AN ABSTRACT OF THE THESIS OF

PAUL A. SZULEWSKI (Name) for the MASTER OF SCIENCE (Degree)

in Electrical Engineering (Major) presented on 7 May 1976 (Date)

Title: A SCHEMATA MODEL OF REAL-TIME INFORMATION PROCESSING SYSTEMS

Redacted for Privacy

Abstract approved:

In an attempt to provide a practical means to measure the information processing capacity of a real-time digital processing system a theoretical framework, the Real-Time Schemata, whose basis is a schematic description and a measure of "useful information," is presented. The description applies to processes that would typically be implemented in an interrupt driven micro or mini processor system.

Real-time processes are subject to interrupts from external sources and time limitations to the completion of tasks which can be related to the quality of the information processed. The Real-Time Schemata yields a descriptive analytic model of this behavior. To analyze real-time processes a structured hierarchical form is most efficient. Fundamental structural elements are described and a procedure to generate them is given. The development of quality software within this framework is also discussed.

A Schemata Model of Real-Time Information Processing Systems

by

Paul A. Szulewski

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

June 1976

APPROVED:

Redacted for Privacy

Assistant Professor of Electrical and Computer Engineering
in charge of major

Redacted for Privacy

Head of Department of Electrical and Computer Engineering

Redacted for Privacy

Dean of Graduate School

Date thesis is presented 7 May 1976

Typed by Connie Chapman for Paul A. Szulewski

ACKNOWLEDGEMENT

I would like to express my gratitude to all those who have made this research an important part of my life.

In particular I would like to thank Dr. W. Richards Adrion, my major professor and friend, whose insight and encouragement directed this research effort and to Dr. Pieter A. Frick for his criticism and often different point of view. I thank the Department of Electrical and Computer Engineering and the National Science Foundation for the financial support they have given me and the faculty and staff of the Electrical and Computer Engineering Department, the Computer Science Department and the Engineering Experiment Station for their help and cooperation.

Finally to all my friends, my family and my wife, Karen, I extend my love and appreciation for the encouragement, help and understanding during the years of my graduate education.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

A SCHEMATA MODEL OF REAL-TIME INFORMATION PROCESSING SYSTEMS

INTRODUCTION

One can state, without loss of generality, that it is important to be able to evaluate the worth of any effort. Computer performance analysis is no exception and deserves consideration. This thesis is concerned with the development of a suitable model to evaluate and analyze real-time computer system performance.

The model, a Real-Time Schemata, is a structured hierarchical graph model designed to identify and analyze information production, flow and loss in the implementation of a real-time process in a digital computer. The motivation for the development of the model was provided by the inapplicability of traditional computer system models to real-time process behavior and the increasing use of limited capacity, low cost, dedicated micro and mini computers in real-time, on-line, process control or data acquisition applications. A real-time environment imposes a number of processing problems not normally encountered in a typical computer installation. These problems are related to the behavior of a sequential process subject to external interference in a time constrained domain.

Real-time data processing has many connotations. However for the purpose of this thesis real-time processing is concerned with the acquisition and sensing of externally generated not readily reproducible data within a time constaint and can exercise a time critical control action producing a physical result.

There are some keywords in the preceding definition that deserve some attention. In a real-time system data is acquired from an external source. This may be a transducer or some other device connected by an interface that converts real world data into a form acceptable to a digital machine. The data may not be reproducible, i.e. real-time data is not generally buffered, rather, it has to be utilized before a new input is generated, otherwise it will be lost. Calculations on that data may be time critical, i.e. in order to prevent loss of information, the calculation must occur in time to process the next input. The communication path between the real-time system and the external sources can be bi-directional. The system may send control or data information to a transducer that will cause a physical action, (e.g., positioning a saw blade in a mill) or the external source may send data to the processor.

The information with which a real-time process deals differs from the strictly quantitative definition of information an information theorist might use. Information in a real-time process is processed, not merely transmitted. It has, as a result, aspects of both quantity and quality where information quality is a measure proposed as a function of the ability of the process to attain its objective in the processing time available.

A digital computer system is an information processing device with finite memory and limited computational capability which is subject to error from its environment (noise), hardware and software failures (reliability) and it memory bandwidth (quantization). The development of larger, higher speed and more reliable computers has

generally preceded the application areas. A large scale computer in a total system has not usually been the weak link, but rather has been sufficiently fast and reliable so that its effect on system performance has been overlooked.

Looking at a real-time system from a design viewpoint, several important parameters can be identified.

Information. What is information with regard to basic mathematical processes being implemented? How can information be viewed within the constraints of the physical system? How can information be measured, quantitatively and qualitatively?

Information capacity, flow and/or generation. Is information generated, processed or merely compressed by the processes? Under what conditions does information flow occur? What factors affect the processing/generation or flow? Is capacity or levels of capacity identifiable?

The rate of information processing/production/flow. What is the relationship between the information being processed, generated or flowing and time? How is processing/generation/flow dependent on input/output rates, processor overhead?

Computational capacity. At what threshold rate can the system be driven before the digital component ceases to process or generate any information, or to allow information flow?

Computation with a fidelity criteria. At what system rate does the process generated information meet a specified qualitative criterion?

Bandwidth. Within what range is the computational component capable of processing or generating maximum information or allowing maximum information flow?

These idiosyncrasies common only to real-time applications which cannot be handled by traditional means are the deficiencies which the Real-Time Schemata was designed to study. Subsequent chapters will show that these problems can be analyzed in a structured manner yielding procedures to generate software independent modules that identify internal as well as external process dependent behavior, and a basis to analyze information production, loss and flow which will be the performance criterion of the real-time system under consideration.

REVIEW OF LITERATURE

Information and Graph Theoretic Contributions

Since a digital computer is an information processing device, it would seem natural that information theory in the Shannon sense could be adapted to model such devices. Rink [R2, R3] in a study of the effect of limited information capacity in digitally controlled linear systems suggested that the feedback loop might be modeled as an ideal controller and cascaded with a communication channel. Adrion, Rink and Sachs [A1] looked at some simple notions of "channel" capacity and rate as applied to similar systems with non-ideal controllers. Other studies examined the effect of finite wordlength in digital feedback control [R4].

The difficulties in handling the analysis of the digital component demonstrated in the work cited above, leads to the conclusion that a comprehensive theoretical framework for the study of digital processing devices is needed. This is particularly true in light of the increasing cost effectiveness of modern LSI digital components for use in real-time, on-line control.

Although much research has been done in applying such diverse disciplines as queueing theory, graph theory, simulation, automata theory and algorithmic theory to the analysis of digital computers, not much effort has been made to apply information theory. Large scale computer systems are extremely complex and do not lend themselves to simple modeling techniques except at a very gross level. However, the systems considered here, while generally complex, are constructed from simple

low cost, limited capacity, dedicated hardware components.

Shannon and Moore in their investigation of the reliability of logic circuits with unreliable components [M9] suggest indirectly that the concept of channel capacity might be applied to logic circuits. Elias [E1] attempted to do this for noisy ANDer circuits with the interesting result in his formulation that the computation rate for arbitrarily high reliability was zero.

The largest effort at applying information theory to unreliable logic nets was performed by Winograd and Cowen [W2]. Their work, a follow on from that of Von Neumann [V1], Moore and Shannon, and others, concentrates on finding a design technique for reliable networks of unreliable computational components. They showed that certain logic structures could be decomposed into a reliable computation element cascaded with a noisy channel. Given an ensemble of input messages X, computational element output ensemble Y and channel output ensemble Z, the following expression for mutual information can be derived

$$I(X;Z) = H(X) - H(X/Y) - H(Y/Z) \qquad (1)$$

The H(X) term in equation (1) represents the input information (entropy), H(Y/Z) the uncertainty due to noise in the channel and H(X|Y) the "loss" due to computation. Computational capacity was defined as

$$C = \max_{X} I(X;Z) \qquad (2)$$

Note that with a completely reliable logic (H(Y/Z)=0) the mutual

information and hence capacity is limited by the computation loss ($H(X|Y)>0$). Maximum capacity occurs for logic nets with no computation. $H(X|Y)$ is a measure of the inability to reconstruct the input to the computational element from knowledge of the output and not the quality of the output. This is to be expected since Shannon's entropy $H(X)$ is a quantitative measure on information.

Winograd and Cowen's work was useful in the design of reliable networks, but for analysis of the limiting effects of computational capacity in real-time systems, a more qualitative approach is required. A second problem with Winograd and Cowen's work is that they deal primarily with memoryless logic networks. If the effect of algorithm choice is considered, then the measure must account for state behavior and provide analysis of nets with memory.

Recently, Hellerman [H2] proposed an information work measure $w(f)$ on a process f over input ensemble X and output ensemble Y, $f:X\rightarrow Y$. The measure for a single step table lookup implementation of f was defined on the partition classes of inverse images under f.

$$X_i = \{x \epsilon X | f(x) = y_i\} \tag{3}$$

$$w(f) = \sum_{i=1}^{k} |X_i| \log_2 \frac{|X|}{|X_i|} \tag{4}$$

where $|X_i|$ and $|X|$ are the cardinality of the partition class X_i and total input set X respectively. Essentially Hellerman's work function is a measure of the difficulty of classifying an arbitrary input into one of the inverse image partition classes. If one notes that the probability of a specific y_i as output given a uniform distribution on

the inputs is

$$\text{prob}(y_i) = \text{prob}\{x \epsilon X_i\} = \frac{|X_i|}{|X|} = P_i \tag{5}$$

then

$$w(f) = -|X| \sum_{i=1}^{k} P_i \log_2 P_i = -|X| H(x) \tag{6}$$

Hence, the Hellerman's work function is an entropy measure on the output ensemble given a uniform input distribution. Again, w(f) is a quantitative measure. Hellerman applies his measure on implementations of f as several step procedures (synergisms) and derives an efficiency measure for comparison of algorithms. Hellerman's representation of multistep procedures, however, seems awkward for analysis of more complex algorithms, particularly when interrupts and looping are considered.

The only attempt to relate information theory to more complex algorithms is some preliminary work by Johnson [J1]. He makes use of both thermodynamic arguments and Petri graphs to analyze procedures.

Some kind of graph structure for the representation of real-time algorithms is necessary. Holt and Commoner [C6] in their excellent paper on Petri-like marked graphs discussed the meaning of information for a state machine represented by a marked graph. Information content was simply related to the state of the graph and hence to the probability of being at a particular node. The net change in information content when moving from one node X to another node Y is that supplied or lost during the transition and is given by P(X)/P(Y) where P(X) and P(Y) are the steady state probabilities of being in nodes X and Y respectively. Information is defined as the log of the information

content I and is given by $\log_2 1/P(X)$. Note that if the steady state probabilities are uniformly distributed, then for N states $P(X) = 1/N$ and the information content of state X reduces to $\log_2 N$, the number of bits to encode N states.

Neither Johnston's nor Holt and Commoner's formalism seems well suited to the representation of real-time algorithms. In searching for a suitable graph structure to represent both time and data dependent branching effects, the work of both Karp and Miller [K3] on parallel program schemata and Estrin and Martin [M1, M2, M3, M4, M5, M8] on directed bilogic graphs was considered.

With the parallel program schemata, Karp and Miller have developed a convenient formalism for the representation of processes. This work is a modification of the algorithm schemata proposed by Ianov [Y1] extended to include an interpreted graph model for computations.

Formally, a parallel schema $S = (M,A,T)$ consists of a set of memory locations M, operations A and a control graph $T = (Q,q_o,\Sigma,\tau)$. The schema is interpreted by specifying an operator C, which defines the memory contents for each memory location i an initial memory contents c_o and for each operation a in A two functions F_a and G_a. F_a specifies the actual operational mapping on memory and G_a calculates the conditional outcome.

In order to analyze the behavior during a computation (represented by a string over Σ^*), an instantaneous description $\alpha = (c,q,\mu)$ is defined. At any step in the computation, c specifies memory contents, q is the control graph state and μ is a set of queues which preserve information for an operation at the time of its initiation.

A computation is any defined string on the control graph which exhibits the finite delay property. This property, in essence, requires operations once defined to be initiated and once initiated to be terminated, both in finite time. A later paper [D2] shows that any determinate schemata must have the finite delay property.

Karp and Miller go on to characterize determinate schemata. A schema is repetition free, if, for a consecutive initiation of the same operation, the domain of that operation intersects the range of another operation which occurs between the repetition of the former operation. This condition must hold for all I-computations on the schema and essentially requires that any computation which initiates an operation multiple times must not be redundant. Persistency, commutativity and permutability are conditions on the allowed I-computations which require all defined operations to be initiated if a state is reached where they are defined or not to be initiated if they are not defined. Losslessness requires the operations to have a range. Schemata which are repetition free, lossless, persistent, commutative and permutable can be shown to be non-determinate only if Bernstein's conditions [B4] are violated. Bernstein's conditions simply state that the result of two operations depends on the order in which they are executed. Allowing them to occur in any order or in parallel leads to an indeterminate result.

Karp and Miller also consider problems of parallelism and introduce several classes of restricted schemata, including flow chart and counter schemata.

Although the parallel program schemata of Karp and Miller provides

a convenient formalism which can be utilized for the representation of algorithms, two concepts are missing for real-time systems. First, the notion of time is vague and secondly, if interrupts are introduced the finite delay property does not hold.

Estrin and Martin studied the execution time of algorithms using a special structure, the directed acyclic bilogic (dab) graph. This graph structure associates probabilities of branching with each node. Two types of node control are defined; conjunctive which allows parallelism and disjunctive which represents conditional branch nodes. An algorithm for mean path length is derived which is a function of the probability of traversing conjunctive subgraphs. Since the computational aspects of the algorithm are costly, an approximate heuristic procedure is also given.

Using a procedure developed by Elmagraby [E2], it can be shown that any directed computation graph can be transformed to a dab graph. Hence, Estrin and Martin's path length algorithm can be applied to real-time program graphs, though the resulting graph, when interrupts are allowed, is not finite state.

It is worth mentioning that several authors [C1, C2, D3, H1, K6, K7] have studied the meaning of information with respect to algorithms. Both automata theoretic computational complexity and control theoretic approaches have been taken. These studies have provided insight into this work but have not proven directly applicable.

Computer Programming Contributions

The art of computer programming can be extended for the study of

computation in real-time.

Real-Time Software

Software requirements for systems designed for real-time data acquisition and control vary according to the needs of the particular user, however, they can be categorized in terms of their function within the system. Real-time software will be one of three forms:

a. Applications software: a program in this category would generally perform a specific task. Examples in this category include solving equations, missile tracking, controlling a machine tool, or monitoring a process.

b. Software to aid development of applications software: this category includes assemblers, higher level language compilers, interpreters, and debugging tools.

c. Control or system software: the operating system, monitor or executive programs would comprise this category.

This thesis is essentially concerned with the analysis of applications software as implemented in a real-time environment. The two main features that distinguish real-time applications software from more general applications software are: 1) real-time imposes a time limit on the completion of a task or to service demands which may not reoccur, and 2) real-time processes are subject to external as well as internal interrupts.

Interrupts

There are two distinct classes of interrupts that can be related to real-time data processing. The class known as traps include all

those faults, that cause an abnormal exit from the currently executing process to another process initiated to check the fault. Conditions that can cause such an interrupt include all those unpredictable events that are specific to the running process; addressing faults, illegal instructions, protection violations, arithmetic overflows and interval time outs. These can be labeled internal interrupts since are generated by the internal system hardware or software. A typical response to this kind of interrupt would either be recovery from the error or termination of the process. External interrupts, those used for system input or output and peripheral device status, are generated by external sources or affect an external source. External interrupts can result from a processor being used to control a number of separate, but not necessarily mutually exclusive tasks.

In this environment, where the processor accepts responsibility for both main process control and interrupt process handling, interrupts are the only means to break out of the main processing sequence of operations. Interrupts are treated as high priority messages to or from external sources and by this relation to the main processing can be considered harmful due to their ability to degrade total system performance.

Identifying the source of the interrupt is typically handled by either polling, i.e. checking all devices on the interrupt lines to determine which one caused the interrupt, or by vectoring, i.e. an interrupting device sends a code word with the interrupt, usually a start address of the interrupt service routine plus the device priority. There are variations among machines but most interrupt

handling techniques fall into these two categories.

Since the probability of multiple interrupts is high there exists an interrupt scheduler, typically a hardware device, that identifies and initiates a process according to some queue discipline when an interrupt occurs. The run time priority of cooperating processes can be assigned on a static or dynamic basis. The interrupt handler makes the processor appear to the outside world as a hardware message queueing facility.

The queue discipline used to assign process order can take a number of forms. The most common is first-in, first-out abbreviated FIFO. Other common queue disciplines are last-in, first-out or LIFO, service in random order, SIRO, and priority service, PRI.

Most real-time processes operate under priority service queue disciplines. Each process is assigned a priority class number say from 1 to n, where the lower priority class number has higher process priority, i.e. processes of priority class i are given priority over processes in priority class j if $i < j$, and processes with priority class 1 have highest priority. Processes within the same priority class are usually serviced in order of arrival (FIFO).

In situations where a process with priority class i interrupts a process with priority class j and $i > j$, preemptive or non-preemptive priority control must be defined. In a preemptive priority queue, control passes immediately to the higher priority process. In a non-preemptive queue, the currently executing process retains control until completion, while the interrupting request waits regardless of its priority class. Variations of the preemptive priority distinction

include preemptive-resume, where a lower priority process, whose service was interrupted, resumes control at the point of interruption and preemptive-repeat where the lower priority process is restarted rather than reentered after the interrupt is serviced.

Structured Modular Design

Real-time software is typically application oriented. It is generally developed for a specific hardware configuration by a team of programmers. Design philosophies, even within a small group, vary. Complex tasks (i.e. processes composed of many sub-processes) can become intellectually manageable only if the overall program structure is well defined. Despite the lack of literature pertaining to the design of real-time software we can exploit some design techniques that have evolved to allow reasonable conjectures about constructing valid and efficient computer programs.

To reduce program size and complexity to levels of human manageability, approaches have been designed around the concept of modularity. One of the principle techniques, a process of step-wise decomposition of the problem with simultaneous step-wise refinement of the program was developed by Dijkstra [D]. The key to the success of this method is that by dividing a task into sub-tasks, the correctness or validity of the overall structure is dependent on the correctness of the components.

Dijkstras approach to the development of a structured program is implemented in a top-down manner. The top-down method of program design uses levels of program refinement. The top level is a descrip-

tion of the total process, specified in a general statement of the problem in some natural language like English. This allows a programmer to write a program without losing sight of the eventual goal. He is not constrained to a realm of machine dependent constructs and can express the problem in his own notation. In successive lower levels, each previous level is refined by division into substructures, concentrating on resolving critical points at that level rather than dwelling on issues that can be put off until lower levels. Dijkstra intended that refinements take the form of static improvements, those concerned with the data structure, and dynamic improvements, those concerned with the algorithm. At each level, the structure should still exhibit a correct and complete form of the original problem statement. When the refinement process is complete, each substructure is coded in the machine dependent target language.

A bottom-up approach is more or less a mirror image of the top-down method. This approach, though once popular, is not often endorsed for it has several disadvantages: coding lower order procedures first requires a main structure to be built around them. This early emphasis on detail often obscures the main objective and is constrained by machine dependent constructs. The resulting program may have a confusing logical flow which is often difficult to debug, maintain and modify.

Structured modular design in a top-down manner offers distinct advantages to the real-time programmer. Complex tasks can be specified in terms of independent sub-tasks. Debugging a program involves only verifying separately the behavior of individual sub-tasks. This

isolates errors to small blocks of code. Independent modular code provides protection against global side effects from local modifications or changes. This feature is an aid to program maintenance and gives the program some flexibility when changes are necessary.

Method of Top-Down Decomposition

To decompose a process into modules there are some structured programming constructs and input-output relationships we have to look for. Structured programs exhibit a form characterized by go-to less structural elements. To say there are no go-to's is a bit of an overstatement, since at some conceptual level branching backward or foreward is a necessary structural element. The if-then-else construct (Figure 10b) and the do-while (Figure 10c) are looping structures that have painfully eliminated the obvious go-to's. These constructs, along with a sequential one-step construct (Figure 10a) make up forms from which complex processes are composed.

Real-Time Process Description

The traditional flow chart is a simple but useful construct that is used to describe a process or an algorithm. This would typically outline the logic of the process in machine independent terms. Therefore, a process can be viewed, at a conceptual level, as a sequence of structural operations whose logical flow can be represented by some kind of flow diagram. We impose the following constraints to the class of processes we wish to analyze:

a. There should be no recursive flow,

b. no dynamically modified code that will change the flow structure, and

c. all successor operations of each process operation must be known or be deducible.

In order to maintain enough generality to discuss processes that have not been written in a particular programming language we use the concept of uninterpreted operations as the basic units of a programming structure. Operations can be broadly classed as conditional, using some criterion to select the next sequential instruction from two alternatives, or unconditional, implementing a sequential function and passing control to the next operation. A conditional instruction has a branch successor that is enabled when a condition is met, and a fall through successor that is enabled when the condition is not met. Not interpreting operations allows the model flexibility in its ability to describe a large class of operations one might encounter, from micro instructions to high level language functions.

Any sequence of legal operations, that is structurally well defined can be considered a process. Figure 4 represents a sample structured process in flow chart form and Table 1 represents the same process in tabular form. Each representation of the process shows the spatial relationships between each operation and its successor.

An analytic and descriptive formula description of the process, that retains the branching and looping substructure characteristics, can be generated as a regular expression of operations. The operator * (iterate) over an operation or a closed set of operations implies that the operation or operation set is to be executed zero or more times. It

is not uncommon to specify an integer n rather than * to indicate a fixed number of iterations.

The process described by Figure 4 is equivalent to the formula:

$$P = a_1[a_2a_3a_4a_5a_6(a_7+a_9a_{10})a_8]^*a_2a_3 \qquad (7)$$

This description provides both a path-wise and a module-wise description of a structured process. Branching and looping substructures are obvious by inspection of Equation 7. The operator + and implicit · are defined as logical union and disjunction respectively.

To further simplify the representation, the operations can be symbolized in terms of the groupings defined previously corresponding to fall through and branch successor relationships. A conditional operation a_i with its branch successor a_j and fall through successor a_k will be reduced by replacing $a_i(a_j+a_k)$ with a_i+a_i', where a_i is the fall through operation and a_i', the branch operation. This replacement will yield in the regular expression notation as:

$$P = a_1[a_2a_3a_4(a_5+a_5'a_6)a_7]^*a_2a_3' \qquad (8)$$

with the flow graph representation in Figure 5. Note that in the flow graph, the primed operation is implicit and occurs when the branch successor option is enabled.

Sub-paths in the graph can be determined by inspection of the regular expression formula, with the added feature, that specifies fixed or indefinite looping substructures, the * operator. With these features a regular expression descriptor of a process provides at least two levels of analysis. An a posteriori path analysis of any structured

process can be made, this can yield path length estimates, useful to determine execution time of a sequence of operations, and a structural analysis can identify modules within a complex process and exhibit relationships and dependencies between modules.

Complex Real-Time Processes

Structured real-time processes can be decomposed into modules by the scheme outlined in the preceding section. Computations performed, corresponding to the process, are concatenations of two forms of program modules; one-step and iterative. All processes can be constructed from these forms. A one-step process is one which reaches a terminal state in a known amount of time, excluding the time used to handle interrupts. A process of this class may not necessarily consist of one operation but at higher level of top-down decomposition can be represented as a one input, one output structured module with a constant execution time. A one-step process can be viewed mathematically as a mapping on the memory from the memory contents just prior to the initiation of the process to the memory contents at the logical completion of the process via the functional operations of the process.

In contrast to the one-step process, the iterative process, although viewed as a mathematical mapping with domain and range requirements, may not terminate in a fixed time interval. However, the process may reach a logical point, in time, where the domain may be sampled for an intermediate result. Further iterations of the process may or may not refine the answer and the logical termination of the process is subject to the imposition of a stopping criteria. The invocation of this

criteria is rather arbitrary. It may or may not be mapping dependent, i.e. it may be a qualitative or quantitative comparison of the process domain with the mathematically ideal range, or the process may terminate in a fixed time interval or after a number of iterations is fixed. This process type is analogous to using an iterative discrete function to approximate a continuous function.

Processes in the one-step class are relatively simple to analyze. Execution times, related to paths through the process, are well defined. Coupled with the execution time of a process is the reliability of the information a process produces. Deviation from the ideal output constitutes an information loss, it may be due to: 1) quantization loss from a continuous input space to a discrete space, 2) roundoff or truncation of data words due to word length, 3) a computation becoming input, output or compute bound and, 4) premature termination of a process to satisfy a time constraint.

Processes in the iterative class are somewhat more complex. Let us view a process as a means of implementing a mathematical function, (Figure 12) whose domain and range in the real world are continuous spaces. The function is implemented on discrete data with a discrete process, which is only an approximation to the continuous function. Thus the reliability of the process is dependent on two factors, the method of approximation (i.e. the goodness of the algorithm) and the time of convergence of the algorithm to an acceptable approximation. This convergence in a one-step process is fixed by the execution time, while in the iterative case it can be a function of the qualitative criteria and varies according to the parameters of the process or any

other imposed stopping criteria.

Topological Considerations

An appropriate mathematical topology to study real-time process maps has not been developed. There are immediate measure theoretic problems. The foundation of a topology is based on a continuous function f that is one-to-one on a problem space S such that the function f^{-1} exists. This map is not one-to-one in a finite memory model with fixed size word length, but many-to-one. To sample the problem space one cannot make an arbitrary choice of a sampling interval. Each point, i.e. each memory element, is an isolated point and has no immediate neighborhood. Measures on computed numbers, for operations as simple as addition, could not be defined due to closure violations, e.g. addition of two 16 bit numbers can yield a number not representable in 16 bits.

In order to develop a useful model for a real-time system some intuitive notion of the system configuration is necessary. One can take a very general approach and represent the system as an information processing device communicating with the real world via transducers as shown in Figure 13.

The model of Figure 14 represents the mathematical behavior of a real-time system with respect to algorithm behavior. Each sub-system is represented by a mapping from one mathematical space into another. Elements of a continuous (input) space are mapped under the composition of two functions, f_1 which discretizes the information and f_2 which encodes the information, into a portion of a discrete space. This dis-

crete space represents the computer memory. An algorithm is then performed, which could be a single step or multistep iterative or recursive procedure. The algorithm can be represented as a single or series of one-to-one and onto mappings of the discrete memory space back into itself. Finally the processed information is returned and represented by the compositional mapping $f_k \cdot f_{k+1}$ which takes elements of the memory space into the continuous output space. One advantage of such a model is that it can be compared with the ideal mathematical function being implemented by the digital sub-system (Figure 12).

To develop the schemata proposed here for the representation of real-time, on-line implementations of algorithms somewhat more detail is provided. The configuration shown in Figure 15 is similar to that of Figure 14, however two bus structures are provided to represent communication with the outside world. Many computers used in real-time applications are bus oriented and it is convenient to provide this feature in the proposed model.

REAL-TIME SCHEMATA

In this section the Real-Time Schemata are defined as structured hierachical graph models of a real-time process. The basic structure of the model was first proposed in [A2] and [A3] and the description here includes modifications necessary to analyze larger and more complex processes.

The Basic Model

Ianov [Y1] proposed a general schemata model which provides adequate analysis of computations where simple sequential control is assumed. Ianov's schemata formulation was modified by R. M. Karp and R. E. Miller [K3] to include parallel computations. Both of these schemata do not model the random interrupt in a computational sequence and avoid representing the time element in a computation. The Real-Time Schemata provides both these features.

Definition 1. A real-time computation schema is a triple $S=(M,\Omega,T)$ where:

i) M is a finite set of named memory locations $\{m_0,m_1...m_n\}$. Mathematically, the computer memory M is viewed as a discrete problem space, partitioned into three distinct but not necessarily mutually exclusive areas I_b the input buffer space, C_p the processing space and O_b the output buffer space. I_b is connected to the real-world by i_b the input bus and similarly O_b is connected to o_b the output bus.

ii) Ω is a finite set of elementary operations (mappings)

$\{a_1, a_2 \ldots a_k\}$. Associated with each $a_i \in \Omega$ are:

a) $D_{a_i} \subseteq M$, the domain locations;

b) $R_{a_i} \subseteq M$, the range locations;

c) $\gamma_i \in M$, a condition flip-flop used to determine whether the jump successor operation $a_i'(\gamma_i=0)$, or the fall through successor $a_i(\gamma_i=1)$ should be enabled.

iii) T is the control graph and $T = \{Q, q_0, q_I, Q_T, \Sigma, \mu, \delta\}$ where:

a) Q is a finite set of states $\{q_0, q_1 \ldots q_k, q_I\}$;

b) $q_0 \in Q$ is the unique initial state;

c) q_I is the interrupt state;

d) $Q_T \subseteq Q$ is a set of acceptor states;

e) Σ is an alphabet corresponding to the allowed set of operations, $\Sigma = \{\sigma_i = a_{i_1} a_{i_2} \ldots a_{i_k} | a_i \in \Omega\}$;

f) μ is a LIFO stack;

g) δ is a partial transition function $\delta: Q \times \mu \times \Sigma \rightarrow Q \times \mu$.

The sequence of allowed operations, corresponding to a computation in real-time, is enforced by the control graph T. The nodes in the graph represent the state of the system's memory and transitions from one state to the next in discrete time by the directed edges between nodes.

Construction of the Real-Time Schemata control graph for a given process is based on a program graph, similar to a flow chart, that represents the algorithmic part of the computation. The process allows interrupts at any node which are themselves processes. Multiple interrupts will be given priority by some preset queue discipline. Interrupt jumps and returns are handled in the interrupt state q_I. The control graph for an interrupt driven process would not necessarily be finite

state without use of an interrupt queue.

Figure 1 illustrates the interrupt state for a system with two interrupt processes. Since in general, the actual interrupt jump and return are handled by hardware, the transition to and from the interrupt state from any interrupted state q_i other than q_I is shown by a dotted bi-directional arc and has no labeled software operation. Once in the interrupt state the appropriate interrupt process is determined by polling, or any other means of identifying the interrupt and if the interrupt process can be enabled, the interrupted state is pushed onto a return state stack μ and the interrupt process is then initiated. When complete, a transition is made to the interrupt state where the return state is popped from the stack μ and control resumes at that state.

Example 1. Consider a simple real-time process corresponding to an interrupt driven, one-step table lookup procedure. The system has one input (i_b) and one output channel (o_b), and a small memory $M = \{m_0, m_1 \ldots m_{10}\}$. Figure 2 shows a program graph for Example 1 which does not employ the LIFO stack. Instead, it uses a last request priority where an interrupt may interrupt another. However, once a request is initiated by an interrupting process on a specified I/O channel, all further requests to that channel will be disabled until the original request is satisfied. Note that even for this simple one-step example a program graph becomes quite complex when nested levels of interrupts are considered. The graph allows the possibility of a countably infinite number of interrupts to occur. A situation where this is possible is when a computation becomes I/O bound. Hence the real-time program graph representing the total process behavior is not finite state.

To simplify the graph model the process can be modularized in a manner that divides the graph into two identifiable subgraphs, an algorithmic process and the remaining interrupt processes. Figure 3 illustrates this simplification for the table lookup example. This structure is equivalent to the structure in Figure 2 if each node is augmented by the interrupt subgraph.

More generally, any real-time process can be represented in a modular fashion, by identifying independent sub-processes in both the algorithmic and interrupt process subgraphs. More generally, any real-time process can be represented by a main control subgraph and one or more interrupt subgraphs.

A TIME DESCRIPTION OF PROCESS BEHAVIOR

A Partition On The Operation Set

In order to develop a description of the time behavior of a process, it is useful to classify the operation set.

Definition 3. The operation set Ω is partitioned in six distinct classes, $\Omega = (A,B,\Gamma,\Delta,E,Z)$, where:

i) $A = \{a_i \varepsilon \Omega \mid i_b \varepsilon D_{a_i}, \gamma_i=1\}$ is the set of input operations,

ii) $B = \{a_i \varepsilon \Omega \mid o_b \varepsilon R_{a_i}, \gamma_i=1\}$ is the set of output operations,

iii) $\Gamma = \{a_i \varepsilon \Omega \mid \gamma_i \varepsilon \{0,1\}\}$ is the set of conditional jump or branch operations. If $\gamma_i=1$, the fall through condition is met and operation a_i is enabled. Otherwise, $\gamma_i=0$, the branch condition is met and operation a_i' is enabled.

iv) $\Delta = \{a_i \varepsilon \Omega \mid i_b \not\varepsilon D_{a_i}, o_b \not\varepsilon R_{a_i}, \gamma_i=1\}$ is the set of data transformation operations, including arithmetic, logic, move and modify operations.

v) $E = \{a_i \varepsilon \Omega \mid \delta(q_i,\mu,a_i)=(q_k \cdot q_j \downarrow \mu), \gamma_i=1\}$ is the set of interrupt and subroutine jump operations[1],

vi) $Z = \{a_i \varepsilon \Omega \mid \gamma_i \varepsilon \{0,1\}\}$ is the set of conditional interrupt return operations if $\gamma_i=1$, the partial transition function $\delta(q_j,\mu,a_i)=(q_k,\uparrow\mu)$ determines the next enabled operation via the fall through operation a_i, otherwise $\gamma_i=0$ and the branch operation a_i' causes a system reset to state q_0.[1]

Processes in a modular structure may depend on other processes or external input for their initiation. This may take the form of a flag

[1] The notation $q_j \downarrow \mu, \uparrow\mu, \mu = q_k \cdot \mu'$ represent "pushing" q_j on stack μ and "popping" stack μ respectively where q_k is the top element of μ.

system that will not allow a process to continue until certain input conditions are satisfied. To symbolize this condition on the schema graph we use a double arrow leading into the node where such a condition is set (Figure 11a). A criteria we have set which will determine when a process yields useful information is the reaching of an acceptor state of the process. This event usually signals the availability of an output and will be symbolized on the schema graph with a double arrow leading out of a node (Figure 11b).

Interpretation of a Schema

In order to assign "meaning" to the evaluation of a calculation within a particular schema, a precise interpretation must be specified. The idea of an interpreted schema is similar to that of Karp and Miller [K3] with the addition of the notion of time.

Definition 2. An interpretation I of schema S with respect to time is specified by:

i) An initial time t_0, and discrete ordered, but not necessarily uniform time steps $t_1,\ldots t_k,\ldots$

ii) A function C which associates with each element $m_i \varepsilon M$, a set $C(m_i)$, the possible contents of m_i. The notation $[m_i(t_j)]$ represents a specific content of location m_i at time t_j, $[m_i(t_j)] \varepsilon\ C(m_i)$ and $[M(t_j)]$ the total contents of memory at time t_j, $[M(t_j)] \varepsilon \bigtimes_{m_i \varepsilon M} C(m_i)$

iii) An initial contents of memory $[M(t_0)]$

iv) For each $a_i \varepsilon \Omega$

a) the time to execute operation $a_i, \tau(a_i)$

b) a function $F_{a_i} : \bigtimes_{m_j \in D_{a_i}} C(m_j) \rightarrow \bigtimes_{m_j \in R_{a_i}} C(m_j)$

v) A function $G: \bigtimes_{m_i \in M} C(m_i) \rightarrow \bigtimes_{\gamma_i \in M} C(\gamma_i)$

The function F_{a_i} determines the outcome of an operation a_i, i.e. it stores the result in the range specified. Upon completion of an operation, the function G performs a test on the associated condition flip-flop γ_i to determine whether the jump successor or fall through successor operation is enabled next.

Table 1 gives an interpretation of operations for Example 1. The condition flip-flops are set as follows: $\gamma_i=1$ iff $[m_0]=0$, otherwise $\gamma_2=0$ and $\gamma_2=\gamma_3=\gamma_4=\gamma_5=\gamma_6=\gamma_7=\gamma_8=1$. The set of acceptor states $\{Q_T\}=\{q_1\}$. The interrupt priority is defined as last request, however an interrupt may not interrupt itself. For example, the operation specified by $\delta(q_i,\mu,a_3)$ is not enabled whenever the top element of the return state queue, μ_1, is q_2 or q_3 and similarly $\delta(q_i,\mu,a_6)$ is not enabled if μ_1 is q_4 or q_5.

This is a simple example, however, it is representative of any multistep algorithm with calculates a specific output or outputs from a single or block input without producing partial results.

Complex Processes

In many real-time applications complex algorithms, realized on the computer sub-systems which may consist of many component algorithms. The previous example given had only one component, yet the schemata as defined has the capacity to model processes of much greater complexity.

The basic schema graph can be partitioned into several subgraphs

P_o, P_1,...P_m, P_{I1}, P_{I2},...P_{In} where P_o, P_1...P_m are structured process modules of the main process algorithm and P_{Io}, P_{I1}...P_{In} are similar constructs based on the interrupt processes. The ordering of partitions in the interrupt substructure can be defined according to the interrupt priorities assigned to a specific interrupt module.

An alternate description of a structured modular process could be based on process flow, in terms of sequential ordering of processes according to some control structure and priority of processes.

Definition: A structured process P is a quadruple (q,P,Q_T,π) where:

i) q is the initial information state of the process, this can be the contents of memory;

ii) P is a regular expression of operations that describes all sequences of operations allowed in the process;

iii) Q_T is the state or set of states in which the process terminates;

iv) π is the process priority.

We specify that a simple structured process begins in an initial state q_o and each subsequent state, in time, is a function of an enabled operation at that time and the present state of the system. If the process terminates, there exists a final state where no further operations are enabled, otherwise there are no terminal states.

A structured complex process is defined in an analogous manner. Decomposition of a complex task into structured modules was discussed in an earlier section of this paper. To allow dependent modules we structure the independent parts such that the final state of one process is

the initial state of another. An interrupt process, which has a dynamic rather than fixed initial state, can be viewed in a similar fashion, i.e. an interrupt process begins in the state from which it was interrupted.

Example 2. Figure 6 is a flow chart description of a typical radar tracking algorithm that might be part of some real-time computer application.

A structured modular decomposition of the process in a top-down manner might yield a process flow as in Figure 7. Each sub-process yields a schema diagram (Figure 8) with an interpretation of operations in Table 3. A regular language descriptor of the process of Figure 7 is:

$$P = [P_0P_1P_2P_3(P_1P_2P_3)^*P_4((P_1P_2P_3)^*P_4)^*P_5(((P_1P_2P_3)^*P_4)^*P_5)^* P_6P_1P_2P_3P_4P_5(P_1P_2P_3P_4P_5)^*]^* \quad (9)$$

Each structured process module is described by the previously defined convention $P = (q,P,Q_T,\pi)$ in Table 4.

An interpretation of operations is given for this example in Table 3. In contrast to the previous example, where a specific range and domain was specified for each operation (Table 2), there are no entries. At this level of conceptualization, machine dependency has not been encountered. The process can be interpreted in such a manner if the top-down decomposition had continued to deeper levels. At this level, however, the process structure is apparent and input-output relations are known.

An Instantaneous Description

Once the distinction between operation classes has been introduced, some way of representing the instantaneous system behavior with respect to a specific string of operations must be developed. The concept of an instantaneous description similar to that used by Karp and Miller [K3] seems appropriate.

Definition 4. An initial instantaneous description is a quadruple $\theta(t_0)=(q_0,\psi(t_0)),[M(t_0),\mu]$ where q_0 is the initial state, $\psi(t_0)=\{a_i\varepsilon\Omega|\delta(q_0,\mu,a_i)$ is defined for $\gamma_i\varepsilon\{0,1\}$ is the initially enabled operations. $[M(t_0)]$ is the initial memory contents and $\mu=\phi$ is the initial LIFO stack state.

For a specific sequence $x\varepsilon\Sigma^*$, $x=a_{i1}a_{i2}\ldots a_{ik}$, the time to execute any subsequence $a_{i1}a_{i2}\ldots a_{ij}$ is given by $t_j=t_0+\sum_{\ell=1}^{j}\tau(a_{i\ell})$.

An instantaneous description at time t_j during the execution of sequence x is given by a quadruple $\theta(t_j)=\{q_\ell,\psi(t_j),[M(t_j)],\mu\}$ where:

i) q_ℓ is the present state;

ii) $\psi(t_j)$ is the set of enabled operations;

iii) $[M(t_j)]$ is the memory contents;

iv) μ is the LIFO stack state.

The instantaneous description $\theta(t_j)$ can be defined recursively.

Definition 5. Given a sequence $x\varepsilon\Sigma^*$, $x=a_{i1}a_{i2}\ldots a_{ik}$ and the instantaneous description after execution of $a_{ij}, j\leq k$, $\theta(t_j)=\{q_\ell,\psi(t_j),[M(t_j)],\mu\}$, a consecutive description $\theta(t_{j+1})=\theta(t_j+\tau(a_{ij+1}))=\{q'_\ell,\psi(t_{j+1}),[m(t_{j+1})],\mu'\}$ is defined iff $\delta(q_\ell,\mu,a_{ij+1})$ is defined. If so,

i) $(q'_{\ell},\mu')=\delta(q_{\ell},\mu,q_{ij+1})$;

ii) $[M(t_{j+1})] = F_{a_i}[M(t_j)]$;

iii) $\psi(t_{j+1}) = \{a_i \varepsilon \Omega \mid \delta(q'_{\ell},\mu',a_i)$ is defined. Operations in $\psi(t_{j+1})$ are enabled by the condition flip-flops set by $[(\gamma_1,\gamma_2,\ldots\gamma_n)] = G([M(t_{j+1})])\}$.

PROPERTIES OF THE SCHEMATA

Some Properties of Processes

A real-time computation corresponds to a well defined sequence of operations, that begins in the initial state and either iterates through or terminates in some designated acceptor state. An acceptor state in the graph model, represents a logical point in the computation where the calculation on input information is in an acceptable condition for output. This may correspond to a terminal state in a fixed length algorithm or an appropriate sample state in an iterative algorithm. Acceptability is a function of the "useful information" calculated in a process.

Definition 6. A string $\sigma\epsilon\Sigma^*$ is a well defined sequence of operations (w.d.s.) for a schema S is all prefixes β of σ, $\delta(q_0,\mu,\beta)$ is defined on S, where $\mu(t_0)=\emptyset$.

If $\ell(\sigma)$ is the length of the string σ (e.g. if $\sigma=a_{i1}a_{i2}\ldots a_{ik}$, $\ell(\sigma)=k$) then the following may be stated.

Definition 7. The set of k-length well defined sequences for a schema S is a finite set $\Sigma^k=\{\sigma\epsilon\Sigma^* \mid \ell(\sigma)=k$ and, σ is well defined$\}$.

Definition 8. Let $\sigma_i=a_{i1}\ldots a_{ik}$ be a k-length sequence in Σ^k for a schema S. The ordered set of states $q_i=q_0,q_{i1}\ldots q_{ik}$, of length k+1 is the q-set associated with the computation σ_i, where q_0 is the initial state and for $i\leq j\leq k$, $q_{ij}\epsilon Q$, $a_{ij}\epsilon\sigma_i$ and $\delta(q_{ij-1},\mu,a_i)=(q_{ij},\mu')$.

Definition 9. A useful calculation occurs in a sequence $\sigma_i\epsilon\Sigma^K$ under schema S if $\{q_i\}$, the associated q-set, contains at least one element $q_i\epsilon Q_T$, the set of acceptor states.

The q-set concept is essential for the analysis of useful information production by a computation. For any sequence of operations, an associated q-set may be generated. An a posteriori analysis of this q-set with reference to the occurrences of acceptor states reflects the "useful information" generated by the process.

In the example of Figure 3, $\sigma_0=a_1a_3a_4a_5a_1'a_6a_7a_8$ is a well defined sequence. The associated q-set is $\{q_0\}=\{q_0q_0q_2q_3q_0q_1q_0q_4q_5q_0\}$ and a useful calculation is performed when state q_1 is reached.

Earlier definitions introduce the concept of length bounded strings. In order to develop a notion of time bounded strings and information production rates, the following are defined.

Definition 10. The set of well defined sequences bounded by time t is defined $\Sigma_t=\{\sigma\varepsilon U\Sigma^i \mid \tau(\sigma)\leq t\}$.
i

It is well known that both Σ^k and Σ_t form a language of regular expressions.

Definition 11. A <u>trace</u> on Σ_t is a set of q-sets $T_t=\{\{q_i\}\mid\sigma_i\varepsilon\Sigma_t\}$.

Definition 12. An <u>incremental trace</u> over time interval $T_{t,\hat{\tau}}=\{\{v\}\mid\{q_i\}\varepsilon T_{t-\hat{\tau}}\ \{q_i,v\}\varepsilon T_t\}$ where $\{q_i,v\}$ is a q-set which consists of the ordered set $\{q_i\}$ followed by the ordered set of states $\{v\}$.

Examination of the elements of $T_{t,\hat{\tau}}$ will indicate generation of useful information during the time interval $(t-\tau,t)$. As was suggested above, useful information depends on the occurrence of acceptor state in a particular $\{v\}$. Mere occurrence of an acceptor state does not in itself guarantee the generation of useful information, but serves to indicate that information was calculated.

Information Flow in a Schemata

The q-set and incremental trace concepts provide a means for determining when a calculation has occurred for a given w.d.s. σ. Since the definition of real-time implies non-reproducible inputs, some way of gauging whether input data is utilized completely or lost is necessary.

Given a time t, a schema S and a set of time bounded w.d.s. Σ_t, a set of strings $\hat{\Sigma}_t \subseteq \Sigma_t$ is said to be input-preserving if the information input, as represented by a particular string, is available for all operations for which it is intended. This "definition" is purposely vague, for it depends completely on the algorithm and the environment where input information is utilized. Obviously, a string which includes two successive input operations with the same range locations is not input preserving unless the input information is not needed for any operation. The presence or absence of a "useful" calculation, as defined by the occurrence of acceptor states within the q-set for a string or incremental trace for the time interval between two inputs in a string, does not guarantee that input is or is not preserved. However, by examining the input-output relationship between operations some sufficient conditions for strings to be input preserving can be derived. For Example 1, a string $\sigma = a_1a_3a_4a_5a_1'a_2a_1$ is input preserving where $\sigma = a_1a_3a_4a_5a_3a_4a_5a_1'a_2a_1$ is not.

Definition 13. A kxk dependency matrix at time t for a schema S with operation set $\Omega=\{a_1,a_2,\ldots,a_k\}$ is $\Lambda^t=[\lambda_{ij}^t]_{\substack{i=1,\ldots,k \\ j=1,\ldots,k}}$ where $k_{ij}=1$ iff

i) $R_{a_i} \cap D_{a_j} \neq \emptyset$;

ii) There exists a $\sigma \in \Sigma^t$ such that for $x,y,z \in \Sigma^*$:

1. $\sigma = xa_iya_jz$ and;

2. For all $y'\varepsilon\Sigma^*$, $a_m\varepsilon\Omega$ such that $y'a_m$ is a prefix of y;

$$R_{a_m} \cap R_{a_i} \cap D_{a_j} = \emptyset.$$

Hence, an operation a_j is said to depend on a_i if a_i possibly produces information used by a_j and for some w.d.s. σ, a_j does not use that information. From well known graph theoretic techniques [R1] the set of operations dependent on a particular operation a_i over time t, $D^t_{a_i}$ can be determined since Λ^t is a restricted incidence matrix for the graph (Ω,H^t) where $H^t(a_i)=a_j$ iff $\lambda^t_i = 1$. $D^t_{a_i}=\{a_j\varepsilon\Omega|$ there is a path from a_i to a_j in $(\Omega,H^t)\}$.

An operation $a_i \notin \Gamma UEUZ$ is trivial (processes no information) if $D^t_{a_i}=\emptyset$.

Remark: Sufficient conditions for $\sigma\varepsilon\Sigma^t$ to be input preserving are: For all $x\varepsilon\Sigma^*$, $a_i\varepsilon A$, such that xa_i is a prefix of σ, a_i nontrivial.

i) There exists a $ye\Sigma^*$, $a_j\varepsilon D^t_{a_i}$ such that xa_iya_j is a prefix of and $\delta(q_0,\emptyset,xa_iya_i)\varepsilon Q$, and

ii) For all $y'\varepsilon\Sigma^*$ such that either $\sigma=xa_iy'$ or $xa_iy'a_k$, $a_k\varepsilon A$ is a prefix of σ and for which there is no y'' for which $y''a_\ell$ is a prefix of y', $a_\ell\varepsilon A$, $R_{a_i} \subseteq R_{a_\ell}$, there exists no $z\varepsilon\Sigma^*$, $a_m\varepsilon D^t_{a_i}$ for which $\delta(q_0,\emptyset,xa_iy'za_m)\varepsilon Q_T$.

The condition requires that at least one useful calculation on each nontrivial input be performed and furthermore that all defined calculations are preformed by the end of the sequence or prior to any other input which writes on the same range memory locations.

The other vital concern in a real-time system is that output information is available, when needed, to implement positive control. Information flow is said to occur for some $\sigma\varepsilon\Sigma^t$ $\sigma=a_{i1}a_{i2}\cdots a_{ik}$ under

schema S, if the information input by some nontrivial operation ($a_{ij}\varepsilon\Lambda$) is processed, either partially or completely ($a_{i\ell}\varepsilon D^{t}_{a_{ij}}$, $\ell>j$, $\delta(q_0,\emptyset,a_{i1}\ldots a_{i\ell})\varepsilon Q_T$) and is output ($a_{\ell m}\varepsilon B, m>\ell$, $R_{a_{i\ell}}\cap D_{a_{im}}\neq\emptyset$).

An exact definition of information flow is difficult to state in general, since much depends on the input and output interrupt rates, the algorithm and other factors. However, occurrence of information flow can be related back to the incremental trace concept.

Remark: A sequence $\sigma\varepsilon\Sigma^{t}$, $\sigma=a_{i1}a_{i2}\ldots a_{k}$, represents information flow only if after some time t_j, $a_{ij}\varepsilon B$ and for some time t_ℓ, $a_{i\ell}\varepsilon\Lambda$, $t_\ell<t_j$, $\varepsilon T_{t_j,t_j-t_\ell}$, where ν is incremental trace on σ, and $\{\nu\cap Q_T\}\neq\emptyset$.

Input preserving sequences, as loosely defined above, are sequences for which input information is completely processed as defined by the algorithm represented by the schema. Sequences which represent maximum information flow would necessarily be input preserving. The output sequences within these sequences must occur at appropriate intervals such that the processed information is output.

Example 3. Consider $F(t,\hat{\tau})=\text{Prob}\,\{\{\nu\}, R(\nu)\}$ where $\{\nu\}$ is an element of $T_{t,\hat{\tau}}$ for some t and $\hat{\tau}$ and $R(\nu)$ is a rule for determining when useful information flow occurs. The probability distribution $F(t,\hat{\tau})$ characterizes the useful information produced in the interval $(t-\hat{\tau},t)$.

For the example of Figure 3, a possible rule R would require t to be the time of an input interrupt and $\hat{\tau}$ the time elapsed since the last prior input interrupt. The decision would then be based on whether ν contains an acceptor state. The probability distribution $F(t,\hat{\tau})$ for this rule, assuming $\tau(a_i)=\tau_0$ for all a_i, is given by Figure 16.

IMPLICATIONS

Analysis of Processed Information

The proposed model of a real-time process yields a structured hierarchy of cooperating processes with convenient formalisms to analyze information production behavior.

The physical system is modeled as a mathematical mapping of information via the digital processing and system. The memory space M and the operation set Ω are partitioned with respect to this map. M is divided into an input buffer space I_b, computation space C_p and output buffer space O_b. The size of these sub-spaces at a particular time instant is indicative of system loading such as I/O or computation bound processing states. The operation partitioning is convenient for determining system behavior.

Several concepts are developed which relate to the information content, flow and processing represented by an operation string. These include: q-set, incremental trace, length and time bounded sets of operation strings, input preserving strings, "useful" calculations and information flow. A schema specifies a set of states where intermediate or final results are produced, the acceptor states. The q-set and incremental trace features provide a means for determining the inclusion of such states in schema behavior over time for a particular operation sequence. The characterization of strings as elements of length and time bounded and input preserving sets enables analysis of the probability of achieving predicted performance. The occurrence of "useful" calculations and/or information flow for specific sequences is also

indicative of algorithm performance with given I/O demand.

If attention is limited to systems with on-line, real-time minicomputer/microprocessor sub-systems, the term "information" can be applied both to qualitative and quantitative aspects. In any such system, there is an identifiable data flow. Data is acquired by the digital sub-system in several states involving sampling, A/D conversion, encoding transmission and storage. A rate of information flow can be derived which depends on the sampling rate, the I/O channel bandwidth, the memory cycle time and the cpu cycle time. Qualitatively, the information acquired is subject to error due to the sampling and encoding processes, and the internal word length/character size.

In a similar fashion, information processing can be quantified by considering cpu cycle time, memory fetch time, average instruction execution time, the complexity of the algorithm, etc. Sources of error include round-off, truncation, lack of precision, arithmetic limitations, algorithm accuracy, and others. Of course, data returned to the system or the user can be analyzed in the same way.

Such representations as suggested above can be made as simple or as complex, as meaningful or as meaningless as desired. Although the points raised are valid and demand consideration, they are not specific to the approach to be taken in this research. Real-time implies that an immediate, dynamic response is expected of the digital system to its environment. Of course, this is impossible. The ideal system, represented by Figure 12, provides an instantaneous complete mathematical mapping (reaction) from one continuous space to another. In reality, time must be allowed to acquire, process and return the data.

Qualitatively, the information that is returned to the system or user is of most interest. The ideal system would react immediately (produce an element of the output space) to each environmental change (input space) in the manner specified by the system designer (the mathematical map). Although the selection of the map is of extreme importance, it is not central to this study except, perhaps, through development of a method for comparing the effectiveness of choices made. Assuming a specified ideal, the areas of concern are the choice of a discrete algorithm to perform this function, the acquisition of data to be processed and the return of processed information within the constraints of real-time and system capability.

Information Networks

The "state of the art" in computer design changes quickly in this era of LSI and MOS technology, and a trend can be forecast for the decreasing use of large general purpose computers and the increased use of distributed multi, micro or mini processors in network configurations where each processing element in the system is responsible for one or more specific tasks.

The Real-Time Schemata can represent any hierarchy of structured processes and identifies independent sub-processes. These processes may be analyzed in a sequential implementation, under simple parallel processing or in a multiprocessing environment with interprocess communication provided through the interrupt structure. Shared resource multiprocessing can be represented provided adequate safeguards for memory protection, deadlocks and mutual exclusion are implemented.

The inherent modularity of the process representation enables any software-firmware structure to be mapped to a variety of physical processor hardware configurations. These include single processors, multiprocessor networks with both central and distributed control and redundant, reconfigurable multiprocessor systems. Both independent and shared I/O processing can be analyzed.

For a specific software-firmware structure, the model along with the parameters of a given hardware implementation will allow performance characteristics to be studied. Since one particular process structure can be analyzed as implemented with different processor configurations, a notion of the class of possible failure tolerant redundant configurations which provide a minimum level of performance might be derived.

With all systems operational the modular software structure (schemata) would suggest a number of task assignment maps onto the hardware configuration; with a "best" map obtainable given a specific performance criterion.

In the event of one or more processor (other sub-system) failures the modular software structure now presents the possibility of selecting an alternate map (reassignment of task allocation on the hardware) to keep the overall system operational, with minimum loss in performance.

BIBLIOGRAPHY

A1 W. Adrion, R. Rink and S. Sachs, "Computational Capacity in Digital Systems," Proceedings of the 1975 ACM Computer Science Conference, February 1975.

A2 W. R. Adrion, P. A. Frick and P. A. Szulewski, "Analysis of Real Time Systems: An Information Theoretic Approach," Proceedings of the 13th Annual Allerton Conference on Circuit and System Theory, October 1975.

A3 _____ "Real Time Schemata," Proceedings of the Ninth Annual Asilomar Conference on Circuits, Systems and Computers, November 1975.

A4 L. P. Akimov, "Realization of Complex Algorithms of Large-Scale Control Systems in Real Time," Tekhnicheskaya Kibernetika (Engineering Cybernetics), Vol. 12, May-June 1974.

A5 A. O. Allen, "Elements of Probability for System Design," IBM Systems Journal, 13, No. 4, 325-348, 1974.

A6 _____ "Elements of Queueing Theory for System Design," IBM Systems Journal, 14, No. 2, 161-187, 1975.

B1 J. L. Baer and G. Estrin, "Bounds for Maximum Parallelism in a Bilogic Graph Model of Computations," IEEF-C, C-18, November 1969, 1012.

B2 J. L. Baer, D. P. Bovet and G. Estrin, "Legality and Other Properties of Graph Models of Computations," J. ACM, 17, No. 3, July 1970, 543-554.

B3 T. Berger, Rate Distortion Theory, Englewood Cliffs, New Jersey, Prentice-Hall, 1971, "The Source Coding Game," IEEE-IT-17, January 1971.

B4 A. J. Bernstein, "Analysis of Programs for Parallel Processing," IEEE-EC, October 1966.

B5 J. A. Breozowski, "A Survey of Regular Expressions and their Applications," IRE Transactions, EC-71, June 1962.

C1 G. J. Chaitin, "Information Theoretic Computational Complexity," IEEE-IT-20, January 1974.

C2 _____ "On the Difficulty of Computations," IEEE-IT-16, January 1970.

C3 H. Y. Chong, "Information Processing Errors in Linear Feedback Control Systems," Oregon State University, Doctoral Dissertation, June 1976.

C4 J. C. Cluey, Computer Interfacing and On-Line Operation, Crane, Russac and Co., 1975.

C5 F. Commoner, A. W. Holt, S. Evan and A. Pnoeli, "Marked Directed Graphs," J. Computer System Science, Vol. 5, October 1971.

C6 F. Commoner and A. W. Holt, "Events and Conditions," in Record of the Project MAC Conference Consurrent Systems and Parallel Computations, New York: Ass. Comput. Mach., 1970.

C7 I. Copi, C. Elgot and J. Wright, "Realization of Events by Logical Nets," J. ACM, 5, April 1958.

D1 O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare, Structured Programming, Academic Press, 1972.

D2 P. J. Denning, "On the Determinancy of Schemata," Record of the Project MAC Conference on Concurrent Systems and Parallel Computations, ACM, New York, 1970, 143-147.

D3 A. De Luca and S. Termini, "Algorithmic Aspects in Complex Systems Analysis," Scientia, July-August 1971.

E1 P. Elias, "Computation in the Presence of Noise," IBM Journal, October 1958.

E2 S. E. Elmaghraby, "An Algebra for the Analysis of Generalized Activity Networks," Mgmt. Sci., April 1964, 10, 494-514.

G1 R. G. Gallager, Information Theory and Reliable Communication, New York: Wiley, 1968.

G2 H. W. Gottinger, "Complexity and Information Technology in Dynamic Systems," Kybernetica, Vol. 4, 1975.

G3 R. M. Gray, ed., "Source Coding/Ergodic Theory Research Seminar Lecture Notes," Stanford Information Systems Laboratory, Tech. Rep. No. 6503-2.

H1 J. Hartmanis and J. E. Hopcroft, "An Overview of the Theory of Computational Complexity," J. ACM, Vol. 18, No. 3, July 1971.

H2 L. Hellerman, "A Measure of Computational Work," IEEE-C-21, May 1972.

I1 M. A. Ikezawa and R. E. Kaytes, "A Structural Calculus for Program Analysis and Testing," Proceedings of the Ninth Annual Asilomar Conference on Circuits, Systems and Computers, November 1975.

J1 R. R. Johnson, "Some Steps Toward an Information System Performance Theory," Performance Evaluations Review, ACM, September 1972.

K1 R. M. Karp, "A Note of the Application of Graph Theory to Digital Computer Programming," Info. and Cont., June 1960, 3, 179-190.

K2 R. M. Karp and R. E. Miller, "Properties of a Model for Parallel Computation: Determinancy, Termination, Queueing," SIAM Journal of App. Math., November 1966, 14, 1390-1411.

K3 R. M. Karp and R. E. Miller, "Parallel Program Schemata," J. ACM, 1969, 3, 147-195.

K4 S. C. Kleene, "Representation of Events in Nerve Nets and Finite Automata," Automata Studies, C. E. Shannon, J. McCarthy, ed., Princeton University Press, 1956.

K5 Z. Kohavi, Switching and Finite Automata Theory, McGraw-Hill, 1970.

K6 A. N. Kolmogorov, "Three Approaches to the Quantative Definition of Information," Problemy Peredachi Informatsii, 1,3, 1965, reprinted in the International Journal of Computer Math., 2, 157, 1968.

K7 A. N. Kolmogorov, "Logical Basis for Information Theory and Probability Theory," IEEE-IT, 15, 1968.

L1 H. F. Ledgard, The Programming Proverbs, Hayden Book Co., New Jersey, 1975.

L2 _____ "The Case for Structured Programming," University of Massachusetts COINS Report, Amherst, Massachusetts, 1973.

M1 D. Martin and G. Estrin, "Models of Computations, Cyclic to Acyclic Graph Transformations," IEEE-EC, February 1967, 70-79.

M2 _____ "Experiments on Computations and Systems," IEEE-EC, February 1967, 59-69.

M3 _____ "Models of Computations and Systems--Path Length Computations," Computer Respository, R67-61, March 1967.

M4 _____ "Models of Computations and Systems, Evaluation of Vertex Probabilities in Graph Models of Computations," J. ACM, April 1967, 281-299.

M5 _____ "Path Length Computations on Graph Models of Computations," IEEE-EC, June 1969, 530-536.

M6 J. Martin, Programming Real-Time Computer Systems, Prentice-Hall, 1965.

M7 R. McNaughton and H. Yamada, "Regular Expressions and State Graphs for Automata," IRE Transactions, EC-9, March 1960.

M8 R. E. Miller, "A Comparison of Some Theoretical Models for Parallel Computation," IEEE-C, August 1973, 22.

M9 E. F. Moore and C. E. Shannon, "Reliable Circuits Using Less Reliable Relays," Jour. Franklin Inst., 262, 1956.

N1 J. D. Noe and G. T. Nutt, "Macro E-Nets for the Representation of Parallel Systems," IEEE-C, August 1973, 22, 718-727.

N2 G. T. Nutt, "E-Nets for Computer System Performance Analysis," AFIPS Conf. Proc., Vol. 41, 1972 Fall Proceedings.

P1 M. S. Paterson, "Program Schemata," Machine Intelligence, Vol. 3, American Elsevier, New York, 1968, 18-31.

R1 C. V. Ramamoorthy, "Analysis of Graphs for Connectivity Considerations," J. ACM, 13, No. 2, April 1966.

R2 R. E. Rink, "Optimal Utilization of Fixed-Capacity Channels in Feedback Control," Automatica, Vol. 9, 251-255, Pergamon Press, 1973.

R3 _____ "Coded Data Control of Linear Systems," Proc. 8th Princeton Conference on Information Sciences and Systems, Princeton University, 1974.

R4 R. E. Rink and H. Y. Chong, "Finite Wordlength Effects in Digital Control," Proceedings of the Canadian Conference on Automatic Control, June 1975.

R5 P. A. Robichaud, M. Boisevert and J. Robert, Signal Flow Graphs and Applications, Englewood Cliffs, New Jersey, Prentice-Hall, 1962.

R6 I. Rubin, "Reduced Memory Likelihood Processing of Point Processes," IEEE-IT-20, November 1974, 729-737.

R7 _____ "Communications Networks: Message Path Delays," IEEE-IT-20, November 1974, 738-745.

R8 _____ "Regular Jump Processes and Their Information Processing," IEEE-IT-20, September 1974, 617-624.

R9 _____ "Rate Distortion Functions for Nonhomogeneous Poisson Processes," IEEE-IT-20, September 1974, 669-672.

R10 _____ "Optimal Sequence Estimators for Statistically Unknown Binary Source and Channels, IEEE-IT-21, No. 2, March 1975, 217-221.

R11 _____ "Information Rates and Data Compression Schemes for Poisson Processes," IEEE-IT-20, March 1974, 200-210.

R12 J. D. Rutledge, "On Ianov's Program Schemata," J. ACM, January 1974, 11, 1-9.

S1 S. Saffer and D. Mishelevich, letter to the editor printed in Communications of the ACM, Vol. 18, No. 9, September 1975.

S2 J. E. Savage, "Computational Work and Time on Finite Machines," J. ACM, Vol. 19, No. 4, October 1972.

S3 _____ "Computational Work and Time," in Computational Complexity, Rustin and Randall, eds., 103-111, 1973.

S4 _____ "The Efficiency of Algorithms and Machines - A Survey of the Complexity Theoretic Approach," 1973, Proceedings AFIPS.

S5 C. E. Shannon and W. Weaver, The Mathematical Theory of Communication, Urbana: University of Illinois Press, 1949.

S6 C. E. Shannon, "Communication in the Presence of Noise," Proc. IRE, 37, No. 1, January 1949.

S7 D. R. Slutz, "Flow Graph Schemata," Record of the Project MAC Conference on Concurrent Systems and Parallel Computations, ACM, New York, 1970, 129-141.

T1 J. F. Traub, "Computational Complexity of Iterative Processes," SIAM J. of Computing, 1, 2, 1972.

V1 J. Von Neumann, "Probablistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," Annals of Mathematical Studies, 34, 43-98, 1956.

W1 C. Wietzman, Minicomputer Systems, Structure, Implementation and Applications, Prentice-Hall, 1974.

W2 S. Winograd and F. D. Cowen, "Reliable Computation in the Presence of Noise," MIT, Monograph, 1963.

W3 N. Wirth, "On the Composition of Well Structured Programs," ACM Computing Survey, 6, No. 4, December 1974.

W4 J. Wozencraft and I. Jacobs, Principles of Communication Engineering, New York, Wiley, 1965.

W5 A. D. Wyner, "Recent Results in the Shannon Theory," IEEE-IT-20, January 1974.

Y1 Y. I. Yanov (Ianov), "On Equivalence and Transformation of Programs," Comm. ACM, October 1958, 10.

Y2 _____ "The Logical Scheme of Algorithms," Problems in Cybernetics, Vol. 1, Pergamon Press, 1960, 82-140.

Y3 _____ "On Matrix Program Schemes," Comm. ACM, December 1968, 1, 3-6.

APPENDICES

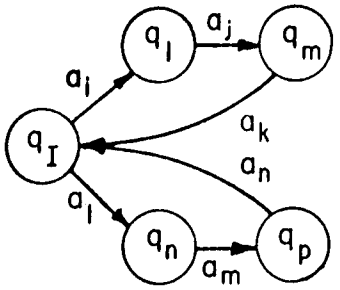

FIG. 1

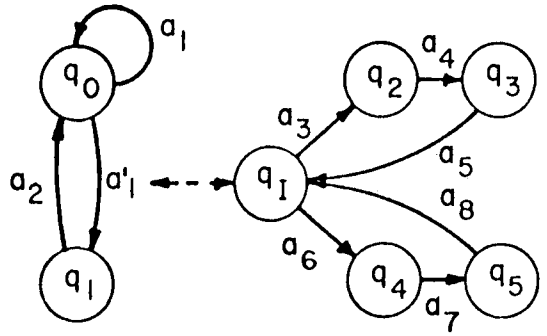

FIG. 3

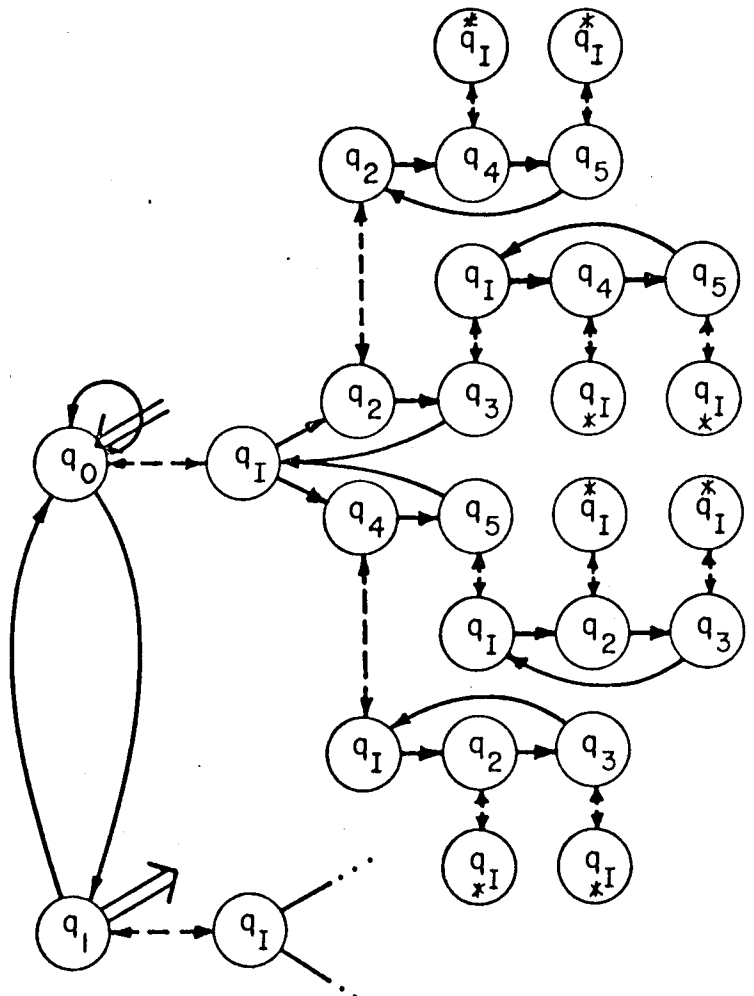

FIG. 2

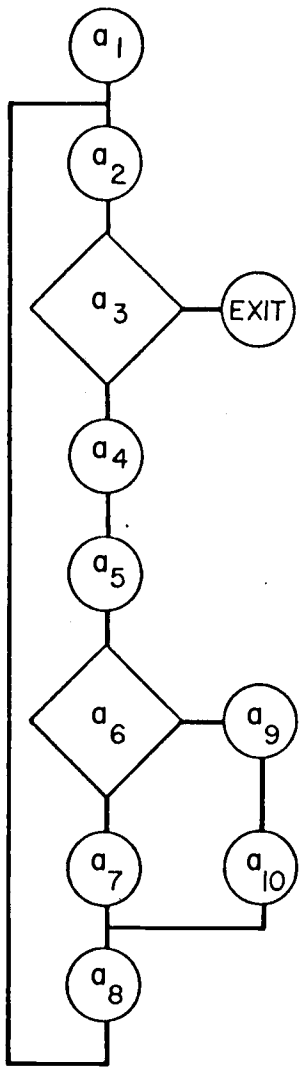

FIG. 4

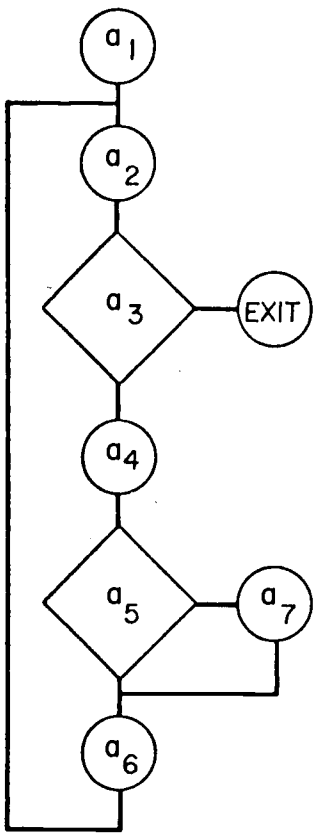

FIG. 5

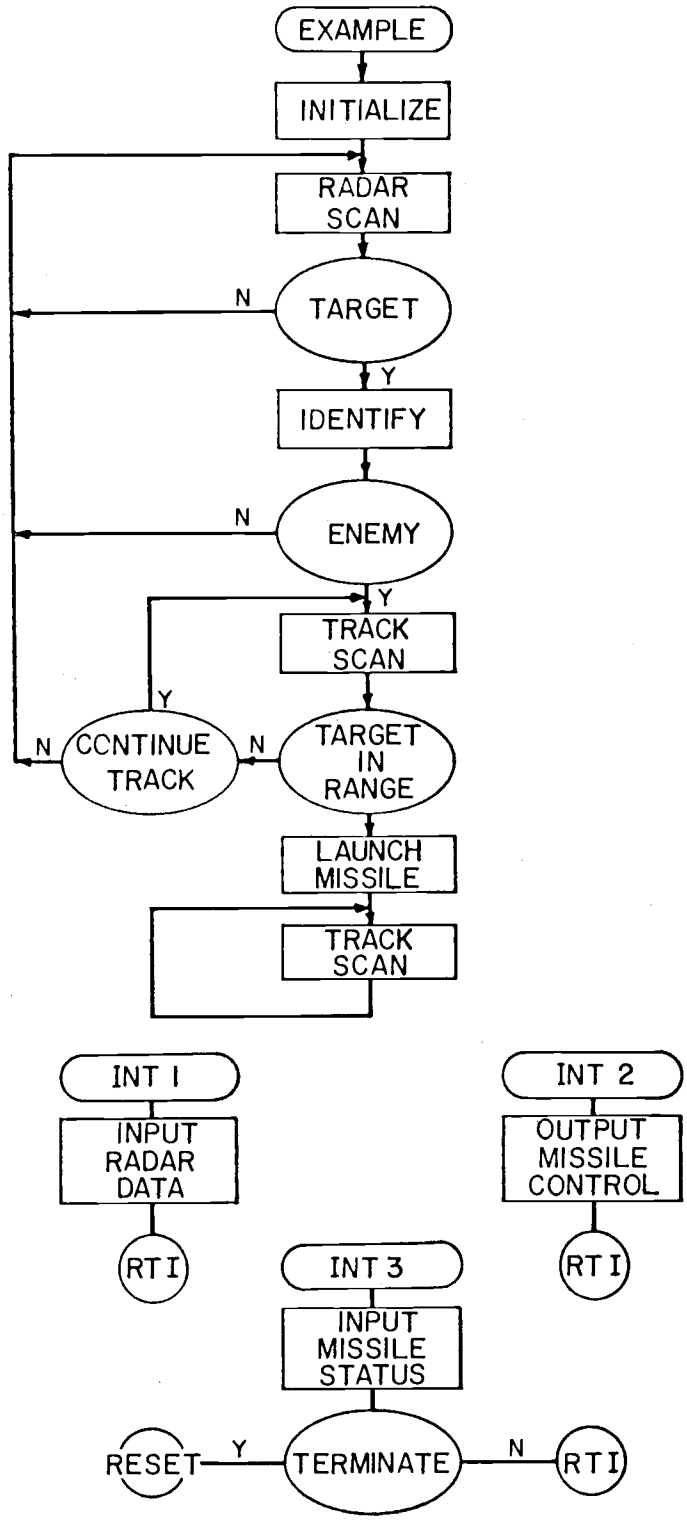
EXAMPLE
INITIALIZE
RADAR SCAN
TARGET
N
Y
IDENTIFY
ENEMY
N
Y
TRACK SCAN
Y
CONTINUE TRACK
N
N
TARGET IN RANGE
LAUNCH MISSILE
TRACK SCAN
INT 1
INPUT RADAR DATA
RTI
INT 2
OUTPUT MISSILE CONTROL
RTI
INT 3
INPUT MISSILE STATUS
RESET
Y
TERMINATE
N
RTI

FIG. 6

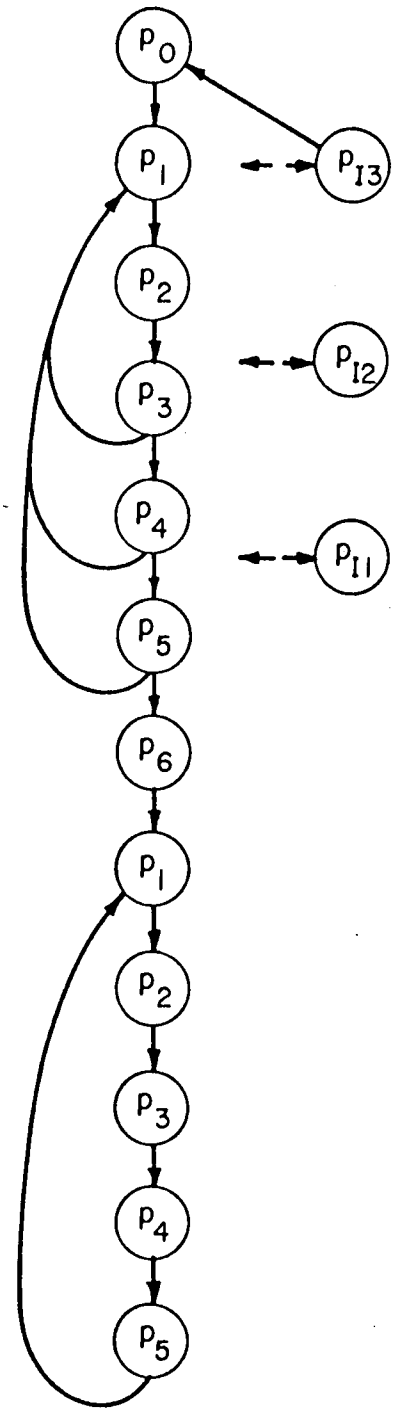
P_0
P_1
P_{13}
P_2
P_{12}
P_3
P_4
P_{11}
P_5
P_6
P_1
P_2
P_3
P_4
P_5

FIG. 7

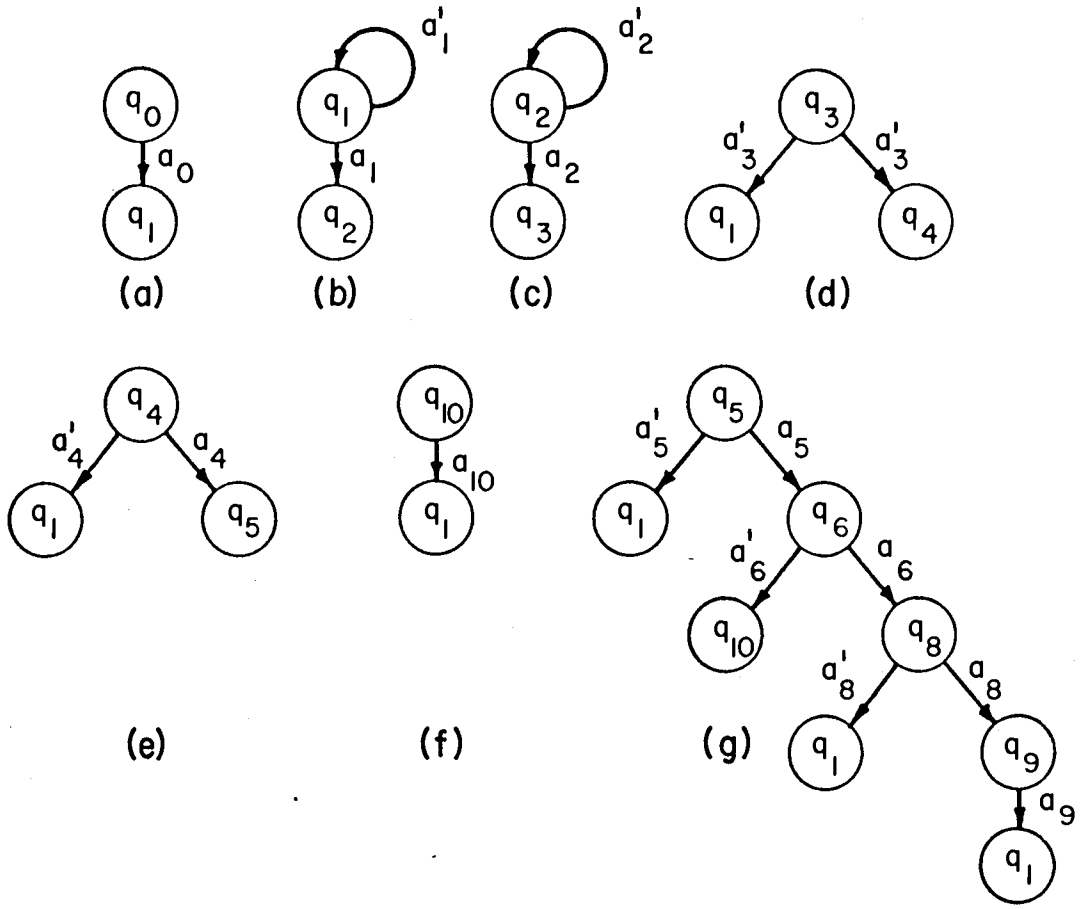
q0
a0
q1
(a)
a'1
q1
a1
q2
(b)
a'2
q2
a2
q3
(c)
q3
a'3
a'3
q1
q4
(d)
q4
a'4
a4
q1
q5
(e)
q10
a10
q1
(f)
q5
a'5
a5
q1
q6
a'6
a6
q10
q8
a'8
a8
q1
q9
a9
q1
(g)

FIG. 8

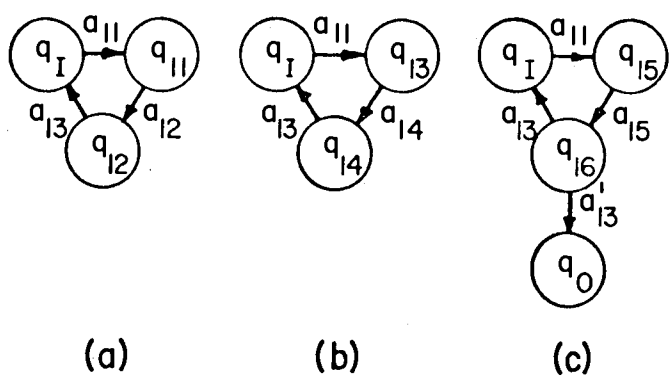
qI
a11
q11
a13
a12
q12
(a)
qI
a11
q13
a13
a14
q14
(b)
qI
a11
q15
a13
a15
q16
a'13
q0
(c)

FIG. 9

FIG. 10

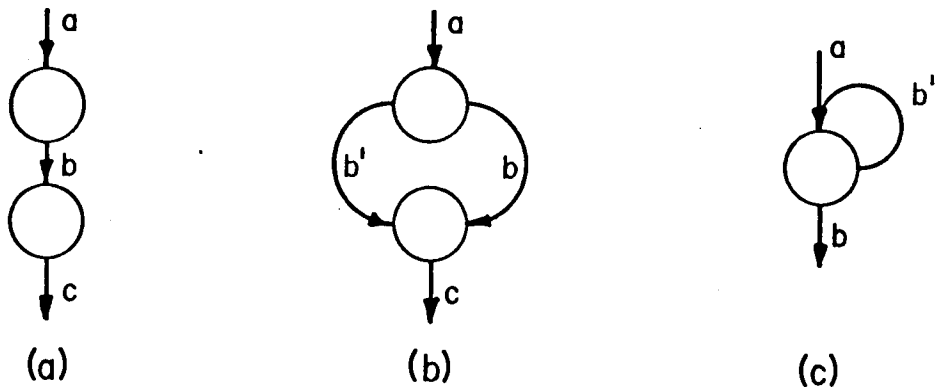

FIG. 11

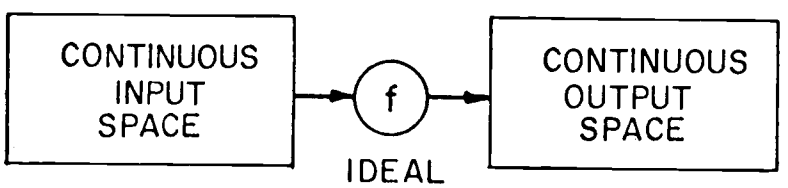

FIG. 12

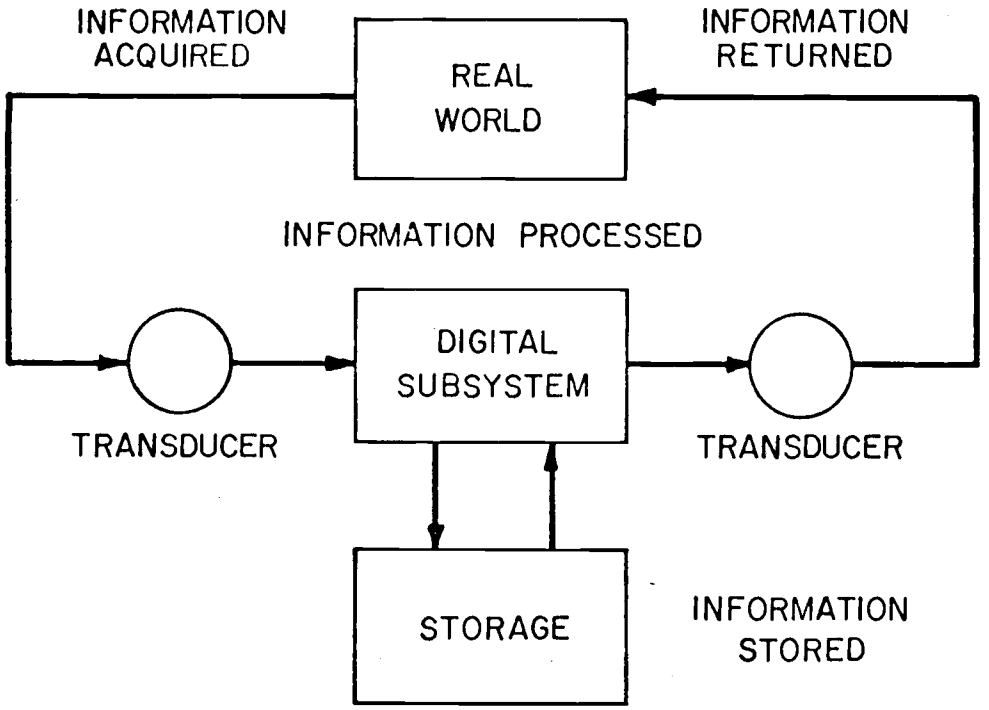
INFORMATION ACQUIRED
REAL WORLD
INFORMATION RETURNED
INFORMATION PROCESSED
DIGITAL SUBSYSTEM
TRANSDUCER
TRANSDUCER
STORAGE
INFORMATION STORED

FIG. 13

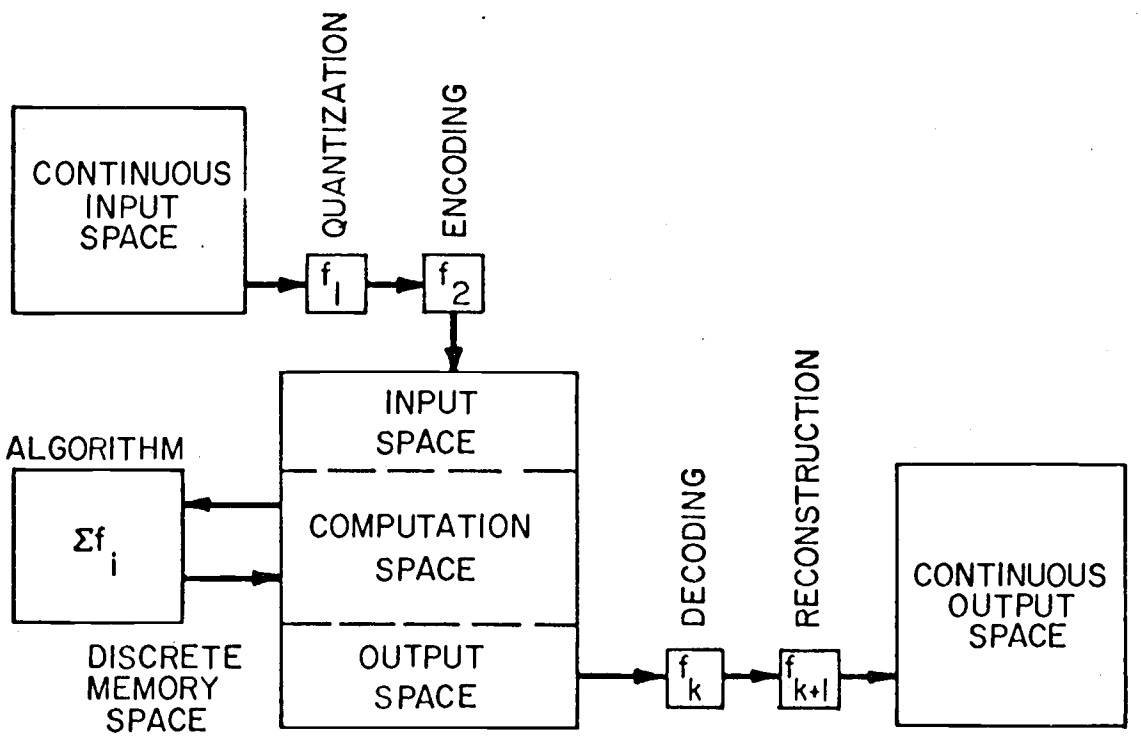
CONTINUOUS INPUT SPACE
QUANTIZATION
ENCODING
f_1
f_2
INPUT SPACE
ALGORITHM
Σf_i
COMPUTATION SPACE
DISCRETE MEMORY SPACE
OUTPUT SPACE
DECODING
RECONSTRUCTION
f_k
f_{k+1}
CONTINUOUS OUTPUT SPACE

FIG. 14

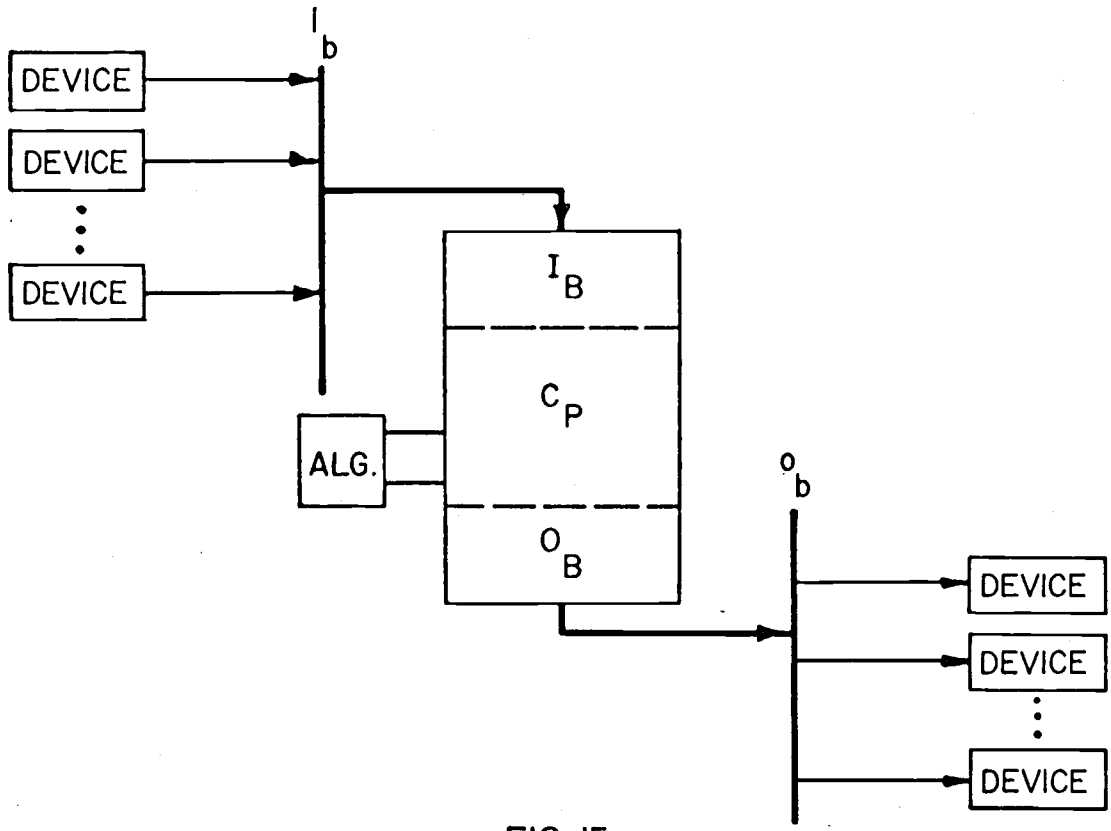
I b
DEVICE
DEVICE
DEVICE
I B
C P
ALG.
O B
o b
DEVICE
DEVICE
DEVICE

FIG. 15

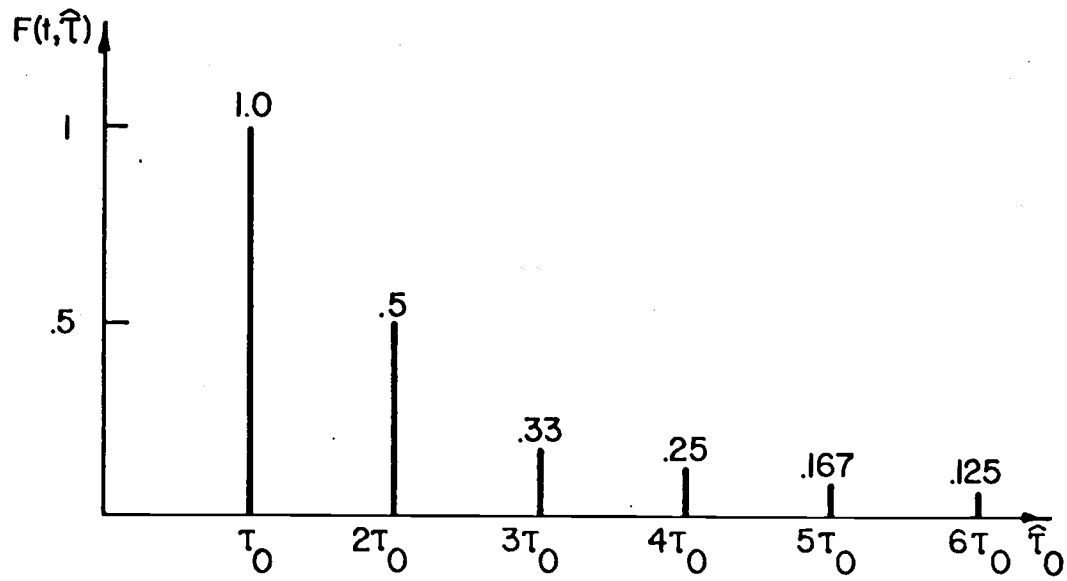
F(t,τ̂)
1
.5
1.0
.5
.33
.25
.167
.125
τ0
2τ0
3τ0
4τ0
5τ0
6τ0
τ̂0

FIG. 16

OP	Type	Successor	
		Fall Through	Branch
a_1	unconditional	a_2	
a_2	unconditional	a_3	
a_3	conditional	a_4	exit
a_4	unconditional	a_5	
a_5	unconditional	a_6	
a_6	conditional	a_7	a_9
a_7	unconditional	a_8	
a_8	unconditional	a_2	
a_9	unconditional	a_{10}	
a_{10}	unconditional	a_8	

Table 1. Operation Table

OP	Class	D_{a_i}	R_{a_i}	Condition, Action	
				γ=1	γ=0
a_1	Γ	$m_0, m_2 \ldots m_9$	m_1, m_0	$[m_0]=0$, ___	$[m_0]\neq 0, m[m_0]\rightarrow m_1$ $0\rightarrow m_0$
a_2	Γ			unc, ___	___
a_3	E		μ	___, $\downarrow\mu$	___
a_4	A	i_b	m_0	___, $[i_b]\rightarrow m_0$	___
a_5	Z	μ		___, $\uparrow\mu$	___
a_6	E		μ	___, $\downarrow\mu$	___
a_7	B	m_1	o_b	___, $m_1\rightarrow o_b$	___
a_8	Z	μ		___, $\uparrow\mu$	___

Table 2. Interpretation of Operations (Ex. 1)

OP	Class	D_{a_i}	R_{a_i}	Action γ=1	γ=0
a_0	Δ			init. memory	
a_1	Γ			scan data avail.	$\overline{\text{not avail.}}$
a_2	Γ			scan complete	not complete
a_3	Γ			target in range	not
a_4	Γ			enemy	friend
a_5	Γ			enemy in range	not
a_6	Γ			missile fired	not
a_8	Γ			update traj.	no
a_9	Δ			update	___
a_{10}	Δ			launch	___
a_{11}	E			interrupt, $\downarrow\mu$	___
a_{12}	A			input scan data	___
a_{13}	Z			return, $\uparrow\mu$	$\overline{\text{reset return}}$
a_{14}	B			output m. control	___
a_{15}	A			input m. status	___

Table 3. Interpretation of Operations (Ex. 2)

Process	Init. State	Regular Expression Description	Terminal St.	Priority
P_0	q_0	(a_0)	q_1	π_0
P_1	q_1	$(a_1'^* a_1)$	q_2	π_1
P_2	q_2	$(a_2'^* a_2)$	q_3	π_2
P_3	q_3	(a_3+a_3')	q_1, q_4	π_3
P_4	q_4	a_4+a_4'	q_1, q_5	π_4
P_5	q_5	$(a_5'+a_5(a_6'+a_6(a_8'+a_8a_9))$	q_1, q_{10}	π_5
P_6	q_{10}	(a_{10})	q_1	π_6
P_{I1}	q_I	$(a_{11}a_{12}a_{13})$	q_I	π_7
P_{I2}	q_I	$(a_{11}a_{14}a_{13})$	q_I	π_8
P_{I3}	q_I	$a_{11}a_{15}(a_{13}+a_{13}')$	q_I	π_9

Table 4. Process Description (Ex. 2)