Massachusetts Institute of Technology
Laboratory for Computer Science

Computation Structures Group Memo 169

Computation Structures Group Progress Report 1975 - 1976

This research was supported in part by the National Science Foundation under grants DCR 74-21892 and DCR 75-04060, and in part by the Advanced Research Projects Agency of the Department of Defense under contract N00014-75-C-0661.

October 1978

COMPUTATION STRUCTURES

Academic Staff

J. B. Dennis, Group Leader	S. S. Patil
C. A. Ellis	H. Weber

Research Staff

D. P. Misunas	A. R. Neczwid

Graduate Students

K. Amikura	M. H. T. Hack
S. A. Borkin	D. L. Isaman
G. A. Boughton	P. R. Kosinski
J. D. Brock	C. K. Leung
R. E. Bryant	G. S. Miranker
D. J. Ellis	K. S. Weng
F. C. Furtek	

Undergraduate Students

K. P. Carey	R. G. Jacobsen
J. R. Chermak	B. J. Miglierina
L. Crooks	A. E. Tracht
T. B. Freeman	T. H. Warner
P. C. Gossett	

Support Staff

A. L. Rubin

COMPUTATION STRUCTURES

A. INTRODUCTION

Research in the past year has concentrated on the further study of semantic foundations of languages and systems and the use of packet communication architecture as a basis for computer system design. Work on semantic foundations includes investigation of the formal specification and semantics of parallel programming languages, examination of equivalence problems in database systems, and the application of Petri nets to the understanding of complex systems. The study of packet communication architecture involves the design of computer systems which have such structure, the development of a formal semantics of memory systems with packet communication architecture, and the development of simulation facilities to be used as tools in the investigation of such structures.

B. THE LOGIC OF SYSTEMS

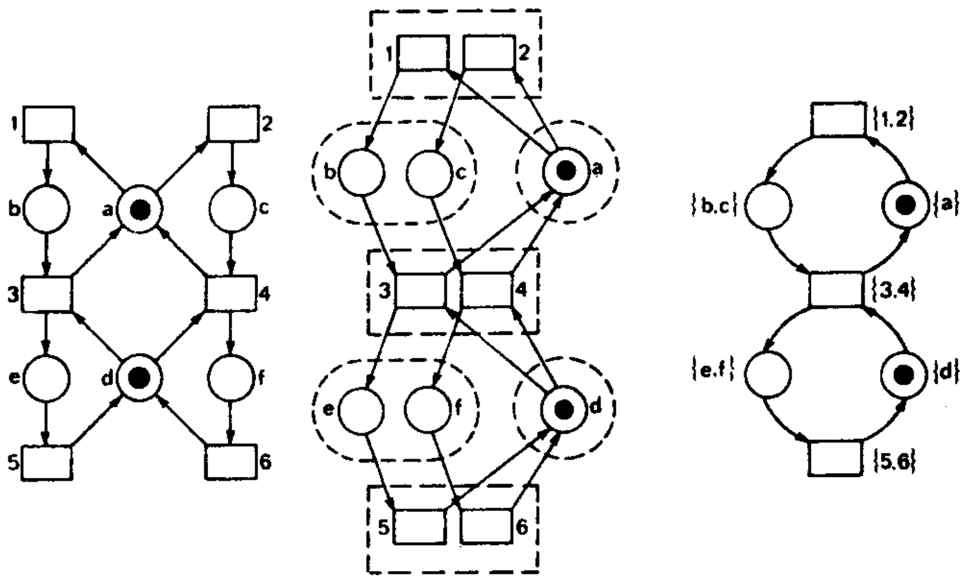
Petri nets have developed as an important mathematical formalism for describing those systems in which both concurrency and nondeterminacy play a prominent role. Unfortunately, the usefulness of these nets has been limited by a lack of analytical techniques for analyzing general classes of nets.

Fred Furtek has developed an approach that treats important kinds of nets which were previously outside the scope of any theory. Although the structure of these nets is restricted -- somewhat severely, in fact -- it is not yet clear to what extent, if any, their representational power is restricted.

The basic idea underlying the approach is quite natural: a system is represented by a Petri net satisfying certain assumptions. these assumptions permit system behavior to be separated into two components: information and control. Information has to do with choices and how choices are resolved. Control is concerned with the fixed, repetitive aspects of behavior, those aspects independent of choices.

The first step in the analysis is to extract the control component of behavior. A standard mathematical technique is used: the construction of a quotient system. This involves "folding" the original net to produce the control structure of the system, as illustrated in Figure 1. Each state (event) in the control structure consists of a set of states (events) from the original net -- these composite states and events are called alternative classes. The essential characteristic of the control structure is the absence of choices (branching on states), which means that the control structure is an event graph, a type of Petri net having very regular behavior. As with any quotient system, the control structure loses certain features of the original system, primarily the ability to distinguish between alternatives (elements in the same alternative class).

Figure 2a is a simulation of the Petri net in Figure 1a, and Figure 2b shows the structure that results when the elements in that simulation are replaced by their alternative classes. This new structure is a simulation of the control structure in Figure 1c. Note that different system simulations may correspond to the same control simulation. For example, the system simulation in Figure 3 also corresponds to the control simulation in Figure 2b.



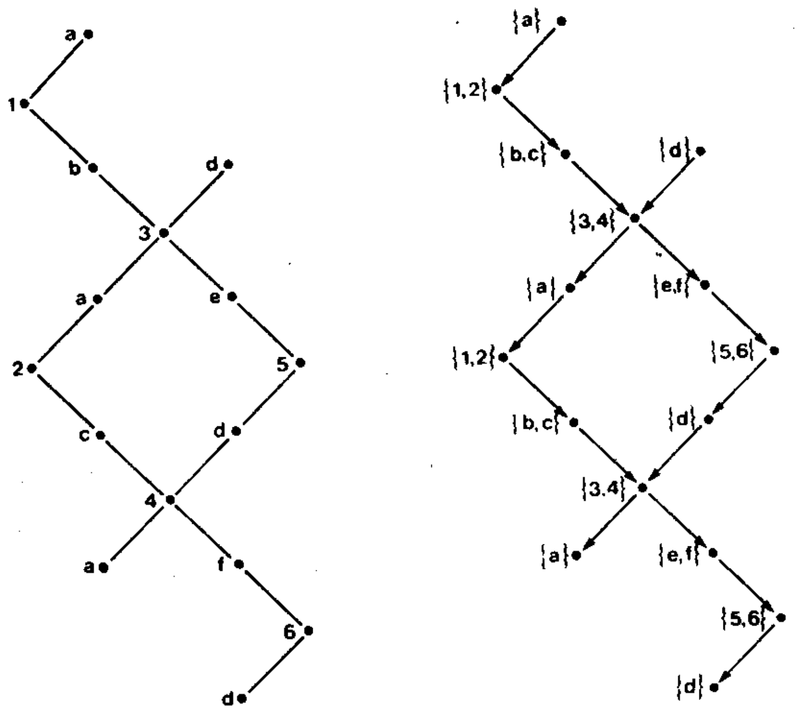
(a) Original Petri Net (b) 'Folded Net' (c) Control Structure

Figure 1. Generating the control structure.

Having succeeded in eliminating all aspects of choice in our quotient systems, we now wish to reintroduce those same aspects -- but in a special way. We assume that the original Petri net is decomposable into modes, each mode being a subnet isomorphic to the control structure. A covering of modes for the Petri net in Figure 1a is given in Figure 4. (It is convenient to identify each mode with a color.) The information content of an element (state or event) in the original Petri net is now defined as the set of modes excluded from that element. The information contents for the elements in our example are as shown in Figure 5. Note that because states a and d are excluded from neither mode, their information contents are both null.

An element in the original Petri net is thus identified by specifying two things: (1) the alternative class to which it belongs and (2) its information content. The advantage of associating colors with the modes is that we can think of information as colors assigned to the "tokens" on the control structure.

"Information flow" can be represented quite vividly by coloring the arcs of a system simulation according to a simple algorithm: an arc connecting Elements x_1 and x_2 is assigned a given color iff the mode associated with that color is excluded from both x_1 and x_2. Figure 6 illustrates the result when this algorithm is applied to the system simulation in Figure 2a. In Figure 6 we note that colors "appear" and "disappear" only at occurrences of events. Thus, we define the information gain (loss) of an event as the information that appears (disappears) at each occurrence of that event. This yields the following result which is consistent with intuition.



(a) A System Simulation (b) Corresponding Control Simulation

Figure 2. Corresponding simulations.

Information is gained by a system at precisely those points where there is forward conflict and is lost at precisely those points where there is backward conflict. Furthermore, the information gained or lost in a conflict situation is exactly equivalent to specifying how the conflict is resolved. For example, in Figure 6, events 1 and 2 are in forward conflict, and it is at these two events that information is gained. Similarly, we see that Events 5 and 6 are in backward conflict, and it is at these events that information is lost.

This forms the essence of the approach for decomposing system behavior into a control component and an informational component. Of course, the value of this approach will be determined in large part by the analytical tools it provides. This is the most attractive part of the theory because the regular properties associated with the control structure together with the ability to trace information flow provides a powerful technique for predicting and describing behavior. This ability is fundamental to solving a great many problems in the systems area.

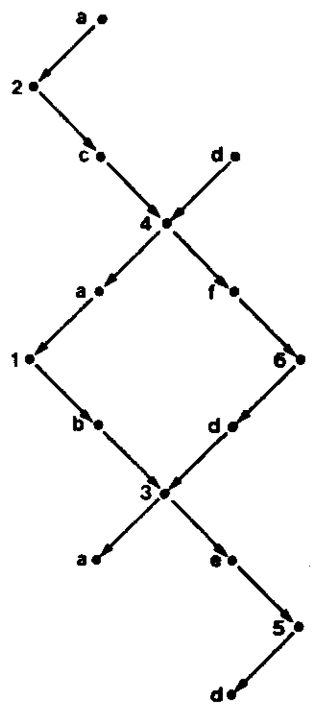


Figure 3. A system simulation.

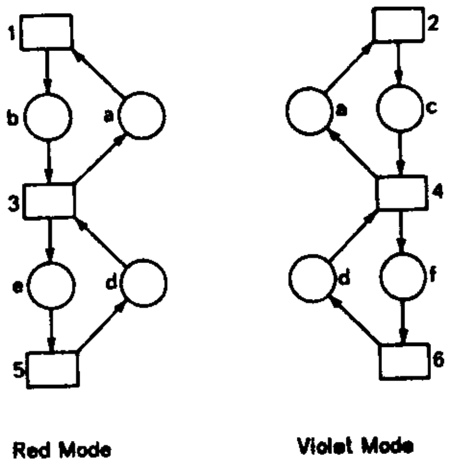


Figure 4. Modes.

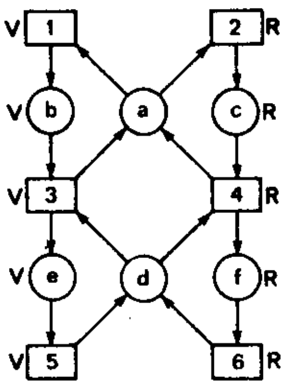


Figure 5. Information contents.

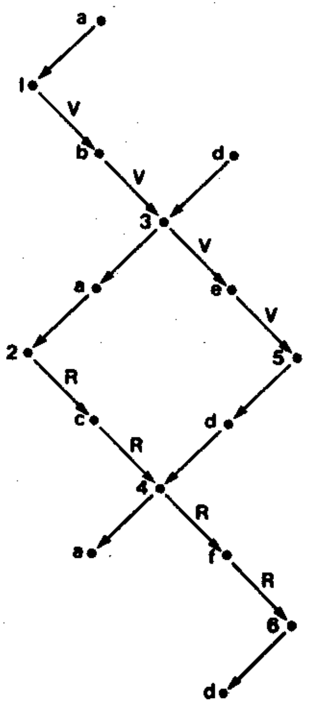


Figure 6. Information flow.

C. DECIDABILITY QUESTIONS FOR PETRI NETS

In order to utilize Petri nets as an abstract model for concurrent systems, one must first gain an understanding of their mathematical properties. The decidability of various problems which arise in this context is an important aspect of this question. The fact that these problems also arise in the context of other mathematical theories provides further motivation.

In his Ph.D. disseratation Michel Hack has completed the work described in a previous report [2]. In the dissertation, he investigates a number of these decidability questions and shows that a number of Petri net problems are recursively equivalent to the Reachability Problem for Vector Addition Systems -- whose decidability is still an open question. These problems include the Liveness Problem (e.g. can a given system reach a deadlocked state?), the single-place reachability problem (can a given buffer ever be emptied?), the persistence problem (can a given transition ever be disabled by the firing of another transition?), and the membership and emptiness problems for certain classes of languages generated by Petri nets.

D. FORMAL SEMANTICS FOR PARALLEL PROGRAMMING LANGUAGES

J. Brock is working on his master's thesis, "Formal Specification of a Language with a Monitor Construct." This research program is divided into three areas:

1. The design of a language, with a monitor construct, in which well-structured parallel programs may be written
2. The formal specification of the syntax and the semantics of this language
3. The derivation of proof methods for proving not only "partial correctness" of programs written in this language, but also for proving behavioral properties, such as the absence of deadlock and starvation.

The monitor construct was developed by Hansen [3] and Hoare [4] for use in programs performing operating system resource allocation. A monitor consists of local "own" data, own data initialization code, and a set of monitor procedures. Monitors have two properties for synchronizing monitor procedure calls: mutual exclusion of calling processes and conditional variables. The former allows only one calling process actively executing within the monitor at any moment. The latter, through use of the statements c.wait and c.signal, where c is the name of a conditional variable, enables monitors to form queues of suspended processes. If c.wait is executed during a monitor procedure call, the calling process is suspended and placed on a queue associated with c. If c.signal is executed and the queue associated with c is non-empty, a process is dequeued and reactivated. The monitor programmer is expected to structure the monitor by defining on the monitor's own data an invariant, I, which is to hold whenever no process is actively executing the monitor, and by associating with each conditional variable c a Boolean predicate C, such that C holds before each c.signal statement and, consequently, after each c.wait statement.

This language design research has three goals. First, the monitor construct is to be modified so that monitors may be written to be complete, efficient representations of operating system resources, instead of schedulers of resources. The Programming Methodology Group has shown the usefulness of data abstractions in structured programs and has designed the CLU language in which "clusters" may be used to implement both the representation and operations of data abstraction. Unfortunately, due to their mutual exclusion properties, monitors, unlike clusters of CLU, cannot generally contain the actual protected resource and the operations on it. For example, a monitor representation of a data base could not allow several read operations to proceed simultaneously. There are two problems with separating the scheduling on a resource, by a monitor, from the resource and actual operations: (1) the resource is inadequately protected when its user is expected to use the monitor protecting it. (2) the correctness of the operations on the protected data base depends on the scheduling properties of the monitor. This complicates the proof of correctness, and understanding of these operations as these scheduling properties must be brought outside of the monitor.

The second goal of the language design research is the delineation of operations for the creation, and possibly the assignment of monitors. In [4], monitors are considered to be global operating system fixtures; however, in many parallel programming applications, it will be necessary to create monitors. For example, in an airline reservation system, monitors representing flight records must be created when needed. The last, and most elusive, goal is the discovery of a modification of the monitor construct that would lessen the possibilities of deadlock and starvation.

E. FORMAL EQUIVALENCE PROBLEMS IN DATABASE SYSTEMS

A user of a database system always has some view of the structure of the data in the database and knowledge of the operations which can be used to alter the database. The types of data structures visible to the user and the operations allowed on these structures determine the database system's data model. The database system may just be a secondary storage file system in which the user interacts directly at the storage device level. In this case, a user's view of the database may consist of a set of files with certain indexing characteristics. The operations are those appropriate to defining and modifying the files.

In an effort to simplify the user's view of the data, to allow for portability across different machines and to allow for easier growth of databases, database systems were designed with greater degrees of physical data independence. The data models used in data independent systems let the user define the logical structure of the data and perform operations without regard to lower level implementation details. Such data models are called logical data models. Existing implementations of such data models present to the user structures such as trees, tables, or linked lists along with appropriate operations.

It is not obvious how to best use a particular data model for a particular application. Nor is it clear whether one data model can represent all information representable in another data model. This has led to the investigation of semantic data models. These are logical data models in which the structures are explicitly meant to represent certain types of objects or concepts which can occur in the real world. The operations defined on these data structures are meant to represent ways in which the state of the real world can change.

Given the variety of data models available, there are several reasons for developing a comparison between the models. Only through an understanding of their features can a user decide which model, if any, is most appropriate for a particular application. The possibility that different applications might desire to see the "same" data through the perspective of different data models requires a definition of what it means for databases founded on different data models to be equivalent. Finally, a desire to translate a database founded on one data model into a database founded on another data model also requires an understanding of equivalence between different data models. It also must be determined if such an equivalence can always be defined; it may be the case that the "expressive power" of data models differs.

Sheldon Borkin has been investigating the problem of formally describing and proving the equivalence of two semantic data models. The first data model being considered, the semantic network data model, is similar to that presented by Schmid and Swenson [5]. This data model postulates two types of semantic objects: entities such as people and companies, and associations between entities such as a person being employed by a company. Both entities and associations may have characteristics such as a person's age or the date a person was hired by a company. The applicable operations are the insertion and deletion of entities and associations. In a database using this data model, a one-to-one correspondence is assumed between objects in the database and objects in the "real world" about which information is being stored.

The second data model being considered is an extension of the relational model of Codd [6]. A database in this model is defined as a set of relations along with a set of constraints. A relation can be viewed as a table where each row represents some true statement about the world being described in the database. A relation is the set of all true statements fitting a certain pattern. For example, a relation might represent all statements of the form: "__ is an employee and operates the __ machine" where the blanks must be filled in with appropriate values. A row in this relation with the values "John, drill press" would represent the statement: "John is an employee and operates the drill press machine." Constraints specify certain conditions which must always be true of the relations in the database. For example, in the relation just described, it might be required that each different machine have only one operator. The operations allowed in this data model are the insertion and deletion of sets of rows in the relations so long as the resulting relations are consistent with the set of declared constraints.

We have developed formal set theoretic definitions for both of the data models -- including both structures and operators. A data model is formally defined as a set of possible databases and a set of operations which map one database into another. A database has two components: a schema and a state. A schema contains the declarative information defining the structures allowed in the database state. For example, the schema of a relational database defines the relations and the "statement" which each row in the relation represents, the types of values allowed in each column of each relation and the constraints on the relations. The state of the relational database consists of the set of rows of each relation representing specific statements about the world being represented in the database.

Data model equivalence is defined for databases with fixed schemas (i.e. consideration is given only to operations which change the state) in a two step process. First, a mapping is defined between the possible schemas of the two data models. This mapping defines equivalent schemas for which it is then shown that: (i) there is a one-to-one mapping between states in the databases (for given schemas) defining equivalent states, and (ii) for any operation in either data model there must be an operation in the other data model such that the two operations map equivalent states into equivalent states. Note that the mapping defining equivalent schemas need not be one-to-one. If there is a many-to-one mapping, this would mean that there are several databases in one data model equivalent to a single database in the other data model.

To define the required mappings and prove the properties outlined above requires a formal proof of data model equivalence. This work is currently being done for the two data models previously described. The equivalence mappings being defined will equate several relational data model schemas with each semantic network data model schema. There are thus several relational "views" of the information stored in a single semantic network database.

F. SYSTEMS OF DATA STRUCTURING OPERATIONS

One issue that continues to be of great interest to us is the occurrence of cyclic structures in the runtime data structures of programs in execution. Previous studies have shown that powerful languages can be implemented without requiring cyclic runtime structures. Whether cyclic structures are in some sense a necessary accompaniment to essential imperative constructs of programming languages, or are essential to the support of database computations is a question that continues to be debated within our group. David Isaman has explored the other side of this question, namely, to show in what sense avoiding the occurrence of cyclic structures can yield more efficient computer architectures.

This work on acyclic data structures examines the design of computer memories capable of directly storing these structures. Primitive memory operations include "select a named substructure of a given structure" and "make one structure a named component of another." Two alternate systems of data structure operations are under investigation, along with their implications for parallel processor design.

A Structure Memory (SM) stores directed graphs, with scalar values attached to the nodes, and selectors labelling the branches. The graphs are restricted only in that the branches emanating from any given node always have distinct selectors. The SM representation of the LISP list (1 2 3) is shown in Figure 7.

External communication of a structure value is by means of a pointer, which is associated by the SM with the root node of that structure value. The primitive operations of the SM include:

Fetch, Assign -- Given (a pointer to) a node *n* read or write the single scalar value attached to *n*.

Select -- Given a node *m* and selector *s*, return a pointer to the node at the end of that branch leaving *m* which is labelled with *s* (Figure 8a).

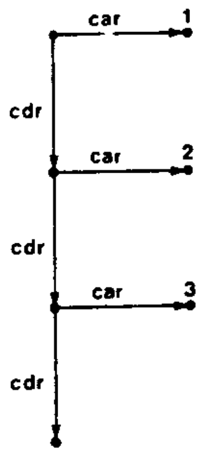


Figure 7. Structure memory representation of a LISP list.

Delete -- Remove that branch from *m* which is labelled with *s*.

Update -- Add a branch from *m* to *n*, and label it with *s*.

First, Next -- Enumerate the set of selectors of branches leaving *m*.

Copy -- Create a new node *m'*, with the same scalar value as *m*, and with branches to the same other nodes, labelled with the same selectors as *m* (Figure 8b).

One immediate consequence of unrestricted use of the above operations in a parallel processor is the danger of nondeterminacy. For example, if a Fetch and an Assign have as inputs pointers to the same node, the outcome of one may depend on whether the other executed first. Whenever the relative execution order of two such conflicting operators is not fixed, nondeterminacy arises.

There are two alternate systems in which non-determinacy is easily eliminated. In the Structure-as-Value (S-V) system, the write-class operations (Assign, Delete, Update) are always coupled with a Copy. The change is made to the new node output by the Copy, that node is available to no other operators until it has been changed, and once it is available, that node represents an unchanging structure value. Figure 9 illustrates an S-V program to perform the concatenation of two list structures. (Append is the coupling of Update with Copy.) All programs using the S-V system of operations are determinate.

The second alternative is the Structure-as-Storage (S-S) system. Nondeterminacy is controlled by the combination of a syntactic constraint on programs and an execution constraint on Select operations. The syntactic constraint, the Determinacy Condition, is a local condition on each distribution group: those operators taking their pointer inputs from the output of a single other operator. Any two operators in a distribution group must be sequenced if either: 1) they potentially conflict, or 2) both are Selects with potentially equal selector inputs.

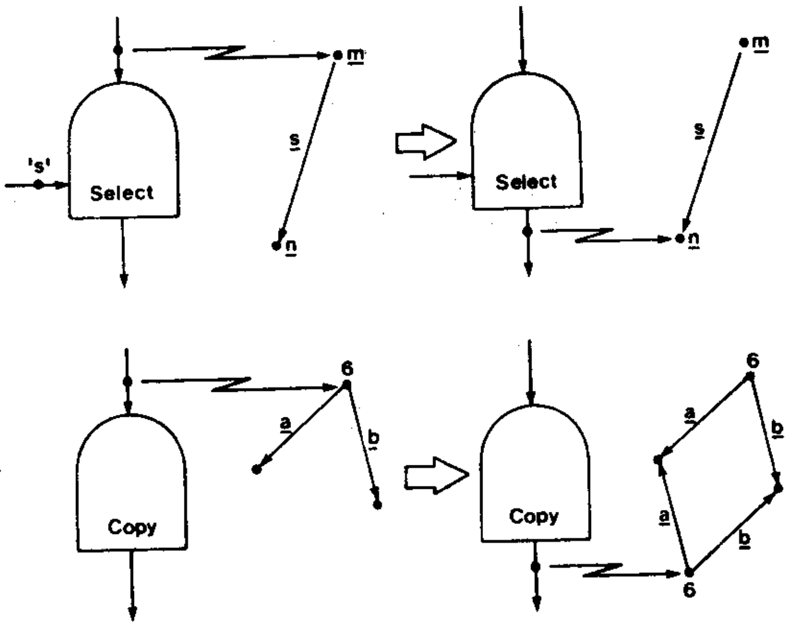


Figure 8. Primitive structure operations.

The necessity for the first constraint is evident from the fact that all operators in a distribution group necessarily operate on the same node. Figure 10 illustrates how the second part of the Determinacy Condition helps coordinate parallel processes accessing a common data structure (the inputs X and Y are the same as those in Figure 9). This program contains two parallel processes, whose inputs are the outputs of Selects S_1 and S_2 The sequencer (seq) operator ensures that S_1 executes before S_2, as the Determinacy Condition directs. The final operators, S_5 and S_6, operate on the same node, and so must always execute in the same sequence. A syntactic sequencing is unacceptable in the more general case of variable selector inputs: the processes' paths may then diverge, so that no conflict ever arise. To preserve maximum concurrency, any sequencing of operator executions must be sensitive to whether their inputs actually point to the same node.

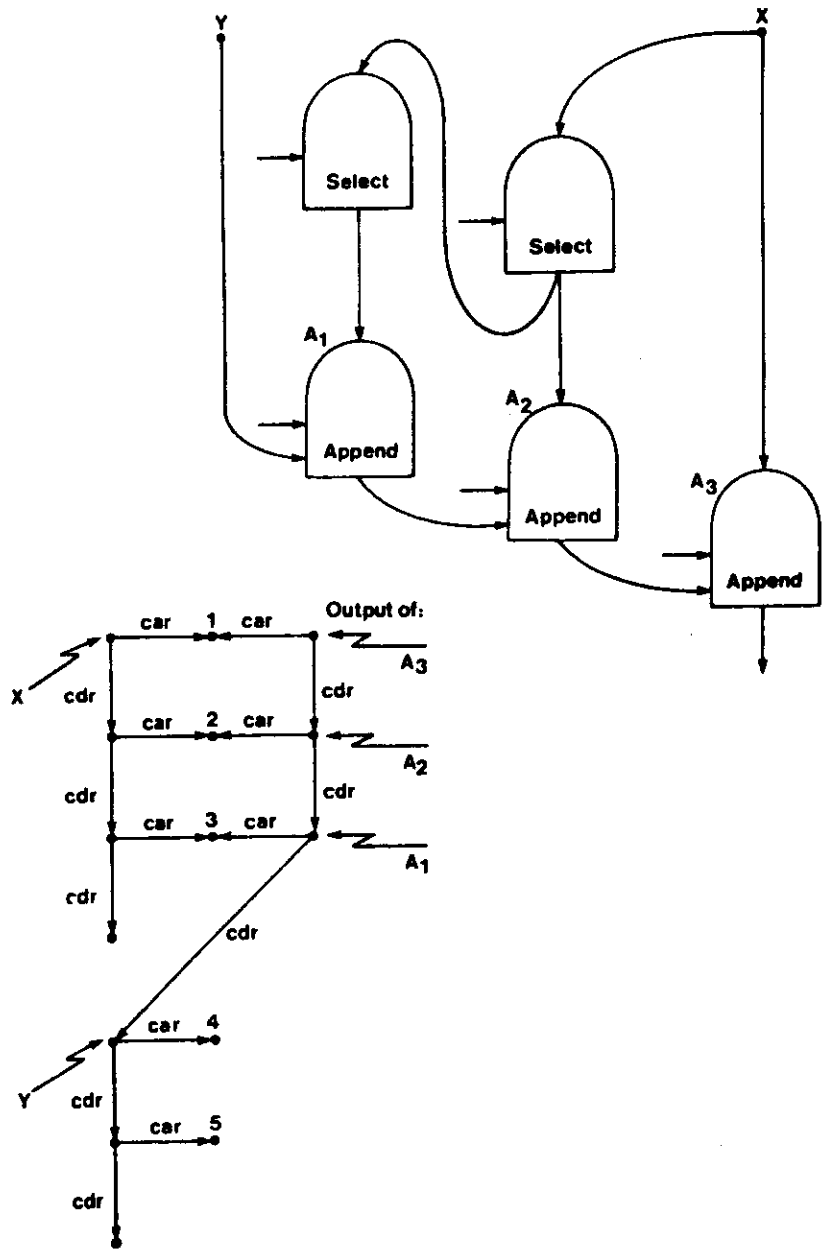


Figure 9. S-V program to concatenate two list structures.

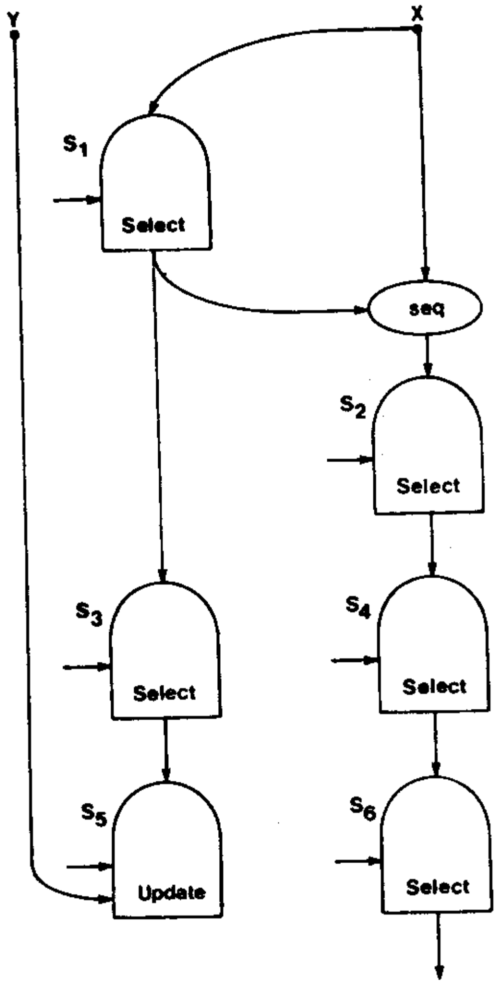


Figure 10. An example S-V program.

This sensitivity is provided by the restriction on Select executions, which is built on the reference-counting method of structure storage reclamation: every execution of a Select having r output arcs creates r pointers to a node n; this requires that at least r more executions with n as input occur before n can be reclaimed. Thus there is available an execution reference count, ERC(n), of the number of outstanding pointers to n. Each Select execution is restricted as follows: if n is the node which would be output, then that execution is blocked (will not occur) so long as ERC(n) > 0. The effect of this in the program of Figure 10 is explained next.

S_2 cannot execute before S_1, due to the sequencer. Since S_2 selects the same node n_2 as S_1, it is further blocked until $ERC(n_2) = 0$; that is, until S_3 has executed. S_4, of course, cannot execute before S_2, so S_4 cannot execute before S_3. Furthermore, since S_4 outputs a pointer to the same node n_3 as S_3, it cannot execute until $ERC(n_3) = 0$; i.e., until S_5 executes. S_6 cannot execute until S_4 does, so the Select S_6 can never execute before the Update S_5, if they conflict. If S_5 and S_6 have different nodes as inputs, S_4 will not be blocked, and the rest of the two processes will be fully concurrent.

This is the alternative Structure-as-Storage system: the original SM operations, plus the syntactic Determinacy Condition and the dynamic blocking of Select executions. This system guarantees that two parallel processes never conflict at a node they have reached via identical paths.

Any program written to run in the S-V system can be easily rewritten to run determinately in the S-S system, and the two programs will be equivalent. Furthermore, the equivalent S-S program can often be optimized to produce a program which is capable of exploiting up to twice as much of the available parallelism. Major contributions of this work include comparative analysis of the tradeoffs between the two systems in terms of expressive power and efficiency of execution, the development of schemes to make the two systems behave comparably, and the investigation of techniques used in proving the schemes' correctness.

G. DATA FLOW SEMANTICS

A Data Flow Program is a directed graph whose nodes are operators and whose arcs are data paths [7, 8]. Operators in a Data Flow Programming Language (DFPL) functionally transform their inputs to their outputs without affecting the state of the rest of the program. Since there is no control flow, there is no GOTO; in spite of this, iterations may be programmed, as well as recursion. Most significant, though, is the fact that unlike ordinary applicative languages, programs may exhibit memory behavior; that is, the current output may depend on past inputs as well as the current input. The effects of memory are local like those of other operators, and they do not permeate the semantics of programs.

Data in DFPL are pure values, either simple numbers or structures such as arrays or records. There are no addresses in DFPL, although certain operators may be programmed to interpret input values in a manner reminiscent of addresses. An operator "fires" when its required inputs are available on its incoming paths. After a variable amount of time, it sends its outputs on its outgoing paths. It may not be necessary for all inputs to be present before an operator fires. Similarly, not all outputs may be produced by a given firing. Synchronous operators fire only when all their inputs are present and produce their outputs all at once, such operators are analogous to subroutines. Some operators produce a time sequence of output values from one input value and operate in a manner similar to coroutines. The operators in a DFPL program operate in parallel with one another, subject only to the availability of data on the paths.

An operator may be either primitive or defined. An operator is defined as a network of other operators which are connected by data paths such that certain paths are connected on one end only. These paths are the parameters of the defined operator. A defined operator acts as if its node were replaced by the network which defines it and the parameter paths spliced to the paths which were connected to that node. In this manner, recursive operators are defined.

The DFPL developed by Paul Kosinski consists of the five primitive operators shown in Figure 11. Of these, two are simple in their behavior: the Fork and the Primitive computational function (Pcf). This latter is really a whole class of operators including the usual arithmetic, logical and aggregate operators (e.g. construct and select). These two operators have the property that they demand all their inputs and produce all their outputs each time they fire. Furthermore, each firing is independent of any past history, that is, the operator defines a function from current input to current output. The functional equations for the Pcf and Fork operators shown in Figure 11 are thus:

$X = F_x(U,V,W)$ & $Y = F_y(U,V,W)$ for the Pcf F,
$X = U$ & $Y = U$ & $Z = U$ for the Fork.

The most complicated operators are the Switch operators, also shown in Figure 11. These operators have the property that although each firing is independent of previous firings, not all inputs/outputs are demanded/produced upon each firing. The Outbound Switch, for example, demands C and U as inputs for each firing, but only one of X, Y and Z receives the output value U, determined by the value received on input C. The Inbound Switch operates conversely, only one of the inputs X, Y and Z is accepted upon firing (C is demanded), and its value is always sent out on U.

Since these operators sometimes do not accept/produce inputs/outputs, we can not describe their functional behavior by such simple equations as before (not producing an output is not the same as producing a null output). But we can describe their behavior if we view them as functions from <u>sequences</u> of inputs to <u>sequences</u> of outputs. Now the functional equations for both kinds of Switches are (1-origin indexing is assumed):

$U^* = \text{Inswitch}\ (C^*,X^*,Z^*)$ and
$X^* = \text{Outswitchx}\ (C^*,U^*)$ &
$Y^* = \text{Outswitchy}\ (C^*,U^*)$ &
$Z^* = \text{Outswitchz}\ (C^*,U^*)$ where

$X_k = U_j$ if $C_j=1$ & $k=\#\{i\leq j|C_i=1\}$
$Y_k = U_j$ if $C_j=2$ & $k=\#\{i\leq j|C_i=2\}$
$Z_k = U_j$ if $C_j=3$ & $k=\#\{i\leq j|C_i=3\}$

The notation $\#\{i\leq j|C_i=q\}$ means the number of times the value q occurs in the first j elements of C^*.

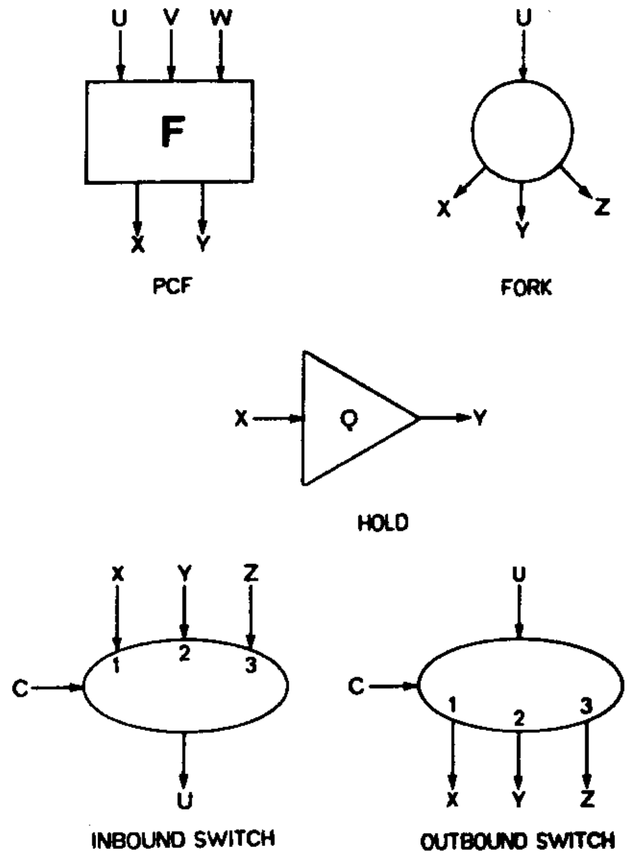


Figure 11. DFPL primitive operators.

The most interesting DFPL primitive operator is the one which behaves like a memory cell. This operator is just a holding station; that is, the output consists of the input on the previous firing. More precisely, $Y^* = \mathrm{Hold}_Q(X^*)$ where $Y_1 = Q$ & $Y_{i-1} = X_i$ $\forall$ $i>1$. The Hold operator is interesting because it is sufficient to construct any kind of memory desired, yet itself is purely and simply functional (albeit from input sequences to output sequences). This operator can also be used to construct iterations.

Now the first three operators can be recast as functions from sequences of inputs to sequences of outputs:

$X_i = C$ for the Primitive constant;
$X_i = Fx(U_i,V_i,W_i)$ &
$Y_i = Fy(U_i,V_i,W_i)$ $\forall i$ for the Pcf F; and
$X_i = U_i$ & $Y_i = U_i$ & $Z_i = U_i$ $\forall i$ for the Fork.

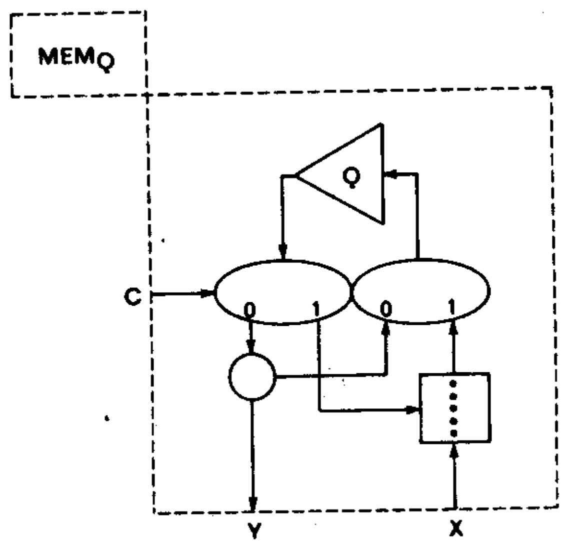


Figure 12. Structure of a memory cell.

All primitive operators are causal in the sense that an output cannot be affected by future inputs, that is, once an output is produced, it cannot be changed. More precisely, if $Y^* = F(X^*)$ & $Y_k{}^* = F(X_i{}^*)$ & $Y_h{}^* = F(X_j{}^*)$ & $j \geq i$, then $h \geq k$.

A fancier memory cell is shown in Figure 12. When a 0 value is presented on the control path C, the current contents are read out on path Y. When a 1 value is presented on C and a data value is presented on the input path X, the cell is updated to contain that new value. The cell has initial contents Q.

More complicated memories may be programmed by substituting other operators for the Fork and ⋯ operators in Figure 12. For example, by replacing the Fork by a Deque operator, and the ⋯ by an Enque, a queue memory results. To program a random access memory, another input path, to carry the "address", must be added, as well as replacing the operators.

This approach to data flow semantics is currently being broadened to include synchronization semantics (to prevent build-up of queues on the data paths) and timing dependent semantics (to study the indeterminate behavior needed to deal with the real world). Also under investigation are the algebraic relationships within and among these three levels of semantics.

H. PACKET COMMUNICATION ARCHITECTURE

In continuation of our work on packet communication architecture, we have been developing formal models for packet communication systems and further investigating the structure of data processors organized in such a manner. Packet communication architecture is the structuring of data processing systems as collections of physical units that communicate by sending information packets. Packets are routed between sections of a packet communication system by networks of units arranged to sort many packets concurrently according to their destinations. In this way, it is possible to arrange that system units are heavily used, provided concurrency in the task to be performed can be exploited.

Previously, we described the structure of three data flow processors organized in such a fashion [9, 10, 11]. We have also developed the application of this architectural principle to the organization of large memory systems capable of simultaneously processing large numbers of concurrent memory transactions [12].

1. Formal Semantics for Packet Communication Systems

As part of our research effort in packet communication architecture, we are developing methods for formally describing the behavior of systems with such structure. Two significant benefits to be derived from the formal semantic models are:

a. A firmer understanding of the behavior of packet systems, and

b. The ability to prove that specific system structures and implementations satisfy desired criteria.

One factor which makes formal specification difficult for packet communication systems is that information passes through these systems asynchronously. The notion of flow of control, which is used as an integral part of conventional program specification techniques, is not present here. Because of this difference, new approaches need to be developed.

The research efforts of David Ellis in this field have focused on formally specifying the behavior of one particular module, a packet memory system which is described informally in [12]. This memory system M (Figure 13) is designed to store values of two types: elementary values and pairs (pairs are essentially LISP CONS-cells). M communicates with a processor P through four ports over which information in the form of fixed-format packets passes.

The input ports C and S accept from P commands and values to be stored, respectively. The output port R passes values retrieved from M on to P. The output port U passes unique id's, which correspond to addresses within M. A significant aspect of the design of the memory system is that the processor P should be able to access no locations other than those denoted by unique id's that have already been received by P from M (through M's U port); M uses a reference count scheme to manage the locations currently in use.

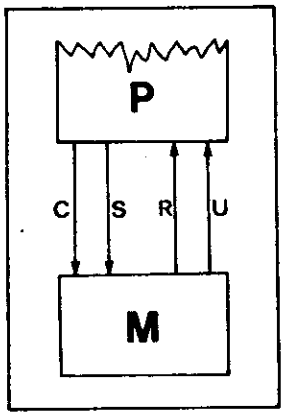


Figure 13. Packet memory system.

In the semantic model currently under development, the behavior of the memory system is captured by the notion of a history, which is a finite sequence of time-ordered events. Each event is the transmission or reception of a particular information packet at one of M's four ports connecting it with P. A formal semantic specification for the system in the model consists of three parts:

a. A list of the legal types of events, which specifies what kinds of information packets may pass through each of the ports. For instance, only commands and not values may be passed along the C line.

b. An inductively defined validity predicate on histories. This specifies the conditions under which it is semantically meaningful for each kind of legal event to take place. For example, a value may be retrieved from a given location in M only if it had previously been stored there.

c. A way of determining the expected and/or possible outputs of M from a given history, including a test for when two histories are equivalent in this respect.

A formal semantic specification has been produced for the packet memory system M, and work is currently under way in proving that certain realizations of M as interconnections of component modules do, in fact, satisfy this specification (given the proper assumptions for the behavior of the component modules).

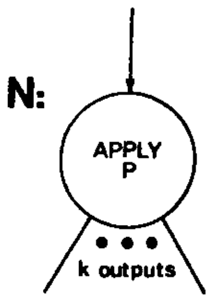


Figure 14. Procedure activation.

2. Data Flow Computer Architecture

Work in the area of data flow computer architecture has proceeded in two directions this year. First, an examination of data flow representations of signal processing computations such as the Fast Fourier Transform has resulted in the development of a processor particularly suited to applications requiring high rates of computation, but involving relatively small programs [13]. The potential performance of the processor has been analyzed, and means of preventing the occurrence of deadlock during operation of the processor have been developed.

The other research area involves the study of procedure activation on a data flow processor. The scheme developed previously for the handling of procedure applications results in a semantics which is analogous to the copy rule of Algol [11]. For each operator in a program, the data values necessary for execution are stored with the operator itself. Consequently, establishment of a procedure's activation requires not only creating a unique data area for the activation, but also setting up a new copy of the code. The problem is further complicated by the fact that computations including distinct activations (such as those arising from recursive calls) of a given procedure proceed in parallel.

The various schemes for procedure activation exploit techniques for dynamic renaming of actors of a program to affect procedure calls. Glen Miranker has investigated the implementation of these techniques through the addition of distinct memory relocation mechanisms to perform the memory mapping function.

There are several ways of invoking a procedure in a data flow language that are consistent with the data flow model. The simplest method is a single argument APPLY actor, as shown in Figure 14. The effect of APPLY P is intuitive. When a data value α arrives on the input arc, a copy of the data flow graph for P is made and α is placed on the input link of procedure P. As each of the k outputs for this activation of P is produced, it is passed from its output link to the corresponding output link of the APPLY and hence to a successor node of N. To be syntactically correct, P must have one input link, and k output links. To be semantically correct, P must be properly terminating. Briefly, this means that P produces (after some finite time) one output value on each of its output links and then undergoes a finite (possibly 0) number of additional actor firings. The structure of the machine to be used for the following implementation of single argument/single output APPLY is depicted in Figure 15.

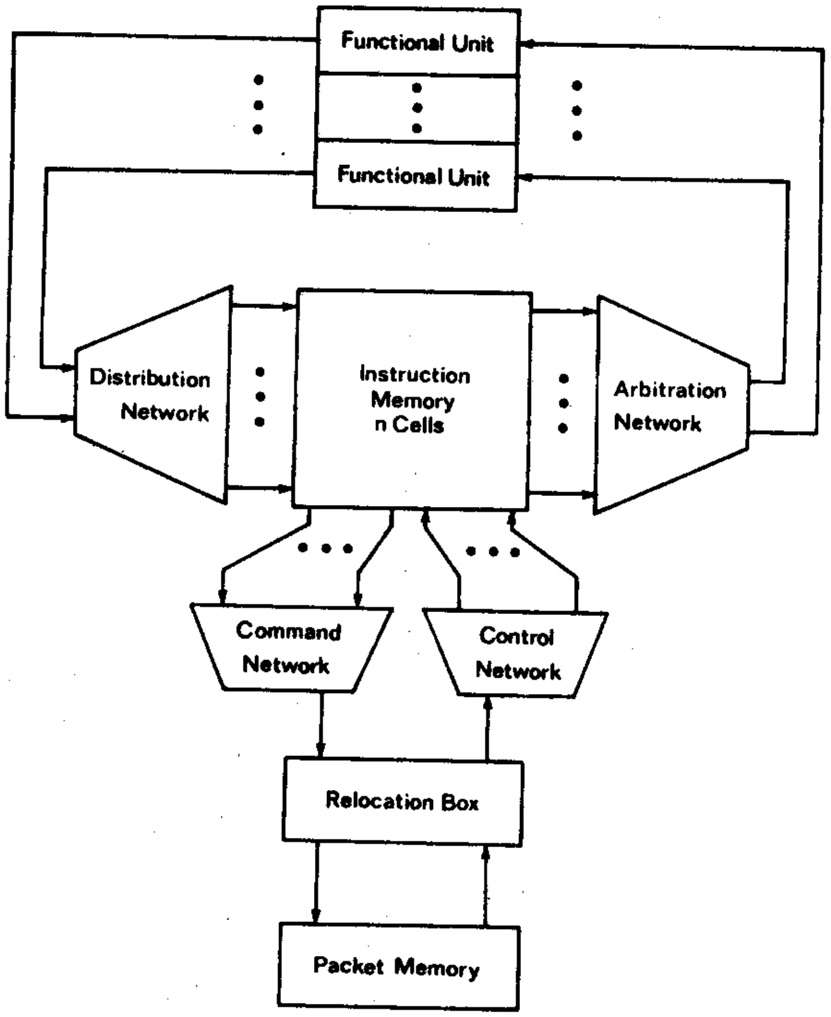


Figure 15. Structure of a data flow processor incorporating procedures.

The processor of Figure 15 is the basic data flow processor described previously [10], with one addition: the relocation box. The operation of the relocation box is quite simple. Upon receipt of a retrieve packet from the memory command network, requesting the retrieval of node α of some program from the Packet Memory:

$$\left\{ \begin{matrix} \alpha, \sigma \\ \text{retrieve} \end{matrix} \right\}$$

It passes the packet:

$$\left\{ \begin{matrix} \alpha \\ \text{retrieve} \end{matrix} \right\}$$

to the Packet Memory. When node α is returned by the Packet Memory to the Memory Control Network, all the names in its destination fields are changed to have suffix σ. The relocation box then passes the node back through the Memory Control Network to the Instruction Memory. It is assumed that with the sole exception of the relocation box and one special functional unit, no other component of the data flow processor of Figure 15 can distinguish if a node name has a suffix appended or not. That is, if the Distribution Network, for example, receives a packet with a destination $\alpha.\sigma$, it transmits the packet to destination $\alpha\sigma$ within the Instruction Memory (the dot separating the name from the suffix is included merely as an aid to the reader's eyes). The essential idea is that a complete node name (i.e. a node name plus an appended suffix) is treated everywhere but the relocation box and the distinguished functional unit as a single entity -- a node designation.

3. Fault-Tolerance in Packet Communication Architecture

The modular structure of a system with packet communication architecture permits the incorporation of fault-tolerant structure at the module level. The highly parallel nature of the interconnection networks of a packet communication system supports the existence of multiple paths between units of the system and hence offers many reconfiguration possibilities in case of component failure. Also, due to the parallel structure of the networks, processor reconfiguration does not significantly degrade either the error detection or recovery capabilities of the system.

An initial investigation of one such scheme for the fault-tolerant structure of an elementary data flow processor has yielded promising results as to the error detection, isolation, and recovery capability of such processors [14]. Hardware errors are detected through redundant computation combined with voting. Upon detection of an error, the processor can be readily reconfigured to bypass any faulty component(s).

I. SIMULATION OF PACKET COMMUNICATION ARCHITECTURES

Two simulation facilities for packet communication systems are currently under development. The concurrent development of both a microprocessor-based hardware simulation facility and a software simulator allows the exploitation of the advantages of each system and provides a novel comparison of the two techniques.

Each simulation facility receives a description of the packet communication system to be simulated in an Architecture Description Language (ADL) which has been developed solely for the purpose of describing packet communication systems. The ADL supports both the structural description and the behavioral description of a system. Intermodule packet communication is expressed with send and receive statements. The behavioral language is an extension of a subset of Pascal. Work in progress involves the further refinement of these concepts and the formal specification and documentation of a version of the language. Informal illustrations of the language are presented in [15] and [13].

The development of the hardware simulation facility was stimulated by the advent of low cost LSI processors. With such processors, it is economically feasible to divide a packet communication system into parts and emulate the operation of each part on a microprocessor. The facility we have designed consists of a number of microprocessor modules arranged so they may easily communicate through a network for the simulation of any packet communication system. A host computer translates system descriptions in the ADL into program modules executed by the microprocessors. The host computer also provides means for debugging and for measuring performance of the simulated system.

The software simulation facility is designed to be executed on a conventional sequential computer, and currently resides on a PDP-11/70. The software simulator also receives as input an ADL description of a packet communication system and some program to be executed by that system. The ADL description is used to create a processor structure which represents the simulated system. Time is broken into small discrete intervals, and a simulation proceeds by executing the program simulating the operation of any unit which is enabled during the current time interval, and then proceeding to the next interval.

REFERENCES

1. Hack, Michel H. T. Decidability Questions for Petri Nets. M.I.T., Laboratory for Computer Science, LCS/TR-161. Cambridge, Ma., 1976.

2. Laboratory for Computer Science Progress Report XII. M.I.T., Laboratory for Computer Science, Cambridge, Ma., 1975.

3. Brinch Hansen, Per. Operating System Principles. Englewood Cliffs, N.J.: Prentice-Hall, 1973.

4. Hoare, C. Anthony R. "Monitors: An Operating System Structuring Concept." Communications of the ACM, Vol. 18 No. 10 (October 1974), 549-557.

5. Schmid, Hans A., and Swenson, J. Richard. "On the Semantics of the Relational Data Model." Proceedings of the ACM SIGMOD Conference. New York: Association for Computing Machinery, 1975.

6. Codd, E. F. "A Relational Model of Data for Large Shared Data Banks." Communications of the ACM, Vol. 13 No. 6 (June 1970), 377-387.

7. Dennis, Jack B. "First Version of a Data Flow Procedure Language." Programming Symposium: Proceedings, Collogue sur la Programmation. Lecture Notes in Computer Science, Vol. 19. New York: Springer-Verlag, 1974.

8. Kosinski, Paul R. A Data Flow Programming Language. IBM Thomas J. Watson Research Center, RC4264. Yorktown Heights, N.Y., 1973.

9. Dennis, Jack B., and Misunas, David P. "A Computer Architecture for Highly Parallel Signal Processing." Proceedings of the ACM 1974 National Conference. New York: Association for Computing Machinery, 1974.

10. Dennis, Jack B., and Misunas, David P. "A Preliminary Architecture for a Basic Data-Flow Processor." Proceedings of the Second Annual Symposium on Computer Architecture. New York: Institute of Electrical and Electronics Engineers, 1975.

11. Misunas, David P. "A Computer Architecture for Data-Flow Computation." unpublished S.M. Thesis, M.I.T., Department of Electrical Engineering and Computer Science, 1975.

12. Dennis, Jack B. "Packet Communication Architecture." Proceedings of the 1975 Sagamore Computer Conference on Parallel Processing. New York: Institute of Electrical and Electronics Engineers, 1975.

13. Dennis, Jack B.; Misunas, David P; and Leung, Clement K. C. A Highly Parallel Processor Based on the Data Flow Concept. M.I.T., Laboratory for Computer Science, Computation Structures Group, Memo 134. Cambridge, Ma., 1976.

14. Misunas, David P. "Error Detection and Recovery in a Data Flow Computer." Proceedings of the 1976 International Conference on Parallel Processing. Edited by Philip H. Enslow. New York: Institute of Electrical and Electronics Engineers, 1976.

15. Leung, Clement K. C.; Misunas, David P.; Neczwid, Andrij R.; and Dennis, Jack B. "A Computer Simulation Facility for Packet Communication Architecture." Proceedings of the Third Annual Symposium on Computer Architecture. New York: Institute of Electrical and Electronics Engineers, 1976.