Massachusetts Institute of Technology
Laboratory for Computer Science

Computation Structures Group Memo 170

Computation Structures Group Progress Report 1976 - 1977

This research was supported in part by the National Science Foundation under grant DCR75-04060 and in part by the Advanced Research Projects Agency of the Department of Defense under contract N00014-75-C-0661.

October 1978

COMPUTATION STRUCTURES GROUP

Academic Staff

J. B. Dennis, Group Leader

Research Staff

D. P. Misunas

Graduate Students

W. B. Ackerman
K. Amikura
S. A. Borkin
G. A. Boughton
J. D. Brock
R. E. Bryant
D. J. Ellis
D. L. Isaman
P. R. Kosinski
C. K. C. Leung
G. S. Miranker
L. B. Montz
K. S. Weng

Undergraduate Students

T. B. Freeman
S. J. Grossman
R. G. Jacobsen
T. L. Kuehn
D. R. Nadler

Support Staff

A. L. Rubin

COMPUTATION STRUCTURES

A. INTRODUCTION

Research in the past year has concentrated on various aspects of data flow programming languages and computer architecture. The current efforts are directed toward the examination of uses of data flow processors, the resolution of architectural issues such as the implementation of procedures and data structures, the development of simulation facilities, the study of hardware and software implementation issues, and the investigation of formal semantic models for data flow languages and systems.

B. APPLICATION OF DATA FLOW PROCESSORS

In two studies carried out this year, different applications of data flow processors have been described and their potential performance evaluated. The two areas of application are representative of many computational problems, and the results of the studies have shown great potential for use of data flow processors.

1. Signal Processing

As a demonstration of a signal processing application for which utilization of a data flow processor appears quite feasible, the fast Fourier transform (FFT) has been expressed in data flow form and its potential performance on a data flow processor evaluated [1].

The discrete Fourier transform of a sequence of $N = 2^n$ input samples $x_0, \ldots, x_{N-1}$ is the sequence of values $f_0, \ldots, f_{N-1}$ where

$$f_k = \sum_{i=0}^{N-1} x_i W^{ik}$$

and:

$$W = e^{-j(2\pi/N)}$$

The direct computation of these values involves the accumulation of N^2 product terms; the Fast Fourier Transform (FFT) is based on the observation that the transform on 2^p data samples can be simply expressed in terms of two transformations on 2^{p-1} samples. Continuing recursively, one discovers that the transform on 2^n points can be expressed in terms of n 2^{n-1} transformations of two points each. Figure 1 shows the flow of values in one arrangement of the FFT computation for eight data points (n=3). This arrangement, in which the computation consists of n stages (the columns of the figure) having identical form, is known as the time decimated constant geometry FFT [2]. Each stage of the computation is composed of N/2 units of similar form, known as "butterflies" which compute two-point transforms.

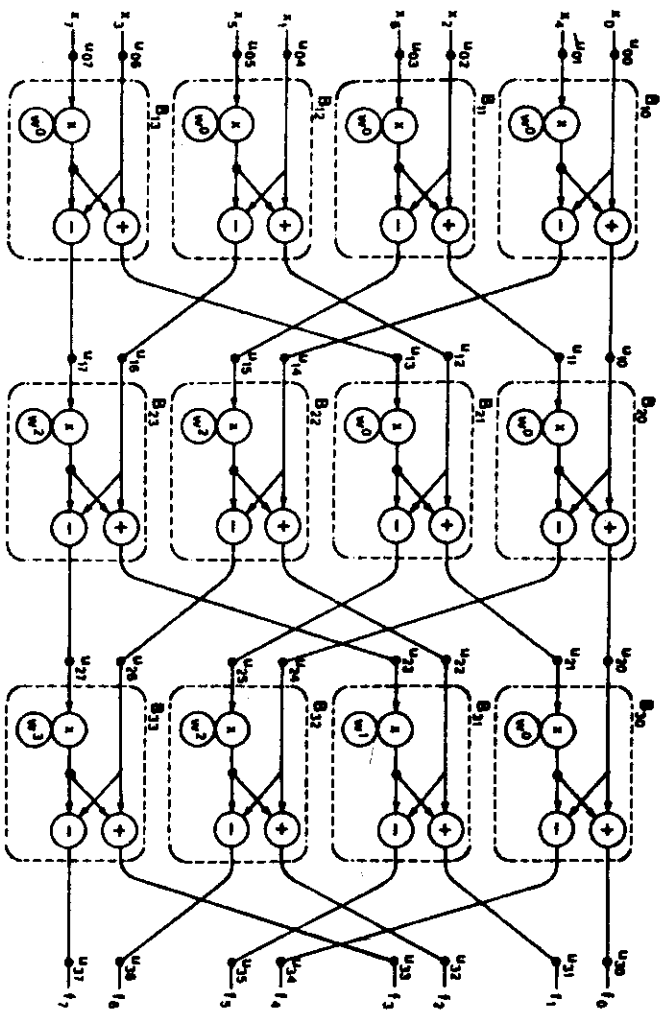

Figure 1. The eight-point, constant geometry, time decimated FFT.

The general form of this FFT algorithm may be described as follows: let $u_{p,k}$ be the k^{th} component of the vector of values computed by the p^{th} stage of the computation. Then $B_{p,q}$, the q^{th} butterfly of stage p computes

$$u_{p,q} = u_{p-1,2q} + u_{p-1,2q+1}W^{e_{p,q}}$$

$$u_{p,q+2^{n-1}} = u_{p-1,2q} - u_{p-1,2q+1}W^{e_{p,q}}$$

where the exponent $e_{p,q}$ of each phase factor $w_{p,q} = W^{e_{p,q}}$ is given by

$$e_{p,q} = 2^{n-p}\ quo(1,2^{n-p})$$

and

$$0 \le q < 2^{n-1}$$

$$0 < p \le n$$

The symbol quo denotes the function quo(m,n) which yields the integer quotient of m divided by n. The input values for stage one are related to the data samples by

$$u_{0,k} = x_i, \quad \text{where } i = rev(k)$$

in which rev is the operation on integers such that the n-bit binary representation of i is the reverse of the n-bit representation of k. The output values are

$$f_k = u_{n,k}, \quad 0 \le k < 2^n$$

We wish to take maximum advantage of parallelism in representing the FFT as a data flow program, but, to conserve space within the machine, we do not want to use a larger program than necessary to exploit concurrency. Since each stage of the computation uses values computed by the preceding stage, it is appropriate to write the program as an n-cycle iteration in which the body consists of the N/2 butterflies comprising one stage of computation written out explicitly. The form of the corresponding data flow program is shown in Figure 2 for the eight-point case.

The constant geometry of the computation illustrated in Figure 1 over all stages makes it possible to use a fixed routing of values from the outputs of the butterflies to their inputs, where they become operands for the next cycle. However, generating the phase factors for each butterfly presents a problem. The usual technique is to use a table lookup in a table of powers of W, but our present data flow language includes no suitable mechanism for this function. Instead, the factor $w_{p,q}$ used for butterfly q in stage p may be computed from the factor $w_{p-1,q}$ used for the previous stage by a simple rule derived as follows: the exponents of W for $w_{p,q}$ and $w_{p-1,q}$ are:

$$e_{p,q} = 2^{n-p}\ quo(q,2^{n-p})$$

$$e_{p-1,q} = 2^{n-p+1}quo(q,2^{n-p+1})$$

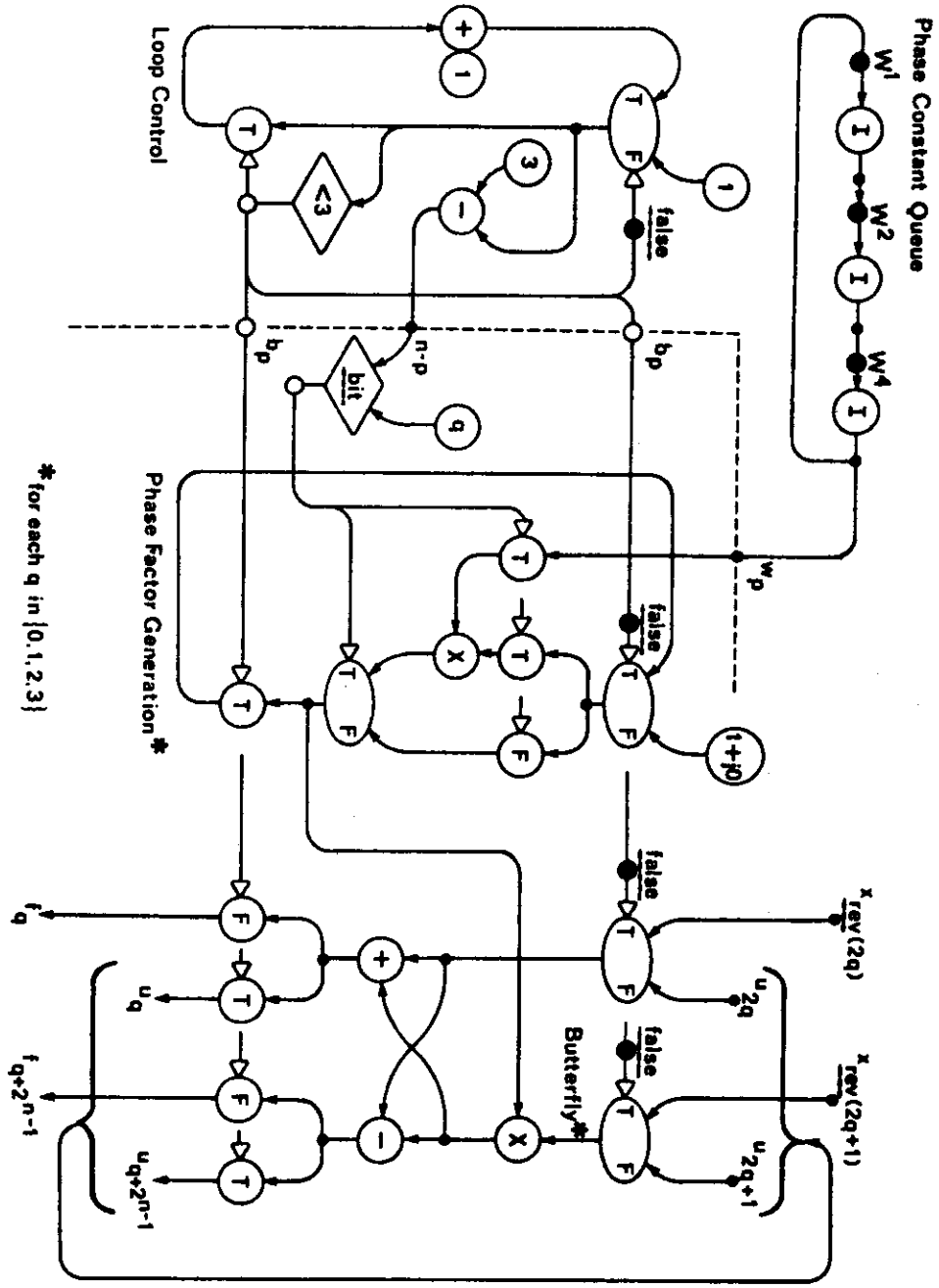

Figure 2. Iterative data flow program for the eight-point FFT.

Then

$$e_{p,q} = e_{p-1,q} + (e_{p,q} - e_{p-1,q})$$
$$= e_{p-1,q} + 2^{n-p}(quo(q,2^{n-p}) - 2\ quo(q,2^{n-p+1}))$$

where the term factor $(quo(q,2^{n-p}) - 2\ quo(q,2^{n-p+1}))$ is denoted $T_{p,q}$. Careful study of the factor $T_{p,q}$ reveals that:

$$T_{p,q} = \begin{cases} 0 & \text{if } rem(q,2^{n-p}) \text{ is even} \\ 1 & \text{if } rem(q,2^{n-p}) \text{ is odd} \end{cases}$$

Thus $T_{p,q}$ is the $(n - p)^{th}$ bit in the binary representation of q. Let bit(r, q) be a primitive function that yields the r^{th} bit of q. Then we have

$$w_{p,q} = w_{p-1,q} * \begin{cases} \text{if bit}(n-p, q) = 1 \\ \text{then } W^{2^{n-p}}, \text{ else } 1 \end{cases}$$

The initial value of the phase factor for the q^{th} butterfly is

$$w_{1,q} = W^{e_{1,q}}, \text{ where } e_{1,q} = 2^{n-1}\ quo(q, 2^{n-1})$$
$$= W^0 = (1 + j0)$$

The computation of the phase factors $w_{p,q}$ is performed by the sections of Figure 2 labelled "Phase Factor Generation" and "Phase Constant Queue."

Signal values are delivered to the program as a continuous stream through a single input operator, and must be distributed among the 2^n input links of the FFT program. This may be done by means of a binary tree of data flow program fragments, which we may call fan-out alternators, connected as in Figure 3. A similar binary tree of fan-in alternators (Figure 4) can be used to form the transform values $f_0, \ldots, f_{N-1}$ into a stream.

To analyze the performance of the FFT computation on a data flow processor, we represent the cyclic execution of the FFT program by a special kind of Petri net known as a marked graph [3]. Each node of the marked graph of Figure 5 corresponds to one machine instruction participating in the cyclic computation or to a source of input values from the Loop Control Section of the program. Each directed arc represents a data path between instructions of the program. The arrowheads of the arcs indicate the type of the packets -- data, boolean, or signal -- that flow over the corresponding path. Tokens are placed on arcs of the marked graph to represent a live and safe initial configuration of the data flow program. Acknowledge values, which are required to maintain a safe configuration, are denoted by ϵ and data tokens carrying unknown values indicated by ( ).

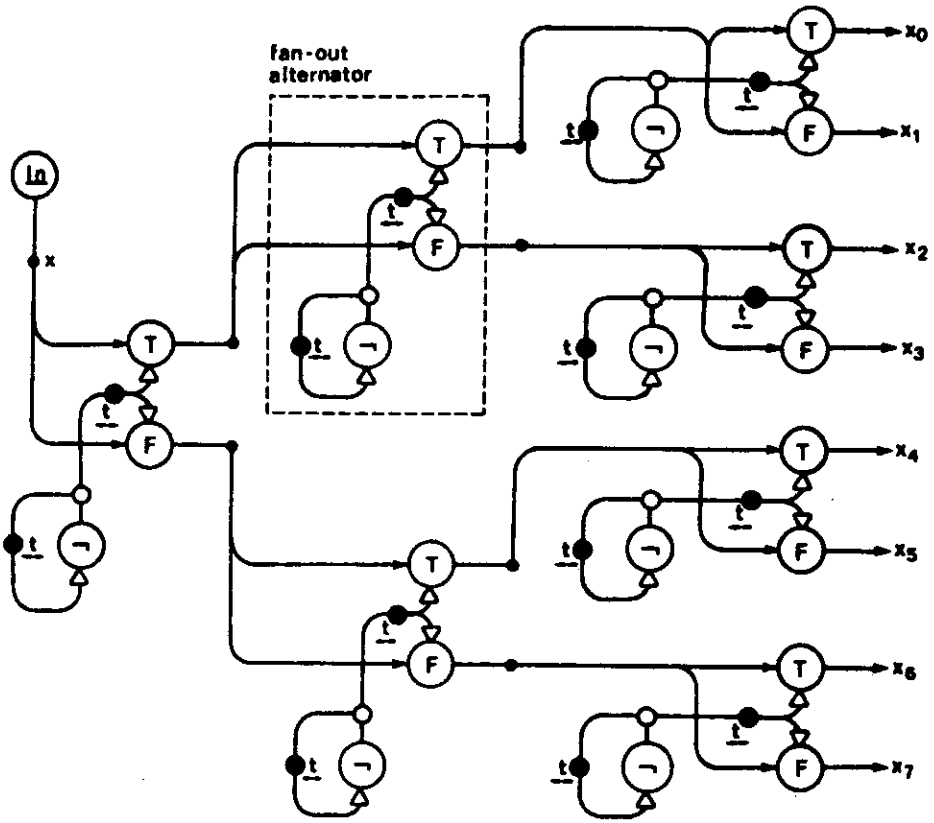

Figure 3. Tree of fan-out alternators for sample distribution.

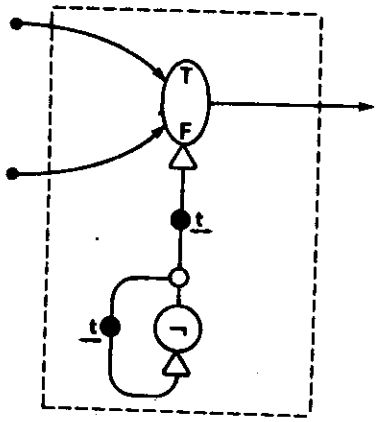

Figure 4. Fan-in alternator.

Each arc of the marked graph has an associated "propagation delay" which is the time interval from the moment a token is placed on the arc by its origin node to the moment presence of the token is observed by the destination node. For data arcs, this time consists of the time required for an operation packet to be transferred from the Memory to a Processing Unit plus the time for the data packet resulting from instruction execution to pass from the Processing Unit to the specified successor instructions in the Memory plus the time for processing an operation packet by a Processing Unit. The propagation delay for Boolean and signal arcs is derived in a similar manner.

Now the question of computation rate for the FFT program becomes a question of the minimum period for the cyclic behavior of a marked graph when each arc has a known propagation time. This problem was previously solved by Karp and Miller [4]: the minimum period is determined by the directed cycle in the marked graph having the largest value of total delay divided by the number of tokens on the cycle. Utilizing packet propagation delays corresponding to use of a technology such as Schottky TTL in the routing networks, we find there are two critical cycles in the program -- one involving nodes Gen-2, Gen-4 and Gen-7, and the other involving nodes Gen-5, Gen-7 and But-2. Each of these cycles has one token and a total delay of approximately 80 microseconds. Thus, data flow processors can be constructed with execution times for the FFT computation as little as (log N)*(80 microseconds) for a feasible technology of interconnection network. For further details of the performance analysis see [1].

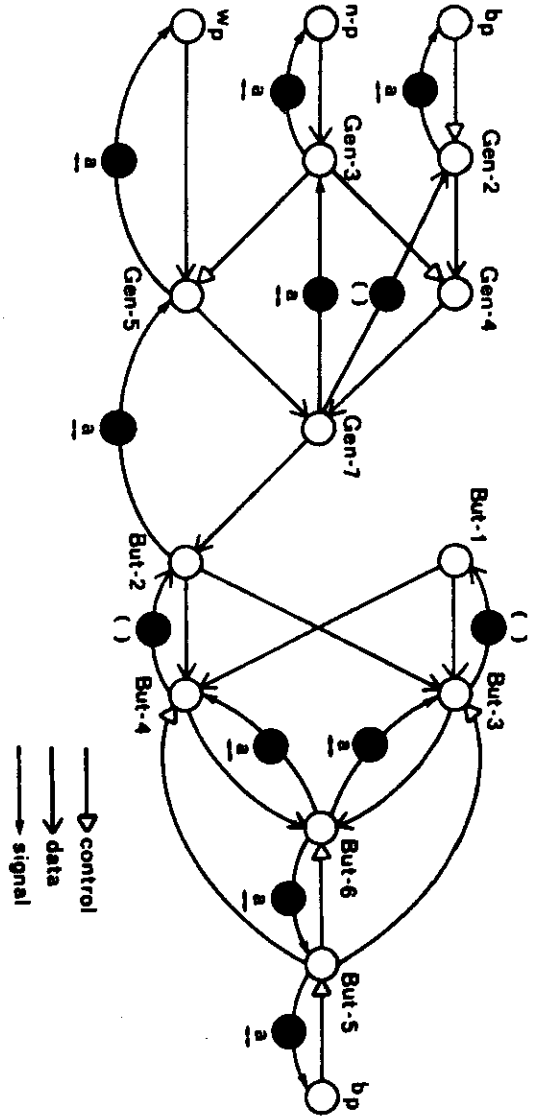

Figure 5. Marked graph description of the FFT computation.

2. Weather Simulation

In an effort to demonstrate application of a data flow processor to a complete computation, a data flow language which includes structure operations on arrays has been used to describe implementation of a global general circulation model (GCM) for numerical weather forecasting [5]. The nature of this computation permits exploitation of much parallelism in the data flow representation, allowing balanced utilization of the units of a data flow processor. Preliminary performance results indicate that a hundred-fold performance increase over the current IBM 360/95 execution time is feasible.

The general circulation model used in this study is the GISS fourth order model developed by Kalnay-Rivas, Bayliss and Storch [6] in which the atmospheric state is represented by the surface pressure, the wind field, the temperature, and the water vapor mixing ratio. These state variables are governed by a set of partial differential equations in the spherical coordinate system formed by latitude, longitude, and normalized atmospheric pressure. In this fourth order model, the computation is carried out on a three-dimensional grid that partitions the atmosphere vertically into K levels and horizontally into M intervals of longitude and N intervals of latitude.

The main computation is the evaluation of the time derivatives of the state variables from the current atmospheric state, using the physical laws that govern the atmosphere. In addition to the main computation, there are several other incidental computations which must be performed, such as polar computation, filtering, and stability computation.

The data flow program for the simulation model is organized so the parallelism of the GCM computation is exposed in two major ways: first, in the main computation, evaluation of the time derivative is carried out concurrently for all K atmospheric levels. This is accomplished by using K copies of the data flow program appropriate for a single grid point. Second, the main computation is coded so grid points are processed in a pipeline fashion by scanning along the latitude lines.

Analysis of the complete data flow program reveals that the machine level program will consist of about 13,000 instructions. If the data flow version is to have a hundred-fold performance increase over the IBM 360/95 implementation, the processor/memory interconnection networks must be able to perform packet switching at 175 MHz, and the instruction execution delay should be no more than 20 microseconds. These speeds are readily achievable for the processor structures under discussion, implemented in a conventional technology.

Several problems related to the structure of the data flow machine and the language have been revealed by this study. The pipeline organization of the program necessitates special instructions in the data flow program for efficient execution of data structure operations. In addition, a higher level language which can be efficiently translated into the machine level representation is necessary for the expression and understanding of the computation. Such a language may be based on an extension of the language previously studied by Weng [7].

C. DATA FLOW COMPUTER ARCHITECTURE

In prior years, a number of data flow computer architectures have been proposed. Such architectures have ranged in complexity from simple processors designed to execute programs with no conditional, data structure, or procedure capabilities to complex machines for the implementation of programming languages on the order of Algol. This year, we concentrated our architectural studies on two important issues: the implementation of data structures and procedures.

1. Data Structures

The structure handling facility being developed by Bill Ackerman is designed to support arrays and records of the type that occur in conventional programming languages. Structures are implemented as binary trees; that is, as acyclic directed graphs in which each node is either a leaf (elementary value) or has two immediate subordinates. In the latter case, the arcs to the subordinates are labelled in a manner which allows the directed path from any node to any descendant to be specified by a compound selector.

The structure handling facility incorporates two structure operations: SELECT and APPEND. The SELECT operation requires as arguments a structure and a bit string, returning the substructure reached by following the directed path indicated by the bit string. The value returned may be an elementary value or a structure. The APPEND operation requires a structure, a bit string selector, and a value. It returns a structure equivalent to the input structure except that the input value is located at the position designated by the selector, replacing whatever was previously located there. The appended value may be an elementary value or a structure.

The appearance of the elementary constant NIL in a structure denotes the absence of any data. In addition, NIL is utilized to denote the empty structure, an arbitrary structure may be created by APPENDing values to NIL.

Data structures in data flow are handled in a purely applicative fashion. The value of an existing structure never changes as the result of any operation performed elsewhere in the program. The change effected by an APPEND operation appears only to the instructions which receive the structure from the APPEND operator over some directed path. If the original structure was shared with other parts of the program, the shared structure is not changed. To achieve this result, the APPEND operation copies any shared part of a structure before modifying it.

A structure controller has been designed which efficiently implements creation, transformation, retrieval, and deletion operations on data structures, using a packet memory for the storage of the structures. The packet memory system which contains the data structures has the property that it can be expanded both laterally and vertically. That is, it can be realized as separate smaller units, each handling a subset of the total address space. Furthermore, these separate units can be realized as separate units in a hierarchy, with the higher level units containing only the most active data. These lateral and vertical expansions are the data flow equivalent to the common techniques of interleaving and use of a cache, respectively.

2. Procedures

In a recently completed master's thesis [8], Glen Miranker has completed the work described in a previous report [9]. The thesis investigates in detail an implementation scheme for procedures on a data flow processor. The language level semantics of several alternative implementations are investigated and presented. The principal scheme exhibits a high degree of parallelism, yet the amount of state information required to be maintained by the processor is bounded and small.

Central to the implementation scheme is:

a. Creation of a virtual cell name space using a hierarchical associative store;

b. Creation and separation of different procedure instances through runtime renaming of the cells that store the encoded procedure instance; and

c. Selective copying of the parts of procedures as they become active.

It is shown that implementation of the general class of data flow schemas with procedures would give rise to nondeterminate behavior in a data flow processor. However, a large syntactic subclass of such schemas is presented and proven to be determinate.

D. SIMULATION OF PACKET COMMUNICATION SYSTEMS

We have been studying methods of simulating systems with packet communication architecture; that is, systems of independent units which communicate through the transmission of information packets, of which our data flow processors are prime examples. This year we made advances in the specification of an architecture description language for use in description of a simulated system, the development of coordination methods for the parallel units being simulated, and the development of a software simulator.

1. Architecture Description Language

One of the major accomplishments this year is the completion of the preliminary design of a formal language for the specification of packet communication systems (PCSs). This architecture description language (ADL) is intended to serve as a medium for system documentation and human communication, as a formalism for design verification, and as the language interface to a design automation and simulation facility. The ADL complements existing computer hardware description languages in that it is designed for architecture description at the algorithmic behavior/system structure level, not for straightforward translation into existing component technology. As it stands, ADL can be used as a design tool to support design methodologies for PCSs and for PCS specification. The novel features of ADL include the adoption of data flow as a basis for its operational semantics, state variables for implementing functions on data streams, and monitors for sharing data objects.

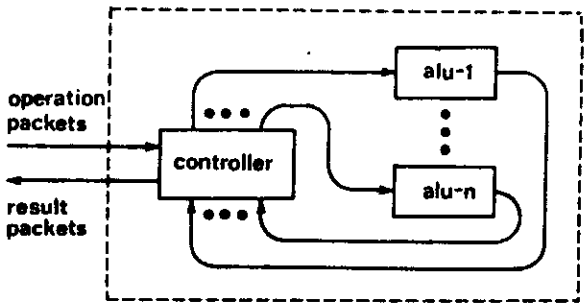

Figure 6. An example of an architectural unit:
The arithmetic logic processor.

Figure 6 illustrates an architectural unit consisting of a number of identical arithmetic logic units managed by a controller. Requests for processing arrive at the unit in operation packets, each operation packet containing a specification of the operation to be performed, the necessary operands, and the destination address(es) for the result(s). This architectural unit provides as output result packets, each holding one copy of a result value produced through execution of an operation specified in an operation packet and each tagged by a destination address. The controller receives operation packets when the arithmetic logic units are available and dispatches each request to a free unit. The controller also appropriately tags the results computed by the arithmetic logic units and transmits them in result packets.

A description of the structure of this architectural unit, the arithmetic logic processor, or ALP, is formalized in ADL as a module structure type. The structural type definition for ALP is shown in Figure 7. This type definition contains declarations of input and output ports, the type of packets received or transmitted at each port, a list of submodules, and a list of interconnections joining pairs of submodule ports. In this example, ALP is defined as an interconnection of its submodules, demonstrating the hierarchical module decomposition of ADL.

The behavior of a module is described in the ADL by composing expressions. The semantics of expression evaluation is based on the principle of data flow: each evaluation of an expression is initiated as soon as a new set of operands is available and the result of the previous evaluation is no longer needed. Many expressions can thus be viewed as functional modules with well-defined input and output interface behavior.

The concept of a module state which can be updated is incorporated in ADL behavioral expressions to allow definition of functions on data streams. Also, a simplified version of Hoare's monitors is utilized to introduce nondeterminism via shared state variables. The incorporation of these features is carefully structured so that expression evaluation is still governed by the flow of data.

```
type ALP = module
            inlet opn-in: operation-pkt;
            outlet res-out: result-pkt;
            param n-of-alu: integer;

        submod
            k: controller (n-of-alu)
                inlet opn-in: operation-pkt, alu-in[1..n-of-alu]: alu-res;
                outlet res-out: result-pkt, alu-out[1..n-of-alu]: alu-opn;
            end k;

            foreach i:integer in [1..n-of-alu]
            {a[i]: alu
                inlet in: alu-opn;
                outlet out: alu-res;
                end;
            }

        connection
            opn-in -> k.opn-in;
            k.res-out -> res-out;
            foreach i: integer in [1..n-of-alu]
            { k.alu-out[i] -> alu[i].in;
                alu[i].out -> k.alu-in[i]
            }
end ALP;
```

Figure 7. Structural Type Definition for ALP.

Figures 8 and 9 illustrate the semantics of the monitor construct in ADL with a simple example. The architectural unit described in Figure 9 receives integer packets at two input ports I1 and I2 and maintains a state variable *sum* which is initialized to 0. The content of each packet received at I1 is added to *sum*, those from I2 are subtracted from *sum*. These modifications are performed using monitor procedures, each of which returns the current value of *sum*. The relevant fragment of the behavioral specification of this module is given in Figure 8.

The semantics of this behavior is given in terms of the data flow program fragment in Figure 9. To simplify the discussion we have used some macro-operators. The *scase* and *mcase* operators perform selection and switching operations, controlled by a name tag which is a monitor procedure name.

The data objects available at data links i and p in Figure 9 are first tagged by the name of the monitor procedure to be invoked. The objects are then *merged* to form a single request stream to the monitor. The *scase* operator C1 switches the input to the data flow program fragment implementing the corresponding monitor procedure. Each monitor procedure invocation may generate a set of return values and/or a set of new values for the state variables of the monitor. The output of each procedure

invocation and the new state variable generated have identical values in this example. The output is then routed to the appropriate link (*j* or *q*) and the new value of the state variable is routed to the state variable link (*sum*) via the *mcase* operator C2.

2. Asynchronous Simulation Techniques

In his master's thesis, Randal Bryant has explored methods of simulating highly parallel asynchronous computer systems. In particular, methods were developed which would exploit the modularity and parallelism in the systems to be simulated and yield a simulation which is itself highly modular and parallel. The simulation techniques which were developed allow execution of the simulation on any computer system which supports communicating processes, including a network of microprocessors.

Besides modeling the functional behavior of the system, a proper simulation must also model its time behavior. To avoid placing real-time constraints on the simulation processes, a time-independent algorithm for simulating the time behavior is required. Furthermore, to avoid the need for a high-speed central controller for the simulation, all the time simulation algorithms must be decentralized, requiring special control operations to prevent the simulation from deadlocking and to ensure its proper termination.

```
m: monitor
       state sum := 0;

       /* procedure to increment sum */
       add: procedure(n: integer) result(integer);
              x: integer = sum + n;
              update sum := x;
              return x;
              end add;

       /* procedure to decrement sum */
       sub: procedure(n: integer) result(integer);
              x: integer = sum - n;
              update sum := x;
              return x;
              end sub;
       end m;
              .
              .
              .
i: integer = I1;
j: integer = m.add(i);
      .
      .
      .
p: integer = I2;
q: integer = m.sub(p);
```

Figure 8. An Example of Monitor Declaration and Use.

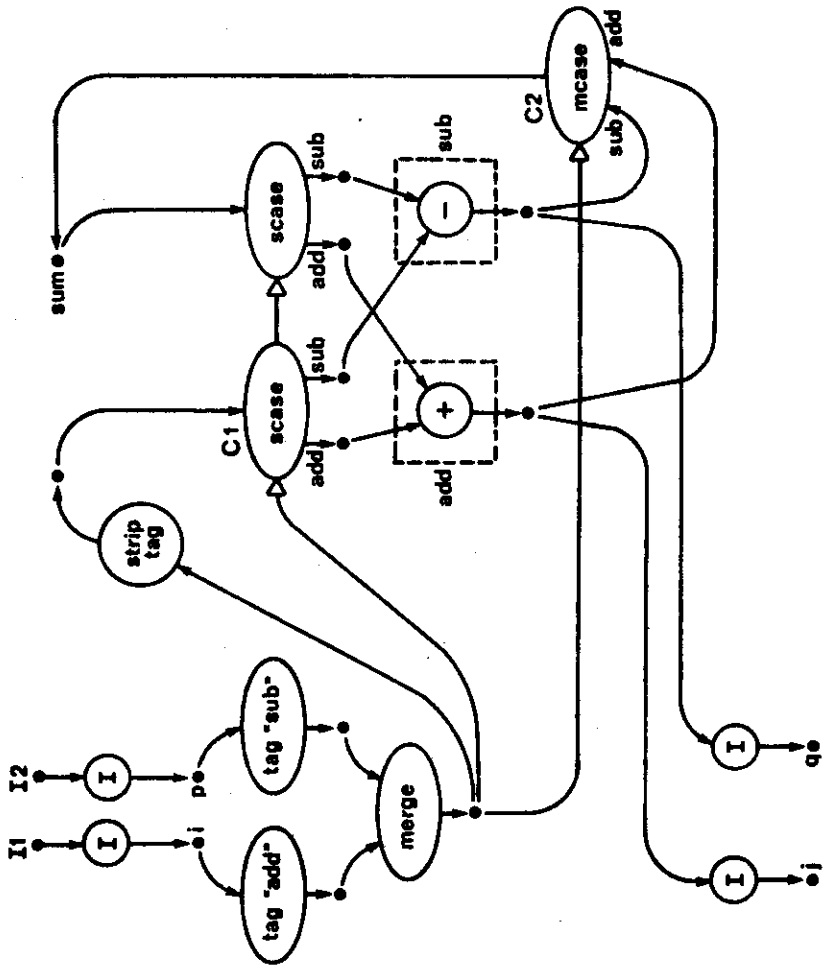

Figure 9. Data flow schema representation of a monitor.

The major contribution of this work involves the presentation of the necessary algorithms for controlling the simulation and proving its correctness. These algorithms involve a number of computations proceeding at different locations concurrently, where each computation has only a limited amount of information about the state of the rest of the system. As is typical of many parallel computations, it is difficult to prove correctness, yet proofs of correctness are almost imperative, considering the great potential forms of incorrect behavior. Fortunately, since a simulation need only model the behavior of some other system, one need only prove that at no point will the values produced by the simulation diverge from those produced by the simulated system, and that the simulation will not deadlock or fail to terminate. Thus, rather than facing the more general issue of proving the correctness of parallel computations, specialized techniques were developed for this particular problem.

3. Software Simulation

A software simulation facility for packet communication systems has been developed. The software simulator resides on the PDP-11/70 computer operated by the Domain Specific Systems Research Group of L.C.S. and consists of a compiler and a simulator. Descriptions of a proposed architecture in the ADL are translated by the compiler into code executed by the simulator. The simulator provides the necessary tools for the execution of simulation programs and the control and monitoring of the simulation process. The system is currently in the final debugging phase and should prove a useful tool over the next few years for the other work in progress.

E. IMPLEMENTATION STUDIES

In a preliminary look at the issues involved in implementing our concept of a data flow processor, we have been studying the design of the unique elements required for its construction. We have examined the structure of the interconnection networks that route packets between sections of the machine and the hardware design of a basic component of the machine: the instruction cell block. Also, we have studied the problem of translating data flow graphs to machine language representations, and the implications of fault tolerance for the design of packet communication systems such as our proposed data flow computer.

1. Routing Network Design

Andy Boughton has conducted extensive studies on the complexity and performance of routing network designs for use in packet communication systems. This work has centered around two basic types of networks and has examined the structure of such networks along with their associated complexity and delay.

Routing networks are packet communication systems and, as such, are constructed from modules and links. Each component module of a network has associated with it an independent controller and is connected via links to a fixed number of other modules. A packet may be transferred between two connected modules over their common link if all preceding packets transmitted over that link have been acknowledged. Thus, a routing network demonstrates the packet communication system characteristics of asynchronous concurrent operation and decentralized control.

The research on routing network design has examined two basic networks (concentration networks and connection networks) from which most routing networks can be constructed. This examination has been concerned with the complexity required to construct a routing network with a particular number of inputs and outputs, and with a particular level of performance. Complexity is measured in terms of the number of modules required by the network. Performance is composed of two components, network throughput and network depth. Throughput is the rate at which the network will accept packets, whereas depth corresponds to the average time a packet spends in the network.

A concentration network has more inputs than outputs. Each packet accepted by the network will eventually be placed on some output. A nonblocking concentration network is a concentration network which will accept all packets on its inputs at a given time unless during some period ending at that time it has accepted a number of packets greater than could possibly be placed on the outputs in that period. We have shown that no N input, fixed-fraction-of-N output, nonblocking concentration network can be constructed with less than O(N log N) modules. We have given the construction of such a network with depth O(log N) and with associated complexity within a constant factor of optimal.

A connection network has the same number of inputs as outputs. Each packet accepted by the network has a label and is eventually placed on the output of the network which corresponds to that label. A nonblocking connection network is a connection network which accepts all packets on its inputs at a given time unless during some period ending at that time it has accepted a number of packets labeled for a particular output greater than could possibly be placed on that output in that period. We have shown the construction of a class of N input connection networks for which the expected throughput in a typical application is close to that of a nonblocking connection network. The complexity of a network of this class ranges from $O(N^2)$ to $O(N (\log N)^2)$, with corresponding network depths ranging from O(log N) to $O((\log N)^2)$.

It is interesting to note the comparison between this connection network and sorting networks, which have been well-studied in the literature (a sorting network sorts a group of packets placed on its inputs based on their labels as opposed to a network such as the connection network which places all packets with a particular label on a particular output). The best known construction for a sorting network has, for an N input network, complexity of $O(N (\log N)^2)$ and a depth of $O((\log N)^2)$.

We have studied two specific varieties of routing network with application to the design of data flow processors: arbitration networks and distribution networks. An arbitration network is a routing network with a larger number of inputs than outputs, whereas a distribution network is a routing network with fewer inputs than outputs. We have developed constructions for arbitration and distribution networks which are simple compositions of concentration and connection networks. These networks have expected throughputs in typical applications close to that of the nonblocking networks.

2. Hardware Design

An implementation scheme for an elementary data flow processor is being developed by K. Amikura. To examine the methods and technologies of such an implementation, the study has concentrated on the most complex part of the processor, the instruction cell block. The cell block under study is the basic building block of the memory of a data flow computer and is composed of 16 distinct instruction cells, each of which holds one instruction of a data flow program in execution on the processor.

An instruction cell performs a number of complex operations, including the reception of packets, the loading of operands, various managerial operations to update the status of the cell, testing of enabling conditions, and the transmission of its contents to a processing unit. In addition, each cell must contain a mechanism for initial loading of the program, a facility to dump its contents for debugging purposes, and an error mechanism for handling received packets that do not have the required format.

The behavior of a cell block was first formally described in the architecture description language. This description was then utilized to generate data flow interconnection graphs and a Petri net control graph of the system. From these graphs, the design was generated by a top-down decomposition of the specifications, utilizing conventional components and asynchronous communication disciplines for both external and internal communication.

3. Program Translation

An examination of the problem of translating data flow graphs to machine language representations for a data flow processor is being conducted by Lynn Montz to provide a second phase to the previous work on textual data flow languages that translate into data flow schemas. The naive translation of a data flow graph into a machine language representation would lose the enabling constraints imposed by data flow firing rules, so that the resulting cell representation would no longer be determinate and could possibly deadlock. A solution of this problem appears possible using the theory of Petri nets, in which the properties of safety and liveness correspond respectively to the concepts of determinacy and deadlock in data flow graphs. More specifically, by representing data flow graphs as free choice Petri nets, we hope to apply theorems dealing with safety and liveness for such Petri nets. Although examples of a desired solution have been worked out, the algorithm for producing such results is currently under development.

4. Fault Tolerance

One of the primary reasons for the development of the architecture description language is to facilitate the design and specification of fault-tolerant packet communication systems. The major challenge in this work lies in the asynchronous nature of operations within a packet communication system. Classical fault-tolerance schemes such as triple-modular redundancy schemes rely on system clock pulses as reference points for sampling input to establish agreement or disagreement. In a packet communication system, fault-tolerant communication protocols must be established to perform any desired comparison. The development and description of such protocols is currently under study.

F. SEMANTIC ISSUES

1. Data Flow Programming Languages

Data flow programming languages (DFPLs) are especially amenable to mathematization of their semantics in the style of Scott and Strachey [10]. That is, a data flow operator can readily be viewed as a function from input data sequences to output data sequences. However, coping with non-determinate programs is a more challenging problem, as the functions must be from sets of sequences to sets of sequences and finding a partial order in which the functions are continuous is difficult. This problem is being extensively studied by Paul Kosinski in his doctoral research.

Since determinate operators are adequately characterized as functions from sequences to sequences, the well known partial order on sequences, namely the "prefix" relation, would be adequate for their semantics [11]. If the infinite sequences are included, the poset characterized by the prefix relation is chain complete. All the determinate operators of DFPL, if viewed as functions from sequences to sequences, are both isotone (monotone) and continuous in this poset.

Unfortunately, non-determinate operators are best viewed as functions from sets of sequences to sets of sequences. Furthermore, determinate operators must be treated the same so that the domains and codomains of all operators are compatible. Imposing a partial order on sets of sequences is a frustrating task. For example, the Egli-Milner ordering [12] is really only a quasi-order, which means that the fixpoint equations can only be solved to yield a congruence class of sets of sequences. For DFPL at least, such congruence classes have the counter-intuitive property that one class contains two sets which have no elements in common!

It is possible to obtain a straightforward partial order by considering sets of tagged sequences of data. Each data sequence in the set has associated with it zero or more tags, each of which identifies the sequence of arbitrary decisions made by a non-determinate operator which contributed to the existence of that data sequence. These tagged sets bear some resemblance to the multi-sets of Lehmann's power domains [13]. Two sets are compared by matching up the tags on each element of the first set with the corresponding tags on the elements of the second set. Only then are the data seqences compared by the prefix ordering. This relation may be shown to be a true partial ordering of sets of tagged sequences, and the resulting poset is chain complete if infinite sequences and sets are admitted.

Any determinate operator, whose functional behavior on simple data sequences is known, may be easily extended to a function on sets of tagged sequences. If the operator has N inputs, apply the sequence function for the operator to each N-tuple of data sequences (in the Cartesian product of the tagged sets) for which all tags associated with those sequences are compatible, where compatible means that decision sequences from the same non-determinate operator are all equal. The output set's sequences are then tagged with the union of the tags of all the inputs in the corresponding N-tuple, assuring that the operator's function is not applied to input sequences which could never co-exist because they arose from different decision sequences of some non-determinate operator. The simple rules of only applying the function to each N-tuple of the Cartesian product of the input sets fails in this regard.

The only primitive non-determinate operator is the Arbiter which, viewed as a function from sequences to sets of sequences, produces the set of all possible ways of merging the input sequences such that each element is tagged by the unique name of the Arbiter (which just tells which Arbiter in the program it is) and the sequence of decisions made (i.e. which input sequence supplied the next element of the merged sequence). For example, if the input sequences "AB" and "CD" were merged by the Arbiter named "Z", the output set would be {Z0011:ABCD, Z0101:ACBD, Z0110:ACDB, Z1001:CABD, Z1010:CADB, Z1100:CDAB}.

Viewed as a function from sets of tagged sequences to sets of tagged sequences, the Arbiter is extended in a manner similar to that of any determinate operator. That is, the merging function given above is applied to each tag compatible N-tuple of the Cartesian product of input sets, with the exception that the output set is tagged with the union of the input tags with the Arbiter generated tag added.

Since the determinate operators are isotone and continuous in the poset of data sequences, they are isotone and continuous in the poset of sets of tagged data sequences. Similarly, the Arbiter is isotone and continuous in the poset of tagged sequences. Therefore, any recursive system of equations involving these operators has a unique minimal fixed-point in that poset. This means that any DFPL program, with or without iteration (cycles in the directed graph), but without recursion, corresponds to a well defined function from sets of tagged sequences to same, and all such functions are themselves isotone and continuous.

Furthermore, since the set of continuous functions from complete posets to same, is itself a complete poset, any system of recursive functional equations has a unique minimal fixed-point in the poset of continuous functions. This means that DFPL programs with recursive operators correspond to well defined functions from sets of tagged sequences to same. Hence, all DFPL programs correspond to well defined functions.

Data flow programming languages have cleaner mathematical semantics than ordinary programming languages. Because they are basically applicative in nature and local in effect, the functions act solely on the data without states, continuations or other complications. The tags associated with the data sequences do complicate matters of course, but this complexity is for the purpose of dealing with non-determinacy, which is not addressed by states, continuations, etc. Furthermore, the tags serve double duty. First, they allow the construction of a straightforward partial order. Second, they are necessary to the specification of how operators functionally transform input sets of sequences to output ones. Hence, they are less onerous than they might seem at first.

2. Packet Communication Systems

To better understand the design principles utilized in the development of systems with packet communication architecture, David Ellis has continued the research program reported last year [9] which is aimed at the development of theoretical models for precisely describing the structure and behavior of packet systems. The program has resulted in a mathematical characterization of the behavior of packet systems in terms of their internal structure. This characterization has been used to formally prove the correctness of several simple packet systems.

A packet system is an interconnection of independently functioning modules which interact only by passing discrete packets of information to each other over one-way data paths known as channels. The structure of a packet system is determined by its composition of modules and channels and always remains fixed for a particular system. Figure 10 depicts a packet system *DAS* composed of three modules *D*, *A* and *S*. There is one system input channel *X* and two system output channels *Y* and *Z*. The internal channel *U* connects module *D* to module *A*, and channel *V* connects module *D* to module *S*.

The behavior of a packet system is specified by two components: its structure and the behavior of its component modules. Thus, if we are given formal descriptions for the operation of the modules *D*, *A* and *S*, then we have all the information needed to characterize the behavior of the system *DAS*.

The behavior of a packet module is specified by a characteristic relation between its inputs and the semantically valid corresponding outputs. Since a module's response to a packet received on some input channel may in general depend on previously received inputs, module specifications must take into account sequences of packets rather than just individual packets. The characteristic relation for a module is thus taken over domains of streams, which are sequences of packets.

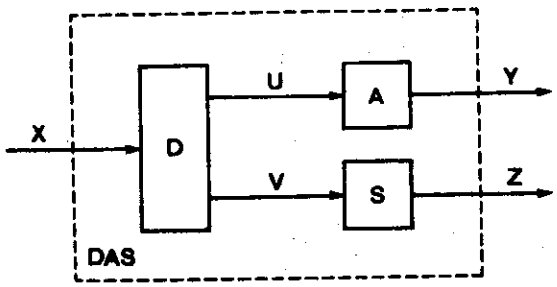

Figure 10. A sample packet system DAS.

As an example, imagine an adder module A which pairs up integer-valued packets in corresponding positions in its input streams x and r, adds the pairs and sends the sums out as a stream on s. If one input stream is longer than the other, the extra packets absorbed from the longer input stream are not reflected in the output response. The behavior of A is specified by a characteristic relation $\sim_A$ which relates input streams from X and input streams from R to output streams on S. The formal definition for $\sim_A$ is

$$((x,r),(s)) \in \sim_A \iff *s = \min(*x, *r) \text{ and } s[i] = x[i] + r[i]\ \forall i \leq *s,$$

where $*x$ denotes the length of the string x. This formal definition states that s is a valid response to the input streams x and r if and only if s has as many packets as the shorter of x and r and each element of s is the sum of the corresponding elements of x and r. As examples, we have:

$x = \langle 8,1,-6\rangle$, $r = \langle 3,-5,6\rangle$, $s = \langle 11,-4,0\rangle$;
$x = \langle 4,-9,0,-18\rangle$, $r = \langle\ \rangle$, $s = \langle\ \rangle$;
$x = \langle 1,3,5,\ldots,2i-1,\ldots\rangle$, $r = \langle 2,4,6,\ldots,2i,\ldots\rangle$, $s = \langle 3,7,11,\ldots,4i-1,\ldots\rangle$,

where the third example describes infinite input streams and an infinite response

The approach used in this research is based on an operational view of systems. We model the operation of a system by recording the progress of a computation in a series of internal system states. The system's response to particular input is characterized by a time-ordered progression of internal states, which we call an execution sequence. In general, there are a large number of possible execution sequences that correspond to a particular system response to some input. The proof of any system property must show that the property holds over all possible execution sequences which may be taken by the system.

The progress of a computation in a packet system is modeled by the succession of internal states in an execution sequence. An important property of execution sequences is that one can construct a system state which represents the computation running to completion. For such a state, known as a limit state, the output streams represent the system's ultimate response to its presented input. An execution sequence for which a limit state is well-defined is said to realize the system's particular output response to its presented input. The system's internal specifications are then the relation between input streams and the corresponding output streams realized by some execution sequence.

The correctness of a packet communication system is determined through the comparison of its behavior with a predetermined set of specifications. These specifications take the form of a relationship between the inputs of the system and the outputs generated in response to those inputs.

The behavior of a packet system is described in terms of its internal composition, which has two parts: a.) a structural description, and b.) the characteristic relation for each of its component modules.[1] This internal specification of a system will take the identical form as the external behavioral specification; that is, a relation between input streams and output streams.

To prove a packet system is correct, one must show that its internal specifications match a given external characteristic relation. There are two parts to such a proof:

a. demonstrating that the behavior realized by a given execution sequence satisfies the external specifications, and

b. showing that all instances of valid system behavior are realizable by appropriate execution sequences.

Although a general proof methodology is not developed in detail, complete proofs have been worked out for several sample systems. The methodology utilized in these proofs, in addition to aiding formal verification, has proven a significant aid to understanding the operation of such asynchronous, nondeterminate systems.

G. LANGUAGE DEVELOPMENT

In continuation of our studies on the expressive power of the data flow language [7], we have been examining the use of this language for the expression of non-determinate computation. A study of the merge operator as a basic approach to representing such computations has led to an exceptionally clean expression of a mock airline reservation system (ARS).

The expression of the ARS algorithm in this form eliminates certain problems associated with a sequential implementation of such a system. A sequential implementation could be determinate, but would not be efficient due to such physical constraints as the differing access times of various storage media. To provide an efficient implementation of the ARS which avoids these limitations requires the exploitation of parallelism within the system and, hence, the use of programming constructs such as the merge operator.

To illustrate this approach we have constructed program modules for a very simple form of airline reservation system (ARS): our airline operates M flights and there are N agents that handle reservations for airline customers (each flight operates exactly once), where M and N are determined by information about flights and agents provided as inputs to the ARS module. This module processes a stream of requests, each tagged with the agent's identity, and yields a stream of tagged responses. The data base for the ARS is a record of the number of seats booked on each flights and its seating capacity. Three kinds of requests are handled: request to reserve or cancel a specified number of seats, or a request for information about the availability of seats. The response indicates success or failure for "RES" or "CAN" requests, and the seats booked and available for an "INFO" request. Figure 11 gives the input and output data types for the ARS module.

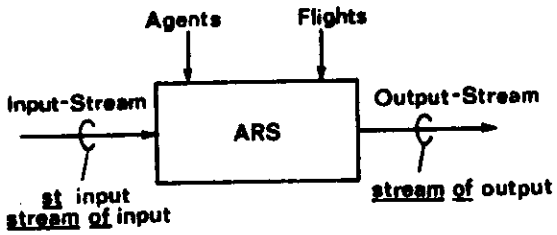

Figure 11. Input and output data types for the airline reservation system.

Figure 12 illustrates the internal operation of the ARS module. The module is composed of two types of processing modules: agent modules (A_mod) and flight modules (F_mod). Requests received by the system are distributed to the agent modules, and the agent modules in turn distribute each request to a flight module according to the flight specification in the request. Note that the data base in this system amounts to the collection of flight modules to which the requests are distributed.

A program describing the ARS system illustrated in Figure 12 is presented in Figure 13. The flight module and agent module programs are determinate programs which are specified in Figures 14 and 15.

The ARS module illustrated in Figure 13 receives a stream of requests from agents on its input stream and handles the requests through appropriate calls to the flight and agent modules, F_mod and A_mod, yielding the appropriate response on the output stream.

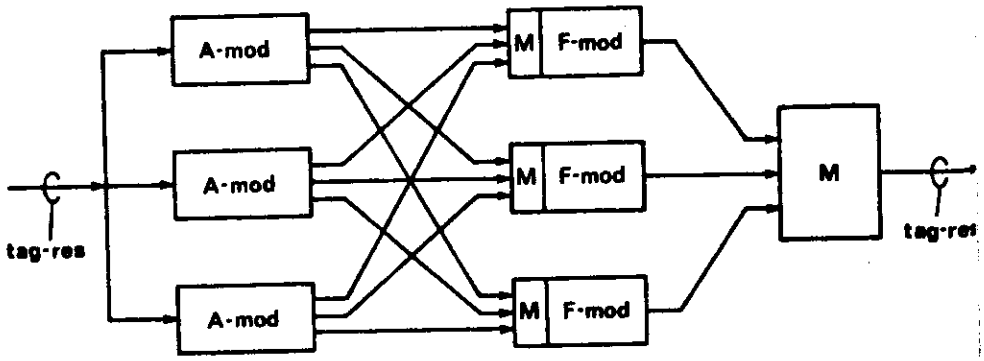

Figure 12. Internal operation of the ARS module.

```
ARS := module (
        Flights: array integer of integer,
        Agents: array string of null,
        Input-Stream: stream of input
        yields Output-Stream: stream of output);

        type input = [agnt: string, req: request];
        type output = [agnt: string, res: response];

        type request = one of [
                INFO: [fl: integer],
                RES, CAN: [fl: integer, quant: integer]];
        type response = one of [
                INFO: [booked, avail: integer],
                RES, CAN: boolean];

        Output-Stream := for all f in Flights merge
                F-mod (Flights [f],
                        forall a in Agents merge
                A-mod (a, f, Input-Stream)
                        );
end ARS;
```

Figure 13. ARS Program Module.

The key program construct in the ARS description of Figure 13 has the form

forall f *in* Flights *merge* <expression>

where f is a free identifier in <expression>, and Flights is bound to a data structure in which flight numbers occur as selectors. The effect of evaluating such a construct is to form a collection of streams by evaluating <expression> with f bound to each selector of Flights; these streams are merged nondeterministically to produce a single stream. In the ARS module, this construct is used to merge the streams of requests that each agent generates for a particular flight f into one stream that is processed by an instance of the request processing module associated with flight f. The construct is also used to merge the streams of responses for each flight into the output stream of the ARS module.

The flight module F_mod and the agent module A_mod, illustrated in Figures 14 and 15, are examples of modules whose function is expressed by means of the recursive definition of a function on streams. The Process module of Figure 14 has two arguments: the stream of requests to be processed and the state of booking for the flights. It specifies that the job to be done is to act on the first request, determine the appropriate response and the new state of booking, and append this response to the response stream that results from processing the remainder of the input in the new state.

The A_mod is similarly structured as a recursive module which forms an output stream for each of the flight modules of the agent requests for that module.

```
F-mod := module (capacity: integer, in: stream of input
      yields out: stream of output);

      Process := recmod (x: stream of input, booked: integer
            returns y: stream of output);

            if empty (x) then []
                  else
            first, rest := first (x), rest (x);
            agent, request := first.agnt, first.req;
            quantity := request.quant;

            response, new = case request of
                  tag INFO:
                        make INFO:[booked: booked, avail: capacity - booked]
                  tag RES: if booked + quantity ≥ 0
                        then make RES: true, booked + quantity
                        else make RES: false, booked
                  tag CAN:  if booked - quantity ≥ 0
                        then make CAN: true, booked - quantity
                        else make CAN: false, booked
                  endcase;
            y := cons([agent: agent, res: response], Process(rest, new));
            end Process;

      out := Process (in, 0);
end F-mod;
```

Figure 14. Program description of the Flight Module.

```
A-mod := module (agnt: string, flt: integer, in: stream of input
      yields out: stream of output);

      Select := recmod (x: stream of input yields y: stream of output);
            if empty (x) then []
                  else
            first, rest := first (x), rest (x);
            agent, request := first.agnt, first.req;
            flight := request.fl;

            out := if agent = agnt and flight = flt
                  then cons (first, Select (rest))
                    else Select (rest);
            end Select;
      out = Select (in);
end A-mod;
```

Figure 15. Program description of the Agent Module.

REFERENCES

1. Dennis, Jack B.; Misunas, David P.; and Leung, Clement K. C. A Highly Parallel Processor Using a Data Flow Machine Language. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 134. Cambridge, Ma., 1976.

2. Gold, Bernard, and Rader, Charles M. Digital Processing of Signals. New York: McGraw-Hill, 1969.

3. Commoner, Frederic; Holt, Anatol W.; Even, S.; and Pnueli, A. "Marked Directed Graphs." Journal of Computer and System Sciences Vol. 5 No. 5 (October 1971), 511-523.

4. Karp, Richard M., and Miller, Raymond E. "Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing." SIAM Journal of Applied Mathematics Vol. 14 No. 6 (November 1966), 1390-1411.

5. Dennis, Jack B., and Weng, Kung-Song. "Application of Data Flow Computation to the Weather Problem." Proceedings of the Symposium of High Speed Computer and Algorithm Organization. New York: Institute of Electrical and Electronics Engineers, 1977.

6. Kalnay-Rivas, E.; Bayliss, A.; and Storch, J. "Experiments with the 4th Order GISS Model of the Global Atmosphere." Proceedings of the Conference on Simulation of Large-Scale Atmospheric Processes, Annalen der Meteorologie Vol. 11, 1976, 25-31.

7. Weng, Kung-Song. "Stream-Oriented Computation in Recursive Data Flow Schemas." M.I.T., Laboratory for Computer Science, LCS/TM-68. Cambridge, Ma., 1975.

8. Miranker, Glen S. "Implementation Issues in Data Flow Architectures." unpublished S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, 1977.

9. Laboratory for Computer Science Progress Report XIII. M.I.T., Laboratory for Computer Science, 1976.

10. Scott, Dana, and Strachey, Christopher. "Toward a Mathematical Semantics for Computer Languages." Oxford University, Computing Laboratory, Technical Monograph PRG-6. Oxford, England, 1971.

11. Kosinski, Paul R. "Mathematical Semantics and Data Flow Programming." Proceedings ACM Symposium on Principles of Programming Languages. New York: Association for Computing Machinery, 1976.

12. Plotkin, G. D. "A Powerdomain Construction." SIAM Journal of Computing Vol. 5 No. 3 (September 1976), 452-487.

13. Lehmann, D. J. "Categories for Fixpoint Semantics." Proceedings of the Seventeenth Annual Symposium on Foundations of Computer Science. New York: Association for Computing Machinery, 1976.