modules. The strip module removes the integers from the words of the document.

1. Count. GENERAL DESCRIPTION: The module creates data records by associating successive integers with each new word. The order of the original document is

reconstructed by using the number associated with each word to determine correct ordering.

DATA DESCRIPTION: Port A receives words. An internal counter assigns successive integers to each new word. The word and its number are written to port B.

2. Distribute. GENERAL DESCRIPTION: The module is identical to the previous distribute modules defined in Section F. The purpose of the distribute module is to improve computing efficiency by distributing the processing to a number of search modules capable of parallel execution. The exact distribution scheme is delayed until module implementation time, but could use an alphabetic distribution of words.

DATA DESCRIPTION: The module receives data records through port A. As determined by the distribution procedure, some of the records are written to port B and some of the records to port C. A common distribution procedure may write words beginning with A through M to port B and N through Z to port C. The pseudo-code is in Figure 57.

3. Search And Tally. GENERAL DESCRIPTION: The module receives words and their associated value. These words are compared with dictionary entries to determine correct spellings. If a match is found, the word is assumed correctly spelled. Additions to the dictionary are possible. The number of references to each word in

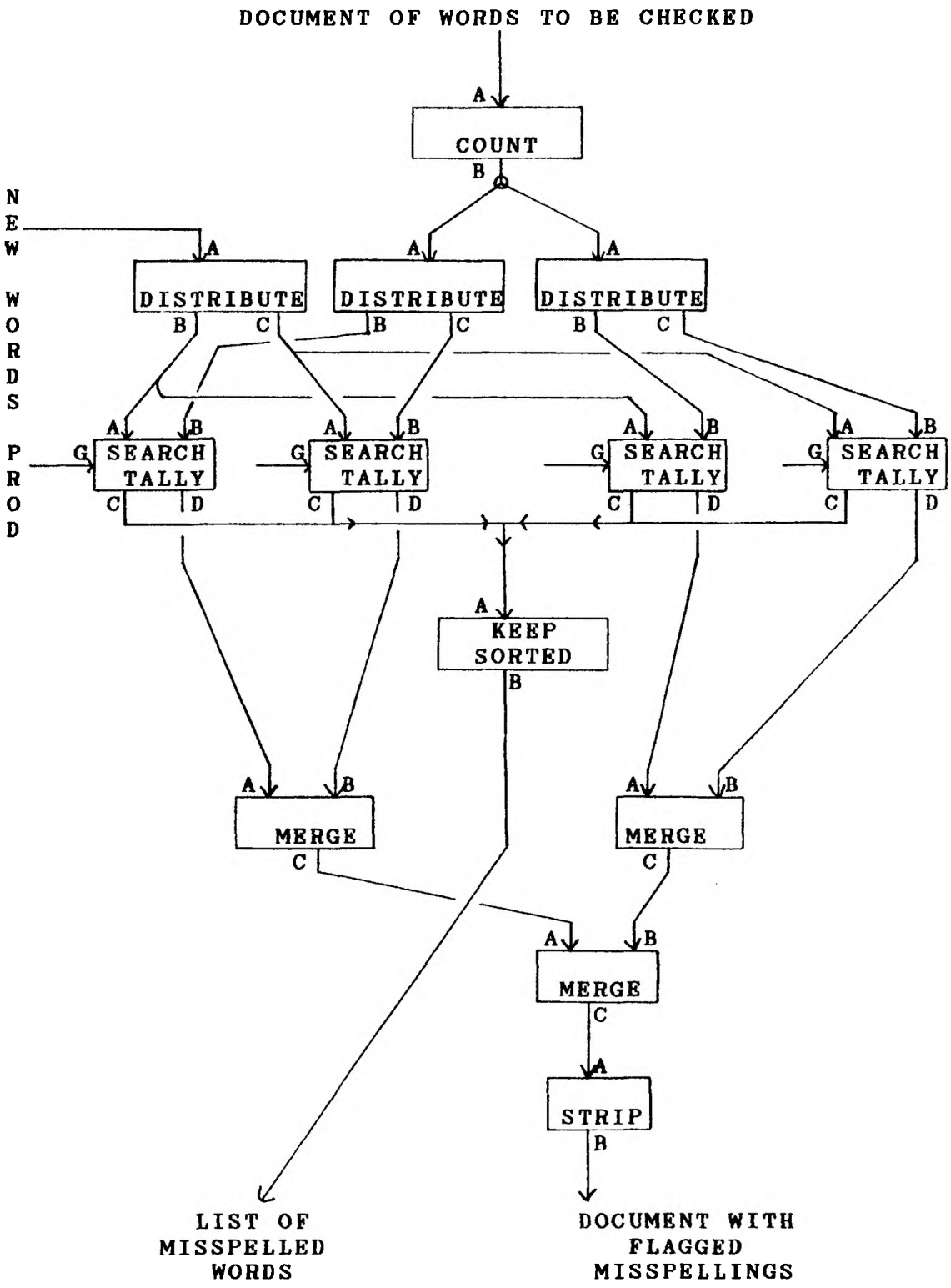

FIGURE 56
SPELLING CHECKER

```
MODULE COUNT:

MODULE TRIGGER:  TRIG  ONE: TRUE,  ENTRY_ONE;

MODULE PSEUDO-CODE:

     ENTRY_ONE:
          COUNTER = 0.
          SLEEP: NEW TRIGGER:
               TRIG ONE: A, ENTRY_TWO;
                    TWO: EOF A, ENTRY_THREE;
     END ENTRY_ONE.

     ENTRY_TWO:
          COUNT = COUNT + 1
          READ PORT A TO WORD
          WRITE PORT B, WORD AND COUNT
          SLEEP  {trigger remains}
     END ENTRY_TWO

     ENTRY_THREE
          WRITE EOF TO PORT B
          SLEEP: NEW TRIGGER:
               TRIG ONE: TRUE, ENTRY_ONE;
     END ENTRY_THREE

END MODULE COUNT.
```

FIGURE 57
MODULE COUNT

the dictionary is recorded for reorganization of the search procedure, thereby enhancing efficiency. The stimulus causing reorganization is generated external to the module.

DATA DESCRIPTION: Port B receives data to be checked. New words are received through port A. Each misspelled word is written to port C and the document, with marked misspelled words, is written to port D. Port G receives the stimulus to reorganize the dictionary. The Search and Tally pseudo-code is shown in Figure 58.

4. Merge. GENERAL DESCRIPTION: Module Merge is similar to the merge module already described in Section E of Chapter 4. The module receives two ordered data streams, combines them and provides an ordered stream.

Keep Sorted, Distribute and Merge modules have been demonstrated in Section F of Chapter 4. The function of each module is conceptually the same as previously described, hence the pseudo-code is not presented in this section.

The program demonstrates the vehicle's ability to represent an algorithm without confusing the overall design with details. Conceptual program verbs such as merge and distribute are emphasized to promote this type of algorithm development. By observing the program graph, the reader is able to see the function of the program. The details of each module are shown in the figures

```
MODULE SEARCH AND TALLY:

MODULE TRIGGER:   TRIG ONE:   A,   ENTRY_ONE;
                       TWO:  B AND NOT A,   ENTRY_TWO;
                       THREE:   G,   ENTRY_THREE;
                       FOUR:  EOF B, ENTRY_FOUR;

(NOTE: the trigger organization gives priority to port a)

MODULE PSEUDO-CODE:

ENTRY_ONE:
      READ PORT A.
      ADD WORD TO DICTIONARY.
      SLEEP.
END ENTRY_ONE.

ENTRY_TWO:
      READ PORT B.
      CHECK WORD WITH DICTIONARY.
      WRITE WORD TO PORT E.
      IF WORD DOES NOT CHECK THEN WRITE WORD TO PORT F.
      SLEEP.
END ENTRY_TWO.

ENTRY_THREE:
      READ PORT G.
      REORGANIZE DICTIONARY
      SLEEP.
END ENTRY_THREE.

ENTRY_FOUR:
      WRITE EOF TO PORT C
      WRITE EOF TO PORT D
END ENTRY_FOUR

END MODULE SEARCH AND TALLY.
```

FIGURE 58
MODULE SEARCH AND TALLY

containing pseudo-code. Enhanced understanding of the program design is accomplished by removing the details to the separate section. The program is constructed with conceptual modules rather than explicit instructions of the implemented procedures.

5. Strip. GENERAL DESCRIPTION: The module removes the integers associated with each word to produce the final checked document with marked misspellings.

DATA DESCRIPTION: Data is received at port A containing words and integers. Each word may have a mark associated with it. The module passes the words along to port B producing the original document. Pseudo-code is given in Figure 45.

```
MODULE STRIP:

MODULE TRIGGER:   TRIG ONE:   A,   ENTRY_ONE;
                       TWO:  EOF A,   ENTRY_TWO;

MODULE PSEUDO-CODE:

ENTRY_ONE:
      READ PORT A WORD AND INTEGER
      WRITE WORD TO PORT B
      SLEEP
END ENTRY_ONE

ENTRY_TWO:
      WRITE EOF TO PORT B
      SLEEP
END ENTRY_TWO

END MODULE STRIP
```

FIGURE 59
MODULE STRIP

H. SUMMARY.

This chapter has shown examples of program construction within the vehicle used to describe algorithms. Features discussed in the previous chapters are demonstrated in this chapter. Common modules show how programming at a higher level can be used to demonstrate solutions to problems. Paths that fan-out are shown in instances where data is copied into both paths. Paths that branch out are shown where data is placed in the path with the least amount of data waiting to be processed. Module duplication by the program designer is exemplified in Figure 4.27. The examples in the chapter show how programs are constructed and how the vehicle is used to construct algorithms. The pictorial representation and the pseudo-coding of module details allow the design of an algorithm to be clearly shown. The goals, to provide a vehicle for the expression of algorithms and to present a new computing environment fostering medium-grained parallelism and awareness of concurrency during algorithm design, are demonstrated by these examples.

V. CONCLUSIONS AND FURTHER RESEARCH

A. CONCLUSION.

A new way to think about algorithm design and computing systems has been described. The concept of programming modules being always present is an extension of current computing systems. Compilers and interpreters now exist in this way. They are system-resident and become active when a program is present. The program that is compiled is data allowing activation of the compiler. The proposed vehicle extends this concept to allow all program modules to be present in the system and become active when data is present. The artificial control (traditional job control languages) used in activating compilers is eliminated by allowing data presence to activate a module.

Generality is achieved by allowing a variety of data presence conditions to activate the module. The envisioned vehicle allows modules to become active when any data or a combination of data presence exists at ports of a module. Complete files or commands from the operator are no longer required to determine activation. Modules become active when any data exists rather than waiting for a complete file. The merge module in Chapter 4 is an excellent example of this concept. Traditional merge facilities require complete files to be present which

delays results. The modules in the envisioned vehicle become active when any data is present and produce results immediately.

Algorithm design is possible without a great deal of complexity involved. Features provided by the operating system are those required to facilitate algorithm design without forcing concern for details. The operating sytem supports module communication by allowing direct construction of paths between ports of modules. The module trigger constructed by the programmer allows module activation flexibility. Modules react to varying data presence conditions tested by the system. These facilities of the vehicle are designed to assist the programmer in developing algorithms.

Increased computer performance relies on the development of parallel systems. Much effort is taking place toward the development of parallel architectures. To receive the greatest benefit a comparable effort is required to develop parallel software. The envisioned vehicle allows design of parallel algorithms. Current parallel systems seek to exploit parallelism from sequentially designed algorithms. Potential parallelism is lost within a sequential solution. By designing an algorithm within an inherently parallel system, potential parallelism need not be lost.

Algorithms designed within the envisioned vehicle are presented in a graphical manner. Using graphs allows the

design to be understood and evaluated. Program designs within this vehicle can be discussed and improved due to the representation of the algorithm. After an accepted design has been developed, the algorithm can be adapted to one of the existing parallel architectures or the described vehicle can be used as a guide for developing an architecture capable of directly supporting the envisioned system.

Chapter 4 shows examples of two applications of the vehicle designed. Figure 42 shows a complete computing system where modules are always resident to satisfy a particular need. The example shows a dedicated computing system designed for a specific application. Parallelism is exploited within the general solution. Figure 56 shows an application of the vehicle to a specific problem. Spelling checking or other similar functions are part of a larger program. These examples show how the vehicle meets the initial goals of the design. The vehicle promotes the specification and parallel solution of problems. It also provides a framework for envisioning a different computing system.

Programming development is enhanced within the vehicle by constructing solutions to problems through the use of high-level verbs. The development of an algorithm is no longer confused by details of the implementation and yet the algorithm is outlined sufficiently for understanding.

The support external to the module is designed to assist the construction of all parallel algorithms. Specific problem details are solved inside a module. For example, the branch-out path is required to enhance processing efficiency. It is a form of data distribution required of all programs. The distribute module, in contrast, is dependent on the problem. Since different problems require different conditions for data distribution, module development is required to solve the specific problem.

An additional advantage to programs designed with this vehicle is a built-in documentation facility. To describe the function of existing modules, the module designer must include a description of what the module does. The description includes data received at each port and data supplied by each port. By investigating the data received and supplied and the function of the module, a reader is able to determine the overall function of the program. Module descriptions are used if the general program function is not obvious from the program network supplied.

The vehicle designed provides insight to a different type of programming environment. The difference is not a large step from conventional programming systems, but is a significant step in organization. The organization allows the development and analysis of parallel algorithms. The goal of this research has been achieved; to promote

thinking about a different type of computing system, and to provide a medium for the development and discussion of parallel algorithms.

B. FURTHER RESEARCH.

In any data flow model, problems arise when errors enter the system. The vehicle described requires continued research to determine effective ways to deal with errors. A module's processor that creates incorrect data requires recognition and notification of other modules supplied by the incorrect module. Errors are also entered into the system from modules incorrectly designed. Not writing the EOF marker into a path is a common design error that causes receiving modules to operate in error. Run-away modules, those that never execute the SLEEP statement, need to be recognized. Further development is required to recognize both processor failures and programmer errors.

This research concentrates on design aspects of the vehicle external to the module. Internal considerations are developed only to the extent required for understanding. Further research is needed to determine appropriate ways of constructing modules. Some considerations for the internal module development are how module design can use operating system facilities most efficiently. Internal module considerations also include required extensions to current languages to allow communication with ports. Some consideration should be

given to determining appropriate languages for use in the module.

Further research is required to consider the implementation details of how modules and complete program networks are entered into a computing system. This research includes the process of program modification. Such questions as determining when a module can be added, deleted or modified in a system which is potentially active at all times needs consideration. Adding revised modules to a system has been considered, but the manner in which modifying or revising a complete network of modules should be determined.

The vehicle can be extended to allow requests for data to be satisfied in any order. As described, the requests are satisfied in the order received. This feature was included to simplify the operating system. As programmers become familiar with the vehicle, additional operating system features can be included.

Research related to extending the model to allow dynamic specification of ports and paths, other than operating system replication of modules already in place, could be considered. Dynamically adding ports and paths gives a program network considerable flexibility for adapting to a variety of problem instances. Much consideration would be necessary to determine when and how to modify intermodular communication and the operating system features required to allow this facility.

If the operating system is implemented to support the vehicle, research is required to determine appropriate hardware configurations. Traditional computing problems such as deadlock, race conditions, and optimal processor assignment algorithms need to be considered. The functions performed by each processor need to be considered. A processor assigned to a module may be allowed to evaluate the trigger. When the trigger is satisfied, the processor can be reassigned to the module. No processor reassignment exists within this description. Efficient processor assignment can be attained through this concept.

More work can be done, but a vehicle for the development of parallel algorithms is in place. The goal of describing a new programming environment has been attained. The operating system has been developed to the extent that one can envision it being in place. Programming examples show how the vehicle supports the development of parallel algorithms. Through the use of this vehicle software engineers can begin to leave traditional sequential thinking about algorithm design and begin to develop fundamentally parallel algorithms.

REFERENCES

1. Ackerman, William B., "Data Flow Languages," Proc. 1979 NCC, pp. 1087-1095, AFIPS Press, 1979.
2. Agerwala, T. and Arvind, "Data Flow Systems," Computer, vol. 15, pp. 10-13, (Feb., 1982).
3. Arvind, and J.D. Brock, "Resource Managers in Functional Programming," Journ. of Parallel and Distributed Computing, vol. 1, pp. 5-21, (1984).
4. Arvind and J.D. Brock, "Streams and Managers," Computation Structures Group Memo-217, MIT, 1982.
5. Arvind, D.E. Culler, R.A. Ianucci, V. Kathail, K. Pingali and R.E. Thomas, "The Tagged Token Dataflow Architecture," Laboratory for Computer Science, MIT, Aug., 1983.
6. Arvind, M.L. Dertouzos and R.A. Iannucci, "A Multiprocessor Emulation Facility," TR/302, Laboratory for Computer Science, MIT, Oct., 1983.
7. Arvind and K.P. Gostelow, "The U-Interpreter," Computer, vol. 15, pp. 42-49, (Feb., 1982).
8. Arvind and R.A. Iannucci, "Two Fundamental Issues in Multiprocessing: The Dataflow Solution," TM/241, Laboratory for Computer Science, MIT, Sept., 1983.
9. Arvind, V. Kathail and K. Pingali, "Sharing of Computation in Functional Language Implementations," Laboratory for Computer Science, MIT, July, 1984.

10. Avrunin, George S. and Jack C. Wileden, "Describing and Analyzing Distributed Software System Designs," ACM Transactions on Programming Languages and Systems, vol. 7, pp. 380-403, (July, 1985).

11. Balzer, R.M., "Ports - A Method for Dynamic Interprogram Communication and Job Control," Proc. 1971 SJCC, pp. 485-489, AFIPS Press, 1971.

12. Boarder, J.C., "Graphical Programming for Parallel Processing Systems," Proc. 2nd International Conf. on Distributed Computing Systems, pp. 467-475, IEEE Computer Society Press, 1981.

13. Davis, A.L. and R.M. Keller, "Data Flow Program Graphs," Computer, vol. 13, pp. 26-41, (Feb., 1982).

14. DeRemer, Frank and Hans H. Kron, "Programming-in-the Large Versus Programming-in-the-Small," IEEE Transactions On Software Engineering, vol. SE-2, pp. 80-86, (June, 1976).

15. Gajski, D. D., D.A. Padna, D.J. Kuck and R. H. Kuhn, "A Second Opinion on Data Flow Machines and Languages," Computer, vol. 15, pp. 58-69, (Feb., 1982).

16. Gaudiot, J. L. and M. D. Ercegovac, "Performance Analysis of a Data-Flow Computer with Variable Resolution Actors," Proc. 4th International Conf. on Distributed Computing Systems, pp. 2-9, IEEE Computer Society Press, 1984.

17. Gomaa, H., "A Software Design Method for Real-Time Systems," Comm. ACM, vol. 27, pp. 938-949, (September, 1984).

18. Gostelow, K. P. and R. E. Thomas, "A View of Dataflow," National Computer Conference Proc., pp. 629-636, AFIPS Press, 1979.

19. Han, S. Y., "A Language for the Specification and Representation of Programs in a Data Flow Model of Computation," Ph.D. Diss., UT/8319598, University of Texas, Austin, Texas, 1983.

20. Jennings, S. F. and A. E. Oldehoeft, "An Analysis of Program Execution on Recursive Stream-Oriented Data Flow Architecture," The Journal of Systems and Software, pp. 147-154, (March, 1983).

21. Kernighan, B. W. and R. Pike, The Unix Programming Environment, Prentice-Hall, Inc., New Jersey, 1984.

22. Kramer, Jeff and Jeff Magee, "Dynamic Configuration for Distributed Systems," IEEE Trans. Trans. Software Eng., vol. se-11, pp. 424-435, (April, 1985).

23. Kunii, H., "Graph Data Language: A High Level Access-Path Oriented Language," Ph.D. Diss., UT/8319622, University of Texas, Austin, Texas, 1983.

24. Lan, M. T., "Characterization of Intermodule Communication and Heuristic Task Allocation for Distributed Real-Time Systems," Ph.D. Diss., CSD/850012, University of California, Los Angeles, California, 1985.

25. Leavenworth, B., "A Data Flow Pseudocode," Report RC 7772, IBM Corp., Yorktown Heights, N.Y., July, 1979.

26. Lesser, Victor, Daniel Serrain, and Jeff Bonar, "PCL: A Process-Oriented Job Control Language" Proc. First International Conf. on Distributed Computing Systems, pp. 315-329, IEEE Computer Society Press, 1979.

27. Matwin, S. and T. Pietrzykowski, "Prograph: A Preliminary Report," Computer Lang., vol. 10, pp. 91-126, (Feb., 1985).

28. McGraw, James R., "The VAL Language: Description and Analysis," Trans. on Prog. Languages and Systems, vol. 4, pp. 44-82, (January, 1982).

29. Mekly, L.J. and S.S. Yau, "Software Design Representation Using Abstract Process Networks," IEEE Trans. Software Eng., vol. se-6, pp. 420-434, (September, 1980).

30. Moriconi, M. and D. F. Hare, "Visualizing Program Designs Through Pegasys," Computer, vol. 18, pp. 72-85, (Aug. 1985).

31. Morrison, J.P., "Data Stream Linkage Mechanism," IBM Systems J., vol. 17, pp. 383-408, (1978).

32. Organick, E. I., "Algorithms, Concurrent Processors, and Computer Science Education," 16th Technical Symposium on Computer Science Education, pp. 1-5, Association for Computing Machinery, 1985.

33. Patnaik, L.M., P. Bhattacharya and R. Ganesh, "DFL: A Data Flow Language," Computer Languages, vol. 9, pp. 97-106, (1984).

34. Pingali, K. and Arvind, "Efficient Demand-Driven Evaluation (I)," TM/242, Laboratory for Computer Science, MIT, Sept., 1983.

35. Price, W. T., Computers and Application Software, An Introduction, Holt, Rinehart and Winston, New York, N.Y., 1985.

36. Quinn, M. J. and N. Deo, "Parallel Algorithms and Data Structures," CS/82-098, Computer Science Department, Washington State University, Pullman, Washington, Oct., 1982.

37. Richards, H., "An Overview of Arc Sasl," Burroughs Corporation, Austin Research Center, Austin, Texas, June, 1984.

38. Shulits, Jon, "A Functional Shell," CU-CS-245-83, Department of Computer Science, University of Colorado, Boulder, 1983.

39. Smith, J. O. Jr., "A Mechanism for Specifying Parallel Procedures," M.S. Thesis, Department of Computer Science, University of Missouri, Rolla, 1979.

40. Solomon, M.M. and R.A. Finkel, "The Roscoe Distributed Operating System," Proc. 7th Symp. on Operating Systems Principles, pp. 108-114, Association for Computing Machinery, 1979.

41. Srini, V. P., "A Fault Tolerant Dataflow System," Computer, vol. 18, pp. 54-68, (March, 1985).

42. Srini, V. P., "An Architectural Comparison of Data Flow Systems," Computer, vol. 19, pp. 68-88, (March, 1986).

43. Stevens, W.P., "How Data Flow Can Improve Application Development Productivity," IBM Systems J., vol. 21, pp. 162-178, (1982).

44. Stone, H. S., ed., Introduction to Computer Architecture, Science Research Associates, Inc., Chicago, 1980.

45. Syre, J.C., "The Data Flow Approach for MIMD Multiprocessor Systems," Parallel Processing Systems, ed. David J. Evans, pp. 239-273, Cambridge University Press, 1982.

46. Thoreson, S. A. and A. E. Oldenhoeft, "Instruction Reference Patterns in Data Flow Programs," Proc. of the Annual ACM Conference, pp. 211-217, Association for Computing Machinery, 1980.

47. van den Bos, Jan, Rinus Plasmeijer, and Jan Stroet, "Process Communication Based on Input Specifications," Trans. on Programming Languages and Systems, vol. 3, pp. 224-250, (July, 1981).

48. Ward, Stephen A. and Robert H. Halstead, Jr., "A Syntactic Theory of Message Passing," Journ. ACM, vol. 27, pp. 365-383, (April, 1980).

49. Watson, I. and J. Gurd, "A Practical Data Flow Computer," Computer, vol. 15, pp. 51-57, (Feb, 1982).

50. Weng, Kung-Song, "An Abstract Implementation for a Generalized Data Flow Language," Ph.D. Diss., MIT/LCS/TR-228, Laboratory for Computer Science, MIT, 1979.

VITA

Roger Edwin Eggen was born on May 3, 1947, in Havre, Montana. He received his primary and secondary education in Joplin, Montana. He received his Bachelor of Science degree from Northern Montana College in June, 1969. He taught for six years at Malta High School, Malta, Montana before beginning his graduate work.

He enrolled in The University of Nebraska - Lincoln where he received his Master of Science degree in Computer Science in 1977. He has been enrolled in the Graduate School of the University of Missouri - Rolla since August 1982. He held an AMOCO Research Fellowship from 1982 through 1985 and a Teaching Assistantship from 1985 through 1986. He is a student member of the Association for Computing Machinery and the Institute of Electrical and Electronics Engineers. He was initiated into the Upsilon Pi Epsilon honor society while at Nebraska.