MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 96

Computation Structures Group
Progress Report 1972-73

This research was supported by the National Science Foundation under research grant GJ-34671.

January 1974

COMPUTATION STRUCTURES

Research in computation structures during the past year has concentrated on the development of models and analytic tools for systems of asynchronous interacting units, and the study of semantic foundations for modular and structured programming. Work on asynchronous systems includes analysis of failure probability for circuits that perform the function of arbitration or synchronization, a general approach for determining whether systems with concurrent activity are free of certain possibilities of deadlock, the use of Petri nets as a model for analysis of system throughput, and contributions to the theory of Petri nets. Work on semantic foundations consists of contributions on formal schemas of programs, study of the semantics of data base systems in terms of abstract models, and the development of semantic models that provide a suitable base for modular and structured programming. In addition, an unusual architecture has been conceived for a highly parallel stored program computer applicable to an interesting special class of computations.

1. Arbiters and Synchronizers

An arbiter is a device that mediates independent requests for use of a shared resource [52, 53]. It appears that no physical realization of an arbiter is possible that responds to each request situation in bounded time. Nevertheless, we reported last year [52] on a method of cascading primitive arbiter circuits to achieve as low a probability of error as desired.

A synchronizer is required to coordinate communication between two systems controlled by unrelated timing signals. It is easily shown that the existence of a synchronizer that responds within a fixed number of clock cycles implies the realizability of a perfect arbiter with bounded response time. Hence no perfect realization of a synchronizer is likely to exist.

The failures of synchronizers and arbiters stem from the presence of a meta-stable state in the circuit (often a flip-flop) used as the elementary decision element of an arbiter or synchronizer. The quality of an arbiter depends on the structure employed in realizing it and the quality of the elementary decision elements. Patil has developed some measures for the quality of flip-flops and other circuits used as decision elements in synchronizers and arbiters [54]. The measures are:

1. The propagation delay δ_0, which is the normal time for the circuit to respond when no critical operation of the circuit is involved.

2. A time constant τ, which is a measure of how quickly a decision element leaves its meta-stable state.

3. The conflict window W_c, which measures the likelihood of the decision element entering its meta-stable state.

The measure δ_0 of propagation delay is easily understood, but the two other measures need some explanation.

The processes governing the exit of a circuit from a meta-stable state are probabilistic, and one can represent this by a plot against time t of the probability that the circuit is in its meta-stable state at time t given that the circuit was in its meta-stable state at time 0. This distribution curve is observed experimentally to be exponential -- as one should expect since the probability density for leaving the meta-stable state at time t is likely to be independent of t. The time constant of this exponential is the measure τ.

The probability that a decision element enters its meta-stable state is a function of the temporal separation of the conflicting events; for example, the change of input signal and a clock pulse in the case of a synchronizer. A plot of this probability function would give comprehensive data on the likelihood of the decision element entering its meta-stable state. For most purposes, we may characterize this function by a single parameter called the conflict window. The conflict window W_c has dimension time and is equal to the area under the probability curve just described. The conflict window has the property that if the separation of conflicting events is distributed uniformly over the interval from -T/2 to T/2, the fraction of times the meta-stable state will result is W_c/T. If other parameters are fixed, the narrower the conflict window the better the decision element.

In terms of this characterization of decision elements, one can analyze the frequency of faulty operation of arbiter and synchronizer circuits. The quality of an arbiter or synchronizer is expressed by an error window W_e. The interpretation of the error window is that if the separation of conflicting events is distributed uniformly over the interval from -T/2 to T/2, then the probability of faulty operation is W_e/T.

We have compared the two synchronizer structures shown in Figure 1. The error window of the n-stage synchronizer is

$$W_e^n = W_c \times \left\{\frac{W_c}{\tau} e^{\delta_0/\tau}\right\}^{n-1} \times e^{-(n-1)\delta/\tau}$$

and the error window of a two-stage synchronizer having the same overall normal delay is

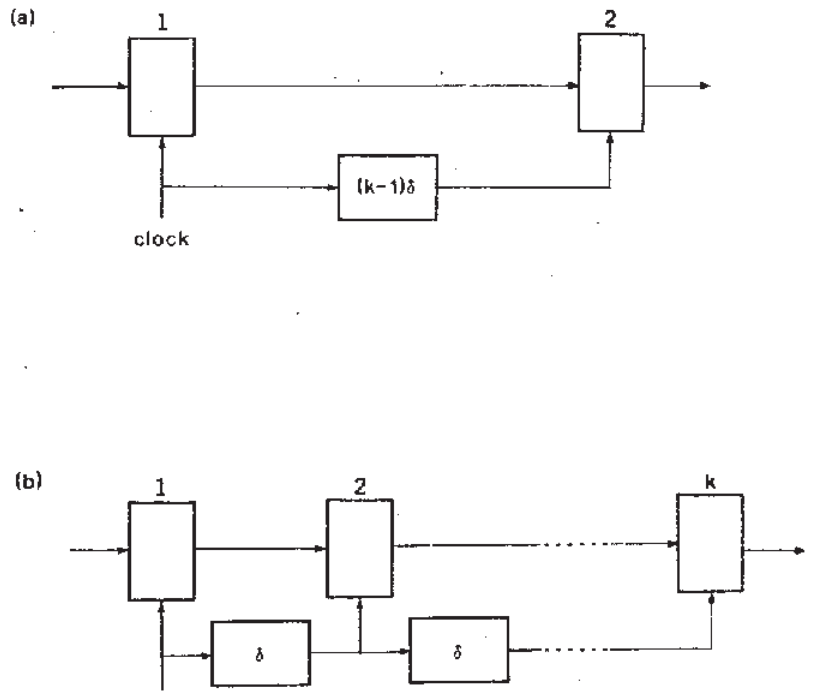


Figure 1. Comparison of synchronizer structures.

$$W_e^2 = W_c \times \left(\frac{W_c}{\tau} e^{\delta_0/\tau}\right) \times (e^{-\delta/\tau})^{n-1}$$

The ratio of the error windows of the two structures is

$$W_e^n / W_c^2 = \left(\frac{W_c}{\tau} e^{\delta_0/\tau}\right)^{n-2}$$

If the quantity $\mu = \tau/W_c\ e^{\delta_0/\tau})$ is greater than one, the n-stage arrangement will yield the smaller error window. Thus μ is a useful figure of merit for the decision elements of arbiters and synchronizers.

The requirements of an arbiter in an asynchronous speed independent system are different from those in a synchronous system because an asynchronous speed independent system can tolerate variations in delay. Therefore, to be called a perfect arbiter, an arbiter used in such systems need only resolve conflicts unambiguously one way or another in whatever time it needs to do so -- the arbiter is not required to resolve conflicts within a fixed length of time. From a practical viewpoint such an arbiter would be satisfactory if its average performance is very good. We have invented what we believe is the simplest circuit for a perfect asynchronous arbiter. The circuit, shown in Figure 2, consists of one flip-flop and two threshold NOT gates which connect the output of the flip-flop to the output terminals of the arbiter.

Initially both inputs of the arbiter are zero, and the outputs of the NAND gates are at one. If both inputs to the arbiter change to one simultaneously, causing the flip-flop to enter its meta-stable state, no signal change will appear at the arbiter output terminals until the flip-flop has definitely left the meta-stable state.

2. The Balance Property for Parallel Computations

In a sequential computation the passing of control from instruction to instruction may be represented by the movement of a "control token" in the flow chart of the program. In a parallel computation, many control tokens may be present in the flow chart indicating that many instructions are executable concurrently. Furthermore, a parallel computation is often organized as a collection of independent flow charts which communicate via message buffers. The message buffers may be modeled by the exchange of "message tokens" between pairs of separate flow charts. For a parallel computation to be free from the possibility of deadlock, there must be a kind of balance in the flow of control and message tokens during the computation.

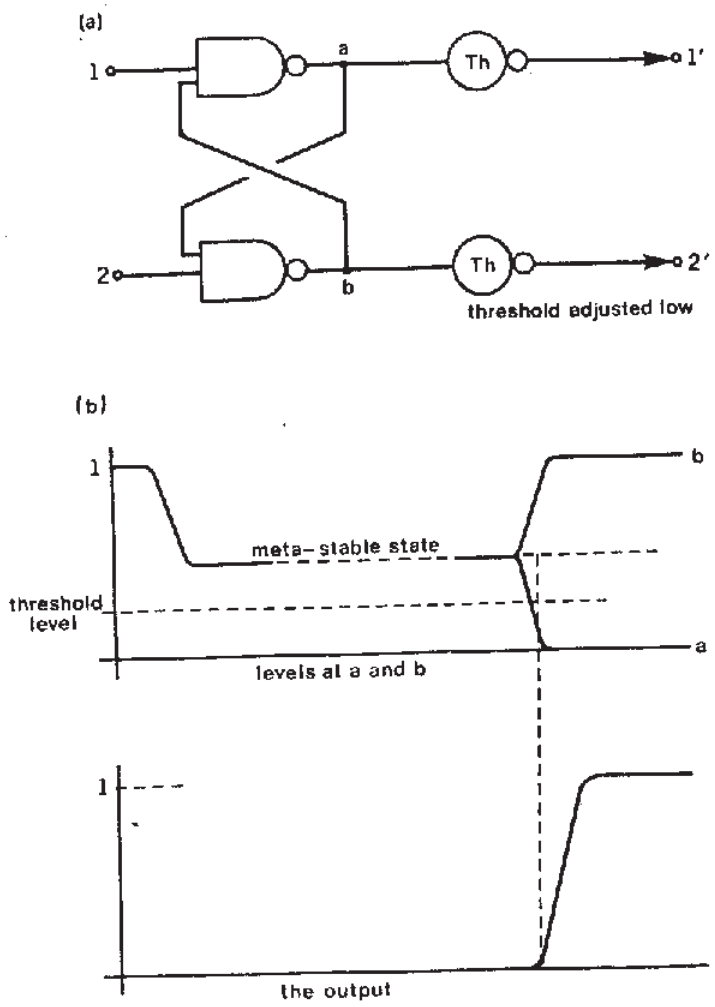


Figure 2. Realization of a perfect asynchronous arbiter.

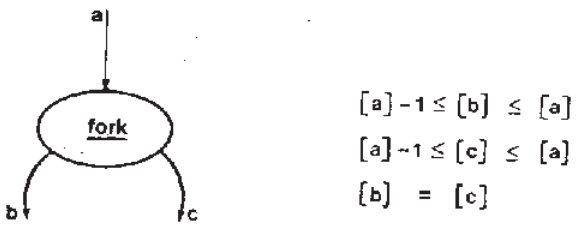


Figure 3. The fork primitive as a fixed module.

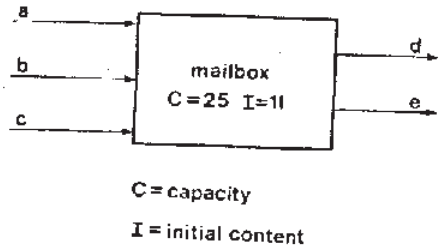
Bruce Lester and Suhas Patil have developed a general model for parallel computations in terms of which the notion of balance may be made precise, and have studied necessary conditions for liveness (freedom from deadlock) of parallel computations [55]. A parallel computation is represented by an interconnection of two types of modules -- fixed modules and [illegible] modules. These modules are characterized in terms of the flow of tokens at their terminals. If a is a terminal of a module, then [a] denotes the total number of tokens that have passed through the terminal since some arbitrarily chosen origin of time.

For a fixed module, the following condition must always be satisfied for each pair of terminals (a, b):

$$R_{ab} \cdot [a] - C_1 \leq [b] \leq R_{ab} \cdot [a] + C_2$$

where R_{ab}, C_1, C_2 are fixed positive constants. Figure 3 shows how the fork primitive of parallel programming may be represented as a fixed module. For each control token that enters at terminal a, one token is sent out at terminal b and one at terminal c. Since [a] and [b] must have a long-term ratio of 1:1, we must set $R_{ab} = 1$. Similarly $R_{ac} = 1$. If we assume that output and input of tokens by a fork module are distinct events, we must have $C_1 = 1$ and $C_2 = 0$, since [a] may be one greater than [b].

We say that an interconnection of fixed modules is balanced if the product of ratios R_{ab} around each simple circuit is exactly one. We have shown that any interconnection of fixed modules that is not balanced will become deadlocked.



Let $A=\{a,b,c\}$

$B=\{d,e\}$

Then

$[A]-14 \leq [B] \leq [A]+11$

Figure 4. A mailbox as a union module.

A union module behaves as a buffer or storehouse for tokens. A module is a union module if its terminals can be separated into disjoint sets A and B such that

$$[A] - C_1 \leq [B] \leq [A] + C_2, \text{ where } [A] \text{ denotes } \sum_{a \in A} [a].$$

The set A contains the input terminals and the set B contains the output terminals of the union module. The mailbox shown in Figure 4 is an example of a union module. The constant $C_2 = I = 11$ is the number of message tokens initially in the mailbox; $C_1 = C - I$ is the initial unused capacity of the mailbox.

The classes of fixed modules and union modules are closed under interconnection. Therefore, an interconnection of fixed and union modules is equivalent to a bipartite interconnection in which each union module touches only fixed modules. Let F_j be one of the fixed modules and let G_i be some union module in a bipartite interconnection. Let t_j be chosen as a reference terminal for module F_j. Then the total number of tokens transmitted from F_j to G_i may be written as

$$a_{ij}[t_j]$$

where

$$a_{ij} = \sum_{x \in X_{ij}} R_{t_j x} - \sum_{y \in Y_{ij}} R_{t_j y}$$

in which X_{ij} is the set of terminals of F_j that connect to input terminals of G_i and Y_{ij} is the set of terminals of F_j

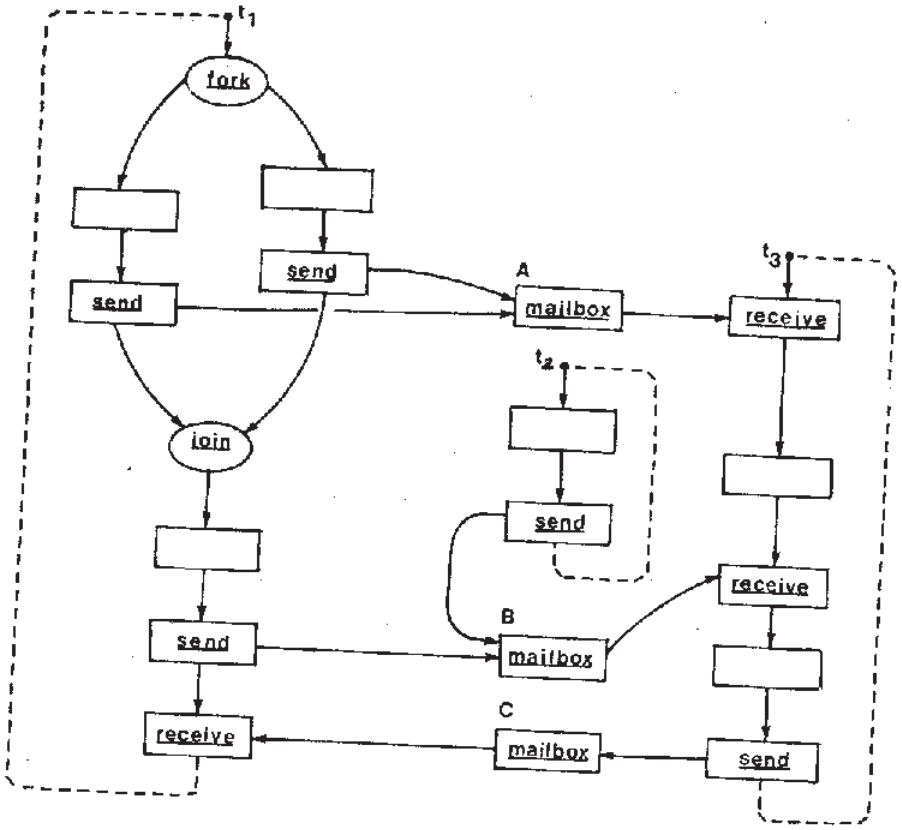


Figure 5. A computation without the balance property.

that connect to output terminals of G_i. Suppose there are n fixed modules and m union modules, let A be the m × n matrix of coefficients a_{ij}, and let $X = (x_1, \ldots, x_n)$, where x_j is interpreted as $[t_j]$. Then we say the interconnection of modules is balanced if the system $AX = 0$ has a solution with $X_i > 0$, $i = 1, \ldots, n$. Again, any interconnection that is not balanced will eventually become deadlocked, either because some union module runs out of required tokens, or because a union becomes filled to capacity and cannot receive more tokens.

Figure 5 illustrates an unbalanced computation. Each of the three flow charts is an interconnection of fixed modules and is therefore a fixed module by the closure property for fixed modules. Reference terminals for the flow charts are labeled t_1, t_2, t_3. Modules A, B and C act as finite capacity mailboxes and are therefore union modules. The conditions for balance are:

$$AX = \begin{bmatrix} 1+1 & -1 & 0 \\ 1 & -1 & 1 \\ -1 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2x_1-x_2 \\ x_1-x_2+x_3 \\ -x_1+x_2 \end{bmatrix}, \quad X>0$$

Since this system has no solution with $X > 0$, the computation in Figure 5 is unbalanced and will deadlock.

3. Analysis of Computation Rate

Chander Ramchandani has shown how computation rate analysis using timed marked graphs [56] can be applied to systems modeled by two more general classes of timed Petri nets [57, 58]. Consider the production system illustrated in Figure 6. The process performed at station 1 produces parts that are delivered alternately to stations 2 and 3 for use in fabricating two products A and B. We wish to determine the maximum rate of operation possible under the assumption that no limitation is imposed by the source of materials or the storing of products.

Figure 7 shows how this production system can be modeled as a timed Petri net. Each component of the production system that performs time-consuming actions is represented by a transition t_j having an associated firing time τ_i that represents the time required by the component to process one item. In a timed Petri net the firing of transition t comprises two events -- an initiation event at which one token is removed from each input place and a termination event at which one token is added to each output place. The termination event always occurs τ time units after the associated initiation event. A transition having no specified firing time is regarded as having a firing time of zero, that is, the initiation and termination events coincide. In Figure 7 transitions t_1, t_2 and t_3 correspond to processing stations 1, 2 and 3 in Figure 6.

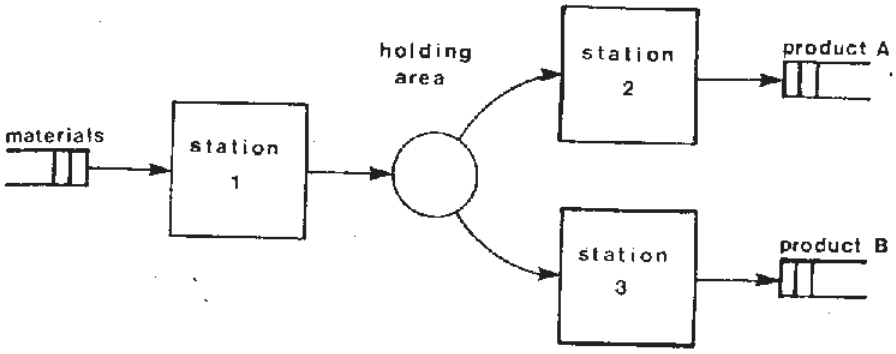


Figure 6

A production system

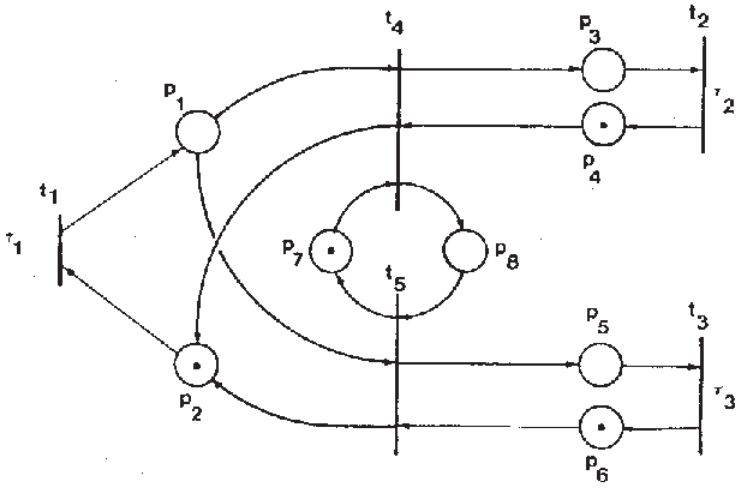


Figure 7

Timed Petri net

The token distribution shown in Figure 7 is an example of a marking of a Petri net that is live, safe and persistent. Liveness and safety are familiar properties of marked Petri nets; a marking is persistent if no firing sequence ever disables any transition except by firing it. For any such marked Petri net N, one can construct a marked graph G that mimics the behavior of N. In general the marked graph will have several transitions whose firings represent firings of some particular transition t of N; all of these transitions in G will be given the same label t. Figure 8 gives a marked graph that mimics the marked Petri net in Figure 7.

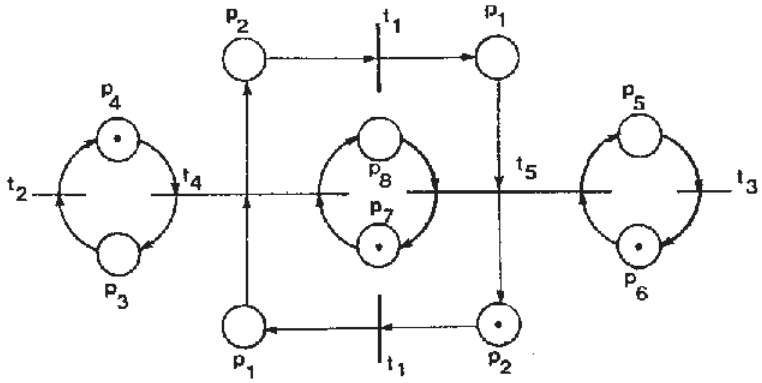


Figure 8

Multiply labelled marked graph derived from Figure 7

The computation rate ρ of a marked graph is bounded by the rate tokens can flow around each simple circuit C_k in the marked graph:

$$\rho \leq \frac{N_k}{T_k}, \quad k = 1, \ldots, m.$$

where N_k is the number of tokens on places in circuit C_k, and T_k is the sum of the firing times of the transitions of circuit C_k. Thus

$$\rho \leq \min \left\{ \frac{N_k}{T_k} \middle| k = 1, \ldots, m \right\}$$

This upper bound on computation rate is known to be achievable [58, 59].

In view of the correspondence of marked graphs to Petri nets illustrated by Figures 7 and 8, this result applies directly to determining the maximum computation rate of any timed Petri net with a live, safe and persistent marking. For example, suppose we have

$$\tau_1 = 2 \quad \tau_2 = 3 \quad \tau_3 = 5$$

in Figure 7. The maximum computation rate of the associated marked graph is

$$\rho = \min \left\{ \frac{1}{4} \quad \frac{1}{3} \quad \frac{1}{5} \right\} = \frac{1}{5}$$

and therefore the maximum firing rates for transitions t_1, t_2 and t_3 are

$$\rho_1 = \frac{2}{5} \quad \rho_2 = \rho_3 = \frac{1}{5}$$

This shows that processing station 3 is the bottleneck of the production system. Throughput of the production system could be increased by including several identical processing units at station 3. This is modeled by putting several tokens in place p_6 of the timed Petri net in Figure 7. Although this new marking is not safe, one can construct a modified net with a live, safe and persistent marking that has the same fastest schedule of initiation and termination events. Hence the same analysis method may be used. For example, if station 3 has two processing units, rates of

$$\rho_1 = \frac{1}{2} \quad \rho_2 = \frac{1}{4} \quad \rho_3 = \frac{1}{4}$$

for the three processing stations can be achieved.

Now reconsider the production system of Figure 6 where each part produced at station 1 is available to either station 2 or station 3. This situation is modeled by the Petri net of Figure 7 with places p_7 and p_8 deleted. The new Petri net (Figure 9a) may be regarded as composed of three simpler nets, as shown in Figure 9b, where each component net is a state machine in which each transition has exactly one input place and one output place. Such a Petri net is said to be state-machine decomposable. Hack has shown [60] that for a subclass of these Petri nets known as state-machine allocatable nets, one can associate a nonzero rate of token flow φ_i with each transition such that token flow is conserved at each place of the net. Any such consistent flow assignment characterizes some possible steady state mode of behavior of the Petri net.

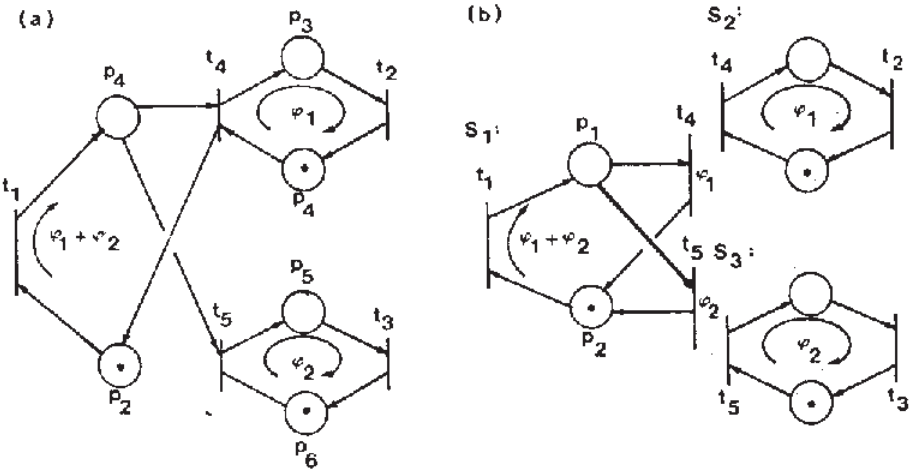


Figure 9

Petri net with a consistent flow assignment and its decomposition into state-machine components

Suppose $S_1, \ldots, S_m$ are the state-machine components of a state-machine allocatable net, and let N_k be the number of tokens on places of S_k in a specified initial marking of the net. We have shown that the computation rate ρ_j of each transition t_j must satisfy the condition

$$\rho_j \leq \varphi_j \rho$$

where

$$\rho = \min \left\{ \frac{N_k}{T_k} \middle| k = 1, \ldots, m \right\}$$

and

$$T_k = \sum \varphi_i \tau_i$$

where the sum is over all transitions t_i of state machine S_k.

For example, if we set

$$\varphi_1 = \frac{2}{3} \quad \varphi_2 = \frac{1}{3}$$

in the decomposed net of Figure 9b, we have

$$\rho = \min \left\{ \frac{1}{1 \cdot 2}, \frac{1}{\frac{2}{3} \cdot 3}, \frac{1}{\frac{1}{3} \cdot 5} \right\} = \frac{1}{2}$$

and therefore

$$\rho_1 \leq \frac{1}{2} \quad \rho_2 \leq \frac{1}{3} \quad \rho_3 \leq \frac{1}{6}$$

This means that no periodic firing schedule for the Petri net of Figure 9a can yield a higher computation rate while giving the specified distribution of token flow. In fact, an exact analysis shows that the maximum achievable computation rate is 1/7 or 6/7 of the bound. This exact value is found by considering each periodic pattern of activity of the state machine components that satisfies the flow requirement. Each such pattern corresponds to a marked graph from which an achievable computation rate may be determined. For our example, one marked graph that yields the best performance is shown in Figure 10. The circuit drawn with heavy lines determines the limiting rate of 1/7.

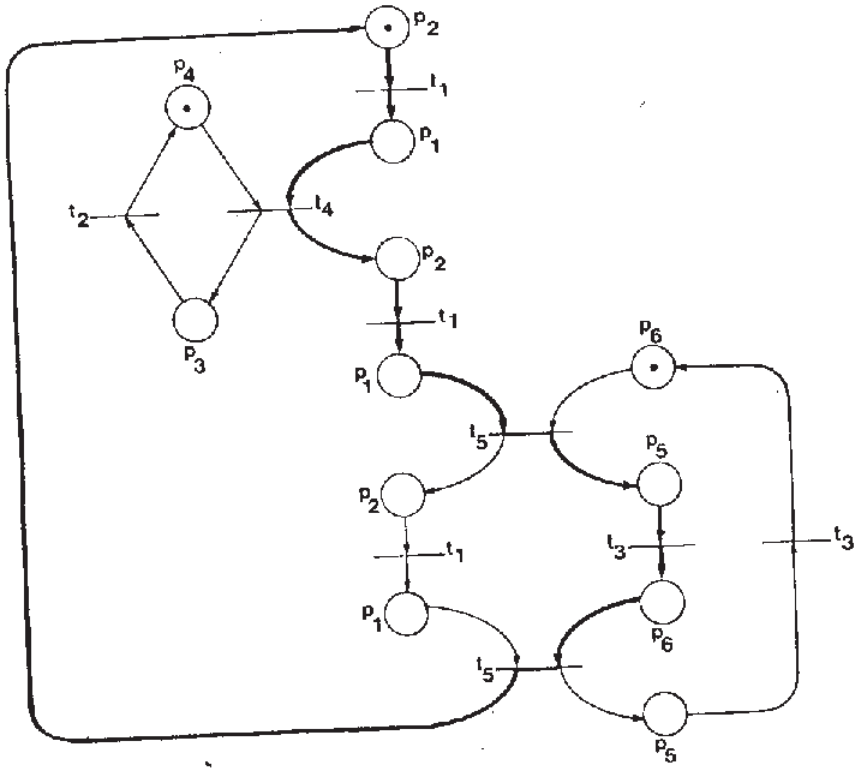


Figure 10

Marked graph for one periodic behavior of the net in Figure 9

4. Behavior-Preserving Transformations of Petri Nets

Petri nets represent the sequencing of events; therefore, if we define the language of a Petri net to be the set of sequences of events allowed by the net, then the equivalence problem of Petri net languages is to decide whether two given Petri nets have the same language. This equivalence problem is still open. Henry Baker [61] has discovered a large class of transformations of Petri nets which preserve their languages.

To study the equivalence of Petri nets it is useful to have simple characterizations of their associated languages. Figure 1 shows a Petri net represented as a bipartite directed graph in which each node is either a place of a transition. The initial marking of the net is shown by the presence of tokens on the places. Henry Baker and Michael Hack have found it convenient to study the behavior of Petri nets in terms of vectors and matrices whose elements are nonnegative integers. Let $N = \{0, 1, 2, \ldots\}$. We can represent a Petri net as

$$(P, T, F, B, M_0)$$

where

P is the set of places

T is the set of transitions $(P \cap T = \phi)$

$F: P \times T \to N$ is the forward incidence matrix of arcs:

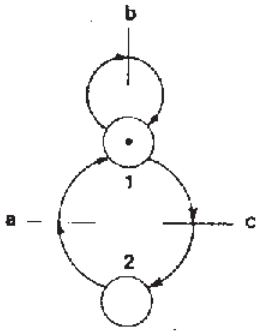
$$F_{ij} = \begin{cases} 1 & \text{if there is an arc from place } i \text{ to transition } j; \\ 0 & \text{otherwise} \end{cases}$$

$B: P \times T \to N$ is the backward incidence matrix of arcs:

$$B_{ij} = \begin{cases} 1 & \text{if there is an arc from transition } j \text{ to place } i; \\ 0 & \text{otherwise} \end{cases}$$

$M_0: P \to N$ is the initial marking vector:

$M_0(i)$ = number of tokens initially on place i



place: 1, 2

transitions: a, b, c,

Figure 11

A Petri net

For the Petri net in Figure 11, we have

$$P = \{1, 2\} \quad T = \{a, b, c\}$$

$$F = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad M_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

A <u>marking</u> of a Petri net is a function, or vector

$$M : P \to N$$

that associates a nonnegative integer M(p) with each place p. A transition t is enabled for marking M if

$$M_p \geq F_{pt}, \text{ each } p \in P.$$

The result of firing t is the new marking M' where

$$M'_p = M_p + B_{pt} - F_{pt}, \text{ each } p \in P$$

More generally we may consider markings that result from multiple firings of transitions. A k-fold firing of transition t is possible if

$$M_p \geq k\ F_{pt}, \text{ each } p \in P$$

or a <u>multiple firing</u> in which each transition t_i is fired k_i times, is possible if

$$M_p \geq \sum_i k_i F_{pt_i}, \text{ each } p \in P$$

If we represent a multiple firing by a $|T| \times 1$ vector X, where X_t is the number of times transition t is fired ($X_t = 0$ if t is not fired), then the Petri net firing rule may be expressed in matrix notation as:

Firing X is possible if and only if $M \geq F \cdot X$
The resulting marking is $M' = M + (B - F) \cdot X$

Here, the dot denotes matrix multiplication.

The matrix $\Delta = (B - F)$ is called the delta matrix of the Petri net since it expresses the change in marking for any firing. The firing of a single transition is denoted by a vector X in which the sole nonzero component is equal to one.

Let us write A : B to mean the matrix formed by adjoining the columns of B to the columns of A. Then a firing sequence σ is a matrix

$$X_1 : X_2 :\ldots: X_n$$

in which each column is some multiple firing of transitions. Similarly, a history H is a matrix

$$M_0 : M_1 :\ldots: M_n$$

in which the columns are the successive markings generated by some firing sequence. If history H is generated by the firing sequence σ, we must have

$$F \cdot X_i \leq M_{i-1}$$

if X_i is a possible firing for M_{i-1}. Therefore

$$F \cdot \sigma \leq H.$$

But each column M_j of H is the sum of the changes in marking due to all prior firings $X_1, \ldots, X_j$. We can write

$$H = M_0 : M_1 :\ldots: M_n$$

$$= M_0 \cdot \underline{1} + (B - F) \cdot \sigma \cdot \Sigma$$

where $\underline{1}$ is the n-component row vector of ones and Σ is an $n \times m$ progressive summing matrix

$$Z_{ij} = \begin{cases} 1 \text{ if } i < j \\ 0 \text{ otherwise} \end{cases}$$

Thus the conditions

$$\sigma \geq 0,$$

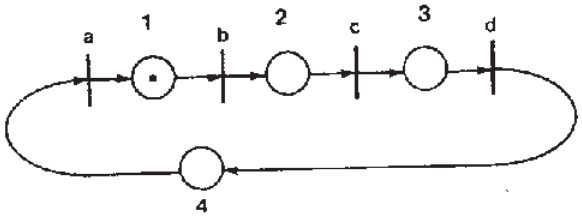
$$F \cdot \sigma \leq M_0 \cdot \underline{1} + (B - F) \cdot \sigma \cdot \Sigma$$

hold if and only if an integer matrix σ is a firing sequence of the net. Hence these conditions completely characterize the language of the Petri net.

Transformation of a Petri net with places P into a net with places P' corresponds to premultiplying the matrices F, B and M_0 by a $|P'|\times|P|$ transformation matrix. So long as this transform matrix is integer and nonnegative, the set of firing sequences of the new Petri net will include the set of firing sequences of the old Petri net.

As an illustration, the Petri net in Figure 12a is transformed into the net of Figure 12c by the transformation matrix Z. Each firing sequence of the original net is also a firing sequence of the transformed net.

(a) original net



$P = \{1, 2, 3, 4\}$

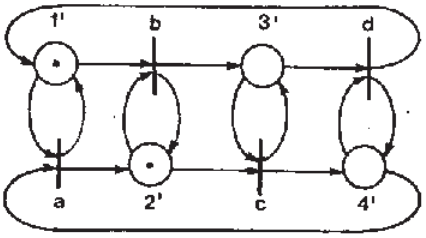
$T = \{a, b, c, d\}$

$$F = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad M_0 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

(b) transformation matrix

$$Z = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \end{array}$$

(c) transformed net



$$M'_0 = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} = Z \cdot M_0$$

$$F' = \begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \end{pmatrix} = Z \cdot F \qquad B' = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} = Z \cdot B$$

Figure 12. Example of a behavior-preserving transformation.

5. Reduction of Generalized Petri Nets

A generalized Petri net has places and transitions, as does an ordinary Petri net, but each place/transition pair may have any number of forward and backward arcs, as illustrated in Figure 13.

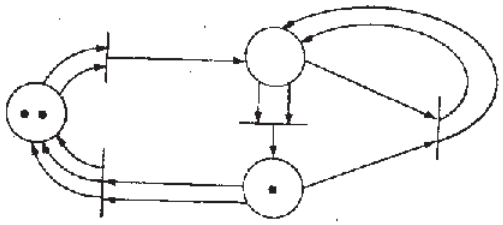


Figure 13

A generalized Petri net

Firing a transition in such a net claims one token from a place for each forward arc joining the place to the transition, and adds one token to a place for each backward arc joining the place to the transition. Mike Hack has shown how any generalized Petri net may be converted into an equivalent ordinary net, where equivalence means, as above, that both nets permit the same sequences of transition firing.

Since there is a direct correspondence between generalized Petri nets and the vector replacement system studied by Keller [62], this result shows that each vector replacement system corresponds to a vector addition system in which each vector component is either +1, 0, or -1.

Figure 14 illustrates the transformation, which is done by performing the following steps for each place p of any generalized net P to obtain an ordinary net P' in which no transition has backward and forward arcs to the same place:

1. Let k be the maximum number of arcs in P connecting place p with any transition.

2. Replace place p with a ring of k places interconnected transitions labeled λ. (The firings of λ-transitions are to be ignored in comparing the behavior of two nets.)

3. Assign the arcs originally connected to place p to places in the ring so that each arc going to or from a transition connects to a distinct place in the ring.

4. Place tokens on places of the ring so that the total over all places in the ring equals the original marking of place p.

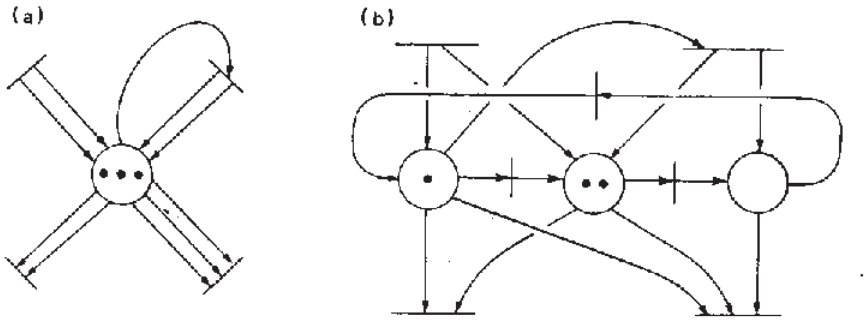


Figure 14

Reducing a generalized Petri net

No matter how the transformation is carried out, the tokens can always arrange themselves in the ring through firing of the λ-transitions so any firing sequence of P is, ignoring the firing of λ-transitions, also a firing sequence of P'. Since the converse is also true, P and P' are equivalent.

6. A Counterexample

We have been able to construct a counterexample to two conjectures which, until now, had not been seriously challenged:

1. Is it true that if a Petri net has a live, bounded marking class, then every marking class is bounded?

2. Is it true that if a Petri net has a live marking, then it has live markings with arbitrarily many tokens?

For most "reasonable" Petri nets both implications are true. However, the generalized Petri net in Figure 15 has exactly five live, bounded markings, and all other markings are either unbounded and non-live, or completely dead for lack of sufficient tokens.

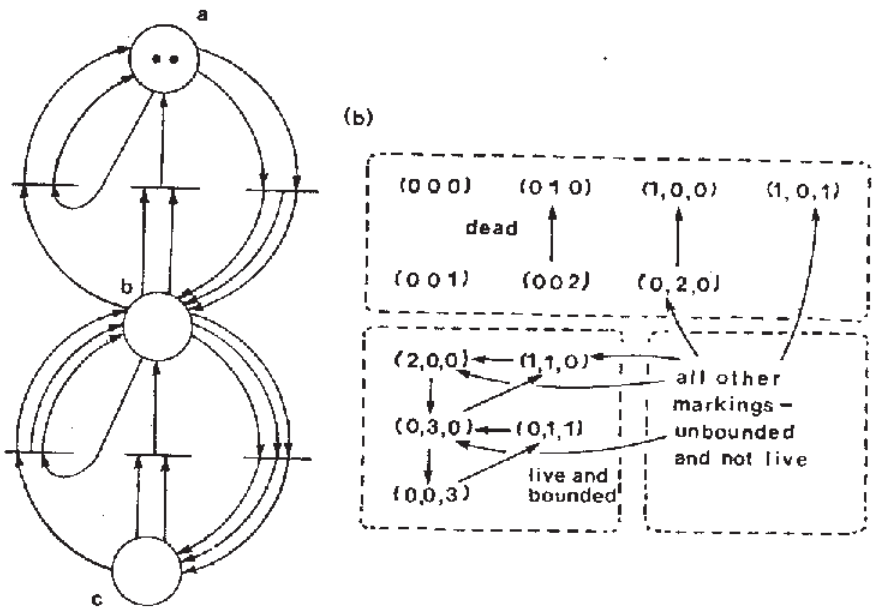


Figure 15

A counter example to two conjectures about Petri nets

7. Semantic Foundations for Procedures and Data Structures

We have set for ourselves the goal of developing a "common base language" for procedures and data structures. This base language must be able to express the important semantic constructs on which the correctness of application programs depends -- including aspects, such as concurrency, data bases, and security, which often fall outside the domain of the language designer. Through this research we expect to gain a better understanding of how computer systems may be specified and built to give more direct support to the semantic requirements of user programs.

Some early ideas appeared in [63], and a possible rudimentary form for the base language was presented in [64]. The work begun in [64] on the translation of block-structured language was reviewed in [65]. This work has been expanded in the master's thesis of Nimal Amerasinghe [66] which gives precise rules for translating programs expressed in a representative block-structured language into a slight extension of the base language of [64]. Although the base language does not directly support the notions of nonlocal identifiers or closures of procedures, the translation rules correctly handle procedure values and nonlocal assignments. In addition, studies in progress concern the representation of procedures involving values that are "references" to other values, and the choice of base language semantic structures for parallel computation.

8. Computation Schemas

Study of the schemes of programs is of interest for the insight they provide regarding the choice of fundamental semantic constructs for the representation of algorithms and data structures. In particular, the current work on a data-flow formulation for the base language stems from an investigation of the formal properties of "data-flow schemas" [67]. This study led to the discovery of a procedure for deciding equivalence of any pair of output variables in any free program schema using only nested *if* ... *then* ... *else* ... and *while* ... *do* ... control structures.

John Linderman has completed his study [68] of necessary conditions for the functionality of the input-output mappings defined by parallel program schemas with distinguished input and output cells. This work is based on a formal model which is at once a generalization and a simplification of the parallel program schemas of Karp and Miller [69], and was described in last year's report [70]. In particular, sets of operators or deciders may bear the same function letter, indicating that they must have the same associated function or predicate in any interpretation. Linderman defines an operator or decider to be *productive* if there exists an interpretation such that the output values resulting from some computation depend on the actions of the operator or decider. The main result is that under frequently imposed conditions (e.g., repetition freeness and persistence), absence of conflict between operators at the memory cells is a necessary condition as well as a sufficient condition for functionality whenever each operator and decider

of a schema is productive.

Dennis Kfoury has studied the question of whether the known decision procedures of first-order logic could have important application to the theory of program schemas developed by Luckham, Park and Paterson [71]. One question is whether, for any given schema S, there exist first-order conditions such that, for any interpretation satisfying the conditions, schema S defines a total function [72]. Since any program schema without iteration defines a total function, the answer to the question is "yes" for these schemas. In general, we have shown that, if the answer is "yes" for some schema S, then S is equivalent to some p ogram schema without iteration. Thus, constraining interpretations by first-order conditions cannot ensure totality of a program schema without also making the iterations in the schema superfluous.

9. Semantics of Data Base Systems

Igor Hawryszkiewycz has completed a thesis [73] in which xx the semantics of data base systems is studied in relation to an underlying semantic model derived from our base language effort. Two objectives were sought in this work: The first was to develop appropriate semantics for data base systems where several users of the system are able to share access to data bases, and concurrent actions by the system on behalf of its users are allowed. The second objective was to show how concurrency of user actions may best be exploited in an implementation. The approach used was to develop a precise correspondence between the execution of commands of the data base system and sequences of actions in the underlying semantic model.

For this study, Codd's relational model [74, 75] was adopted as the starting point for the semantics of data base systems. A data base consists of relations defined on n-tuples of domains. If $d_1, \ldots, d_n$ are domains, then an n-ary relation on these domains is a set of n-tuples

$$\{e_1, \ldots, e_n\}$$

where each e_i is an element of domain d_i.

As an example of a relation consider the order information displayed below:

item	description	quantity
shoe	black	2
shoe	brown	4
coat	brown	1

As a relation this information involves three domains:

```
article = {'coat', 'hat', 'shoe'}
color   = {'red', 'brown', 'black'}
integer = {0, 1, 2, ...}
```

We shall illustrate the commands of our data base system by showing how a relation representing this information would be entered into the system. For the moment, suppose the system knows the three domains listed above. The first step is to create and name a new relation variable. As in the displayed information, it is valuable to let the data base designer choose names for the fields of a relation independent of the names given to the domains. Therefore, the command to create the relation variable is

```
declare relation (order, ((item, article), (description,
    color), (quantity, integer)))
```

This command states that the relation called 'order' will have three fields where, for example, the first field is named 'item' and in each member or instance x of the relation, its first component will be called the value in the 'item'-field of x and will be an element of domain 'article'. After execution of this command, the value of relation variable 'order' is the empty set. Instances are added to the relation value by use of the command add ri (for "add relation instance"):

```
add ri (order (description, shoes), (color, black),
       (quantity, 4))

add ri (order (description, shoes), (color, brown),
       (quantity, 2))

add ri (order (description, coat), (color, brown),
       (quantity, 1))
```

The data base system includes commands to delete specific instances, and to retrieve instances of a relation having specified domain elements in certain fields.

Domains are established by first creating a unary relation:

```
declare relation (article, string)
```

The format of this command is simpler since no field names are needed for a unary relation. Executing the command creates a relation variable named 'article' and declares that the instances of its relation value will be elements of the natural domain string, which contains all strings constructed from some standard alphabet of symbols. The natural domains of strings and of integers are permanently accessible to all users of the data base system. The second step is to enter the desired elements:

add ri (article, shoe)
add ri (article, coat)
add ri (article, hat)

Finally, the relation is made to be the value of a newly declared domain:

declare domain (article, article)

For investigating issues of sharing and concurrency in implementations of data base systems we need a semantic model in which all access paths to information in the data base may be represented explicitly and directly. In addition, the model should be machine independent so it is applicable over a variety of computer systems; yet the model must have a reasonably straightforward relationship to practical computer systems through such implementation tools as hierarchical directories and hash-coding techniques.

We decided to use the directed acyclic graphs adopted as the fundamental data structures for the base language development. Thus domain values and relation values are represented by directed acyclic graphs and the representations are designed so directed paths in the graphs correspond to access paths in the data bases.

The representation of a relation value is illustrated in Figure 16 and has three parts: The domain value structure links the representation of the relation value to the graphs that represent the domain values associated with each field of the relation. The description structure contains data indicating how the presence of relation instances is represented in the association structure, and, in particular, which modes of access to the relation instances are directly represented by paths.

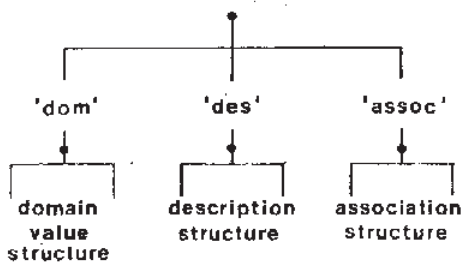


Figure 16

Parts of an abstract relation value

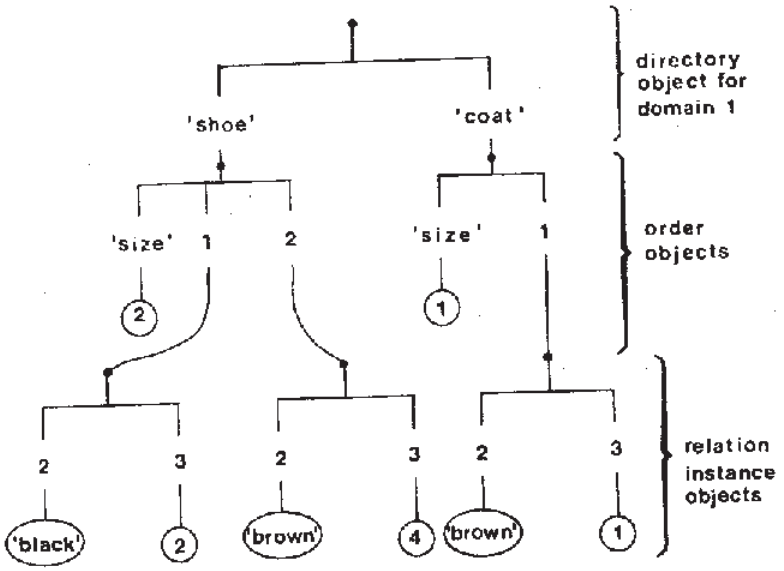


Figure 17

Association structure for one mode of access

Figure 17 shows how the 'order' relation is represented by an association structure, where only access according to the value in the first field of the relation is required. The association structure has one component for each element of the domain 'article' that occurs in some instance of the relation. This structure is called a directory object since it directs access to a subset of relation instances having specified values in certain fields -- in this case a specified element in the first field. Since there may be many relation instances satisfying the specifications, each component of a directory object is an order object that gives access to the relation instances according to some arbitrary ordering of them.

The components of each order object are relation instance objects that contain the domain elements of the instance accessed by field number. In this example, the domain elements are given only for fields 2 and 3 since the only access to the relation is by means of field 1, hence the field 1 value is known to the computation.

Figure 18 illustrates the form of the association structure when several access paths are provided -- by field 1 or by field 2 in this example. In this structure each order object has as its components all relation instances that have the same value in field 1 and the same value in field 2. Since

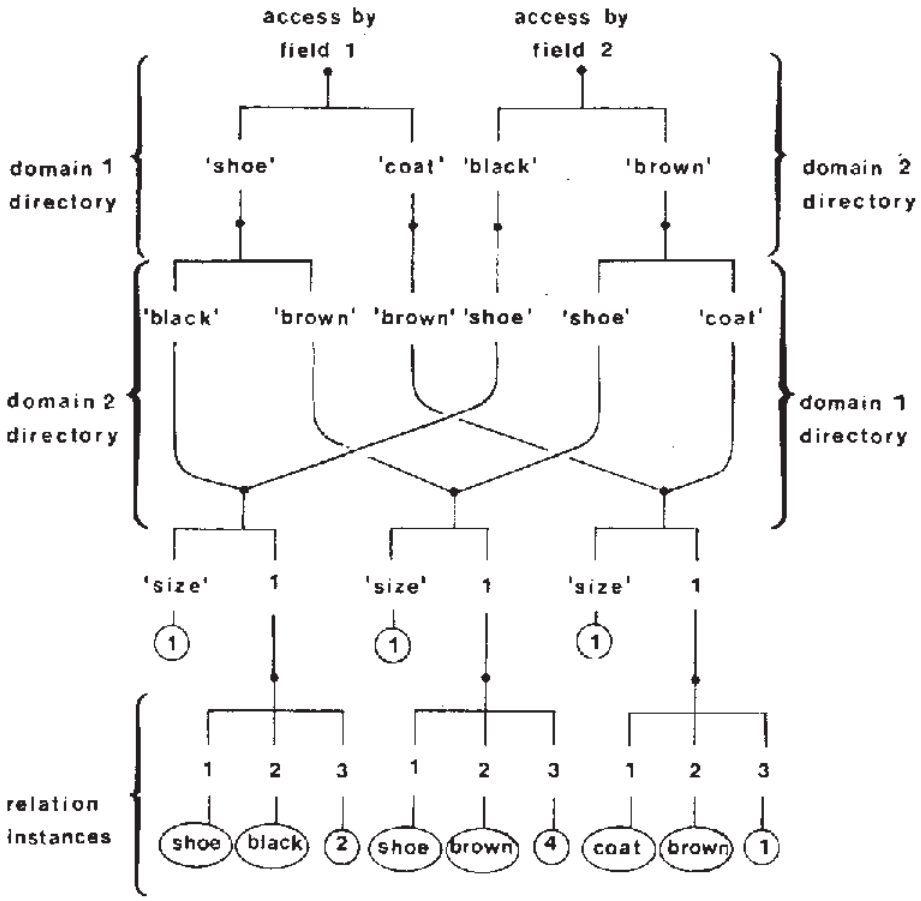


Figure 18

Association structure for two access modes

there is at most one instance in the relation 'order' for each combination of values, the order objects have one component each. The two levels of directory objects provide for access to the order objects by making the first access by field 1 or by field 2.

The data base model includes the commands share domain and share relation through which a user may make any of his domain values and relation variables accessible to designated users of the system. The commands borrow domain and borrow relation are given by the borrowing user to name and establish access to a borrowed domain or relation. When a relation is borrowed, both users get the ability to access the relation

using commands to add, delete and retrieve relation instances. When a domain is borrowed, subsequent changes to the domain value made by the owner are not permitted to affect the value seen by the borrower.

In the relational data base model, the meaning of each command is specified in terms of the changes made to set theoretic components of the data base state. For the study of implementation issues of access paths and concurrency, each command is also expressed as an algorithmic procedure that performs a corresponding transformation of the abstract state. Let s_2 denote the data base state resulting from giving command c with the data base system in state s_1:

$$s_1 \xrightarrow{c} s_2$$

The corresponding transformation of the abstract state is the series of transformations making up execution of the semantic procedure for command c:

$$a_1 \longrightarrow a_2 \longrightarrow \ldots \longrightarrow a_n$$

where abstract states a_1 and a_n represent s_1 and s_2, respectively.

Suppose commands c_1 and c_2 are given concurrently by two users of the data base system. Since the commands may refer to the same relation value the resulting data base state may depend on the order in which the corresponding transformations areaapplied to the data base state:

$$s_0 \xrightarrow{c_1} \xrightarrow{c_2} s_1 \qquad s_0 \xrightarrow{c_2} \xrightarrow{c_1} s_2$$

States s_1 and s_2 both represent correct responses by the relational model. In the abstract model, correctness of the semantic procedures means that concurrent execution of the semantic procedures for commands c_1 and c_2 is guaranteed to yield a final abstract state that corresponds to one of the two valid data base states.

We illustrate the methods used in constructing the abstract model by presenting one of the simpler semantic procedures, the procedure for the command

declare domain (n-1, n-2)

As discussed earlier, this command creates a new domain variable for the user, gives it the name 'n-1' and gives it a value equal to the current value of the unary relation named 'n-2'. To define the command in terms of transformation of the relational data base state, we must identify the relevant components of the state:

U - set of user names

D - set of domain variables

R - set of relation variables

R_1 - set of unary relation variables ($R_1 \subseteq R$).

W_1 - set of unary relation values ($w | w \subseteq$ strings $\cup$ integers)

N_D - set of domain names

N_R - set of relation names

A_D: $(U \times N_D) \rightarrow D$ - domain access function:

$(('u', 'n'), d) \in A_D$ means that user 'u' has access to domain variable r by the name 'n'

A_R: $(U \times N_R) \rightarrow R$ - relation access function;

$(('u', 'n'), r) \in A_R$ means that user 'u' has access to relation variable r by the name 'n'

►

V_D: $D \rightarrow W_1$ - domain value function: associates a unary relation value with each domain variable

V_R: $R \rightarrow W$ - relation value function: associates a relation value with each relation variable

The relational model definition of declare domain is given in Figure 19 where 'u' is the name of the user giving the command.

The semantic procedure for declare domain is given in Figure 20. The statement principal → p in line 2 assigns to variable p the name of the user for whom the procedure is being performed. The variables P, DOM, REL are pointer variables whose values identify specific nodes in the abstract state. In line 3 of the procedure, the dots denote selection of a component of the abstract data structure denoted by the expression to the left. Here P refers to the root node of the abstract state. If p has the value 'u', the value of P • p is a pointer to the 'u'-component of the state, and DOM is assigned a pointer to the 'domain'-component of this data structure, as shown in Figure 21. The primitive predicate selt (DOM, a2) tests whether the structure to which DOM refers has a component having the value of variable a2 as its selector. The transformation performed by this semantic procedure is illustrated in Figure 21 and results in the insertion of the two branches enclosed by the dashed boundary.

```
declare domain ('n-1', 'n-2'):
   if A_D(<'u', 'n-1'>) is undefined
       and A_R(<'u', 'n-2'>) = r
       and r ∈ R_1
   then effective completion with
       D' = D ∪ {d}, where d is a new domain variable
       S'_D = A_D ∪ {<<'u', 'n-1'>, d>}
       V'_D = V_D ∪ {<d, V_R(r)>}
   else ineffective completion
```

Figure 19. Relational definition of the command declare domain.

```
1.  procedure declare domain (a1, a2);
2.  principal → p;
3.  P • p • 'domains' → DOM;
4.  P • p • 'relations' → REL;
5.  if selt (DOM, a1) = false
6.       and selt (REL, a2) = true
7.       and val (REL • a2 • 'access' • 'size') = 1
8.  then begin link (DOM • a1, REL • a2 • 'value', 'value');
9.       return true; end
10. else return false;
11. end
```

Figure 20. Semantic procedure for the declare domain command.

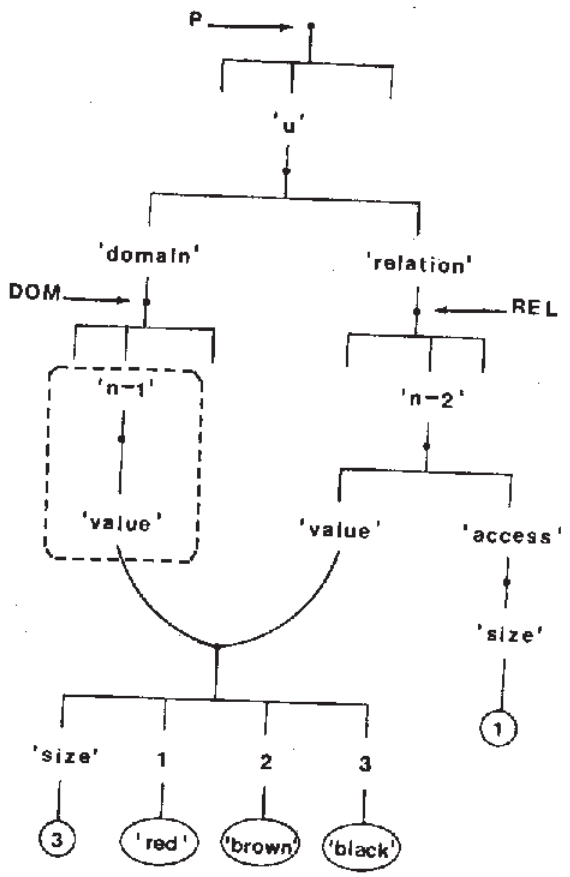


Figure 21

Whenever a relation of the data base is accessible to more than one user, concurrent requests to add, delete, and retrieve relation instances for the same relation variable may occur. For this general case, the semantic procedures are considered correct if and only if the abstract state resulting from each possible computation by the semantic procedures is a consistent representation of the new data base state defined by some merging of the corresponding user command sequences. Igor Hawryszkiewycz has shown how the semantic procedures may be augmented so that correct concurrent execution of commands is realized [73]. xx Furthermore, the procedures are designed so that actions on behalf of distinct users can proceed independently except

where arbitrary sequencing of user computations would allow inconsistent abstract states to result. This is accomplished by establishing a queue for suspended computations at a node of the abstract state whenever some user computation must wait for another computation to complete its work on the structure associated with the node. When a computation completes its access to a structure, it awakens some process in the associated queue if there is a queue and it is nonempty. In this way, add commands and delete commands for the same relation variable interfere with each other only if their access paths intersect in the association structure.

From this work, we can conclude that any implementation of the abstract model provides a correct realization of the relational data base system.

10. Modular and Structured Programming

We are studying the implications of the concepts of modular programming and structured programming for the design of programming languages and computer systems. The goal of this research is to understand just what program and data structures are desirable for building correct and reliable programs. We intend to apply the knowledge gained from this effort to the development of the base language. Work is underway on three interrelated subjects: the design of a formal model of a computer system that meets all requirements for modular programming; the design of a programming language with appropriate facilities for structured programming; the development of a formalism for expressing desired properties of new data types and operations in a representation-independent manner.

11. Modular Programming

An important property of a programming system or language is the ability to support modular programming: one should be able to build new program modules by combining modules developed by other programmers without knowledge of their internal design [76]. A computer system that supports modular programming should ensure that program modules may be readily combined and that their behavior is predictable and independent of their context of use. A computer or programming system that meets these goals is said to have the property of modularity.

D. Austin Henderson, Jr. is studying a model of a computation system in which modularity can be achieved. One aspect of programming languages which can lead to unpredictable behavior is the use of free variables. A free variable of a program is an identifier in the program which derives its meaning from the meaning of that identifier in the context in which the program is used. For example, the Algol "non-local variable" takes the meaning it has in the enclosing block.

Consider a procedure that implements a linear transformation on integers:

```
procedure f(x); integer x;
    begin
        integer y;
        y := a × x + b
        f := y
        return
    end;
```

Notice that the behavior of procedure f is unpredictable because it depends on the values of a and b in the procedure within which f appears.

One means of achieving predictable behavior is to demand that all free variables be added to the parameter list; the example becomes

```
procedure f(x, a, b); integer x, a, b;
    begin
        integer y;
        integer :=a × x + b
        f := y
        return
    end
```

This solution has the drawback that procedure f cannot be viewed as an operator implementing a single linear transformation. To achieve that effect, the particular values of a and b must be passed around with f to all programs wishing to use the linear transformation operator. (In more complicated cases, the information contained in free variables may be sensitive, and the "definer" may wish to make it inaccessible to users.)

The solution chosen in our model is to substitute a notion of functional for the conventional notion of procedure, and introduce a mechanism for incremental binding. To illustrate, let us follow the creation of a linear transformation operator in the model.

We start with a piece of code:

```
1.  {x, a, b|
2.      declare ('y', a × x + b)
3.      result is (y)
4.  }
```

In line 1 of this code, identifiers x, a, and b are specified

as being external (they precede the '|' symbol). This means that the values associated with these identifiers will be determined outside the operator. In line 2 of the code identifier y is declared to be local to the code and is given the value denoted by the expression "a × x + b".

The code is introduced into the system as a string constant (let us call it c). This string is used to create a functional which is the result of applying the primitive install to c:

```
ℓt = install (c)
```

The value of install (c) is a functional in which no values are associated with any of its external identifiers. We picture such a functional as in Figure 22. The stub to the right represents the binding of external identifiers and shows, in this case, that none have associated values.

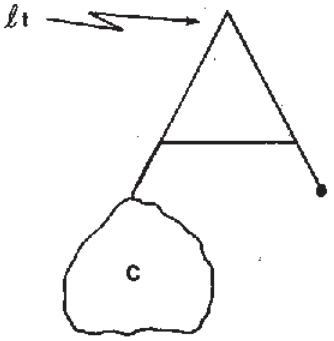


Figure 22

A functional

To create a functional which implements a particular linear transformation, values are "bound" to the externals of ℓt using the primitive operator bind:

```
t = bind (ℓt, 'a', 3)
ℓ1 = bind (t, 'b', -2)
```

We use t and ℓ1 to denote the values produced by bind at each stage: as in Figure 23 they are functionals with one and two of their externals bound. Notice that ℓt has not been altered by bind. Consequently it may be used again to produce different linear transformations. For example

```
ℓ2 = bind (bind(ℓt, 'b', 3), 'a', -14)
```

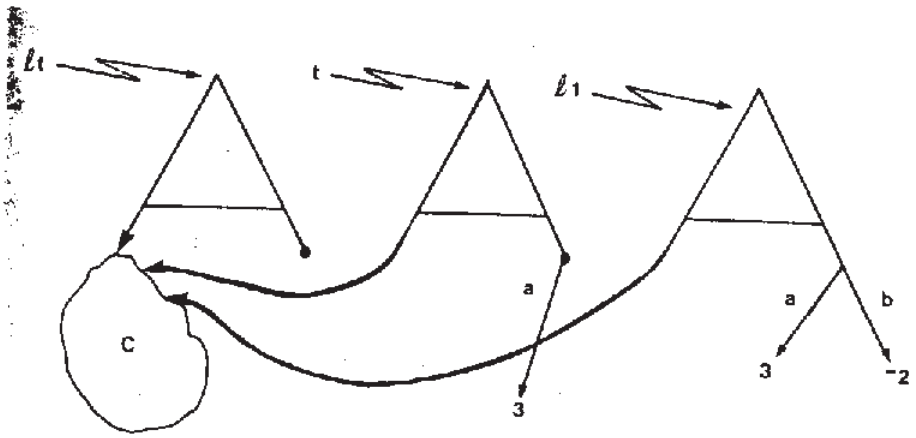


Figure 23

Incremental binding

Functionals ℓ1 and ℓ2 may be used in any context without loss of meaning. Each has predictable behavior.

To use either of these operators, the external identifier x must be bound, and the resulting functional evaluated:

```
u = bind (ℓ1, 'x', 72)
r = eval (u)
```

Evaluation of a functional is done by creating an activation. An activation is the bundle of information necessary to carry out the actions specified in the functional.

Figure 24 shows the two pieces of an activation of the functional r: the C-component is the code of the functional; the E-component holds the values bound to the external identifiers. As identifiers are encountered during evaluation of the functional they are replaced by their values as specified by the E-component of the activation. The expression "a × x + b" yields 214. Hence "local" identifier y is associated with this value (Figure 25). The statement result is y indicates the completion of evaluation and specifies the value that is the result of the functional. Thus eval(r) denotes the value 214.

Incremental binding is a modular alternative to the use of free variables. In the example given here all values used were constants. Another source of unpredictable behavior is introduced if "variable" values (values with memory, or state) are permitted. It would, therefore, be nice to exclude them, but unfortunately that restriction would greatly reduce the utility of our model. Fortunately, large classes of predictably-behaving operators can be identified even when variable values are permitted. Another source of unpredictability is concurrent computation using shared values. It appears that this source can also be controlled.

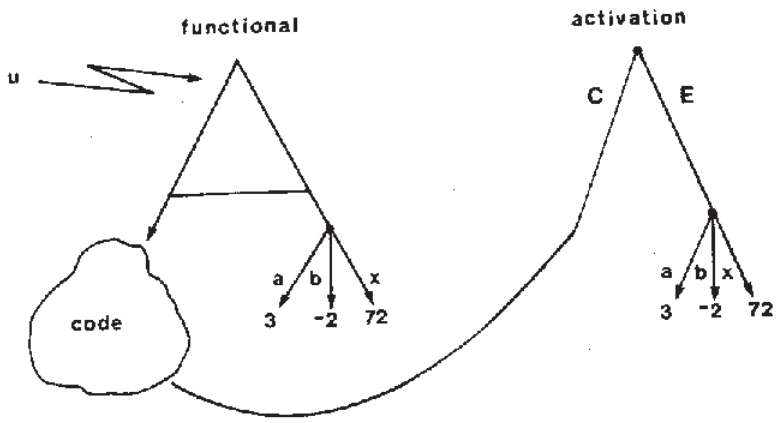


Figure 24

Components of an activation

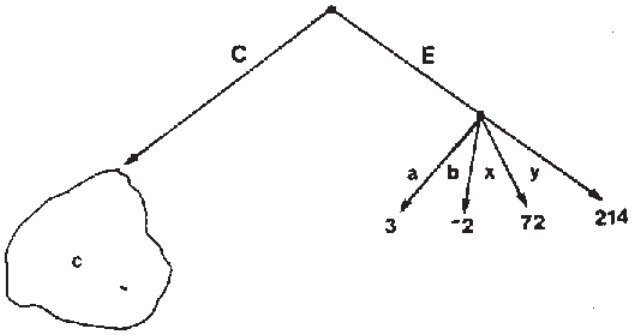


Figure 25

Result of evaluation

The model is being used to study these sources of unpredictability. Some concepts adequate to control the unpredictability arising from these sources, and means of expressing those concepts in the model are being isolated. The model with these expressive tools included will permit the construction of predictable operators, thus aiding in the achievement of modularity.

12. Structured Programming

Structured programming is a programming discipline intended to support the production of correct, understandable programs which are easy to modify and maintain. While modular programming addresses the problem of how to facilitate one programmer's use of another programmer's program, structured programming addresses the problem of building programs. To support the production of reliable software, a system supporting modular programming should be designed to meet the requirements of structured programming. Our research on structured programming is directed toward discovering the nature of these requirements and how they may be met in the design of programming language and systems.

The term "structured programming" has been applied to many different coding and design techniques; three different interpretations may be distinguished in the literature:

M1) Structured programming is goto-free programming [77].

M2) Structured programming is top-down programming (control only) [78].

M3) Structured programming is top-down programming [79, 80].

Only M1 and M2 are really well defined. The third interpretation M3 is not supported by any existing programming language, as Dijkstra pointed out in his Turing lecture [81]. For one x thing, M3 is really a philosophy about how programming ought to be done -- an issue not yet resolved. For another, those parts of M3 which are understood appear to require new programming language constructs for their support, and many programming language issues must be rethought (block structure, for example [82]). The three interpretations are related:

$$M1 \subset M2 \subset M3$$

in the sense of containing ideas or concepts; the reasons M1 is necessary in each case are discussed in [83].x

Top-down programming (both M2 and M3) involves the following: The first code written is the very "top" of the system or program; it describes the relationship among the major functional components of the program. This code constitutes a structured program segment. Each component will eventually be described by a structured program segment, and is referenced in the top segment by its segment name. The

code of each segment is itself expressed in a very limited repertoire of basic control structures. The only structures permitted are: concatenation, execution of a statement based on the testing of a condition, and iteration. Selection of the next statement by a goto is not permitted.

```
begin
integer relation;
boolean must_scan;
string symbol;
stack parse_stack;
must_scan := true;
'push'(parse_stack, eof_entry);
while not 'finished'(parse_stack) do
     begin
     if must_scan then symbol :='scan_next_symbol'( );
     relation := 'precedence_relation'('top'(parse_stack),symbol);
     'perform_operation_based_on_relation'(relation, parse_stack,
                                          symbol, must_scan);

     end
end
```

Figure 26. An example of a structured program segment. The segment names appear in quotes.

Figure 26 is an example of a structured program segment which is the "top" of an operator precedence compiler. The example, written in an Algol-like notation, uses many segment names including 'scan_next_symbol', 'precedence_relation', etc. The segment shown is a complete description of the compiler in the sense that if we had a machine with all the segment names as primitives, it would run. However, such a machine is unlikely to exist, so the next step is to select a segment name and to code the segment which explains it in terms of other segment names. The process will continue until all segments have been defined by code.

So far, the description and example could fit either M2 or M3. The difference between them may be illustrated by considering the segment name 'push' and the data type stack. The two are obviously related: stack is a set of abstract data objects to be operated on by the abstract operator 'push'. The M2 approach is concerned only with control; it ignores the problem of how to handle a data type such as stack and instead permits ad hoc solutions (for example, stack could be defined on the spot in terms of the data type array). In the M3 interpretation of structured programming the code of a segment must not have any implication on the choice of representation for abstract data types such as stack. Furthermore, only segments that define operations on a data type (such as 'push', 'finished', and 'top' in the example) should have access to representations of objects of the type. These ideas are not supported by existing programming languages.

The requirements of M3 interpretation may be summed up in two criteria:

C1) It must be possible to make use of an abstract data type without specifying its representation.

C2) Only those segments which perform meaningful operations on an abstract data type should have knowledge of the way objects of the data type are represented.

These criteria are being used to guide an investigation into the adequacy of some existing languages to support M3 structured programming. The languages being considered in this study are PL/I, EL [84], Pascal [85], and Simula 67 [86, 87].

13. A Structured Programming Language

B. Liskov is designing a language to support and encourage M3 structured programming. This language will be based on the concept of a function cluster; the concept was developed as the result of the analysis of the need for abstract data types in a structured programming language and satisfies the two abstract data criteria C1 and C2.

A function cluster, or cluster for short, exists to support an abstraction found useful in conceiving the design of a program. The abstraction represented by a cluster is an abstract data type together with the operations which may be performed on objects of the data type. A cluster consists of a group of related functions which share a local environment

of data belonging to the cluster. The local environment of a cluster is private and may not be accessed from outside of the cluster. To use a cluster, a variable is declared to be of the abstract type supported by the cluster; then the functions of the cluster are used to perform operations on the variable. No other way of operating on the variable is possible.

We believe clusters provide a very good way of incorporating abstract data types in a language, and we intend clusters to be a fundamental construct of our structured programming language. The cluster will be a syntactic entity of the language, so that clusters may be defined and used. In addition, the language will be strongly typed so that a variable of some abstract type may only be operated on by the functions of the cluster supporting that type. For example, a cluster defining stack as an abstract data type would contain segments 'push', 'top', etc. that define these operations on stacks. The representations chosen for stack would be implicit in the declaration of data in the local environment of the cluster and the content of segments 'push', 'top', etc. From outside the cluster for stack, this information will be accessible only through application of the stack operations of the cluster to some object of type stack.

The concept of a function cluster does not appear in any existing programming language, but the language Simula 67 [86] contains a similar concept in its class construct. Like a cluster, a Simula 67 class gathers together all information about an abstract data type. This information consists of both declarations describing the representation of the abstract data objects and definitions of the functions which perform operations on the abstract data objects. The most important distinction between Simula 67 classes and function clusters is that in Simula 67, the functions in the class and the information about the representation of the abstract objects are equally accessible. Therefore Simula 67 does not satisfy our second criterion for abstract data.

Although developed through considerations of structured programming, the concept of a function cluster is useful for modular programming because it appears to provide a very satisfactory form for a program module.

14. A Formalism for Data Structures

Steve Zilles is developing a formal treatment of data structures which will allow a representation independent formal definition of data types and will also yield a framework for proving the correctness of particular representations. The primary goal of this work is to facilitate structured and modular programming by providing a language in which to specify the meaning of function clusters that represent structured data types.

A secondary goal is to be able to define data in a way which is suitable for automatic programming. The main point is that one should not be forced into making unnecessary representation commitments in expressing a solution to a problem. This allows operations to be expressed more nearly

in terms of the original problem domain. Then a wide range of possible representations can be explored to find the one which is most efficient for the given problems without requiring any changes in the high level problem representation.

Consider the approach to defining data structures proposed by T. A. Standish [88]. Standish defines a data system M to be 6-tuple $M = \{S, A, C, \Lambda, \sigma, \alpha\}$ where S is a non-empty set of selectors, A is a set of elementary objects, C is a set of compound objects, Λ is a distinguished null object. Letting $O = A \cup C$, σ is a map $O \times S \to O$ which specifies the operation of selection on objects. Thus $\sigma(o, s)$ is the component object of object o chosen by selector s. The map $\alpha: O \times S \times O \to C$ specifies the operation of appending. Thus $\alpha(o, s, o')$ is the compound object obtained by making object o' the s-component of o.

A data system M must satisfy the following axioms.

1. Axiom for Elementary Objects
   If o is an elementary object then $\sigma(o, s) = \Lambda$ for each $s \in S$.

2. Axiom for the Null Object
   $\Lambda \in A \cup C$

3. Axiom for Equality of Compound Objects
   For any two compound objects $c_1, c_2 \in C$; if for all $s \in S$, $\sigma(c_1, s) = \sigma(c_1\ s)$ then $c_1 = c_2$

4. Axiom for Assignment
   For any $s_1, s_2 \in S$ and $o_1, o_2 \in O$: if $s_1 = s_2$ then $\sigma(\alpha(o_1, s_1, o_2), s_2) = o_2$ else if $s_1 \neq s_2$ then $\sigma(\alpha(o_1, s_1, o_2), s_2) = \sigma(o_1, s_2)$

The purpose of the axioms is to formally characterize the operations of selection (σ) and component-wise update (α) that are applicable to many if not most data structures.

A particular data structure class (i.e., data type) would be defined by augmenting the above axioms with additional axioms which would limit and define the domain of the selector set S, limit and define the purpose and equality of the elementary objects A, and also limit the range of modifications that are possible via α. Any particular class of data structures would be a model for the axioms if it is consistent with the axioms.

Steve Zilles has studied the class of models for the above axioms. A characterization of the class of models was developed to determine which data structures satisfy the axioms. The second aspect of the research was to extend the ideas used to define the axioms to define other classes of data structures. With respect to the class of models, the following have been shown:

Theorem: Let M = (S, A, C, Λ, σ, α) be a model for the data axioms. If $|S| \geq 2$ and $|C| \geq 2$ or $|S| \geq 1$ and $|A| \geq 2$ then the cardinality of O is at least denumerable.

Theorem: Every finite model of a data system M = (S, A, C, Λ, σ, α) has $|S| = |A| = 1$ and for each n there is a one-to-one correspondence between models having $|C| = n + 1$ and the set of permutations on n distinct symbols.

These results have as a corollary the fact that it is possible to simulate Peano arithmetic using the primitives of any unbounded model of a data system.

To further restrict the class of models to those that can be constructively generated, we add an axiom that corresponds to Peano's principle of induction.

5. Let X be a set of elements in O such that:
   a. $A \subseteq X$
   b. $\alpha(o_1, s, o_2) \in X$ whenever $o_1, o_2 \in X$ and $s \in S$.

   Then X = O.

One can show that each model for axioms 1. through 5. is isomorphic to a set of finite rooted trees with labeled edges. One can also show that the objects defined for the Vienna Definition Language [89] satisfy these five axioms. Therefore, the data systems inductively define the class of tree structured data structures.

Not all data structures fit naturally into the above class. For example, a set is a data structure without selectors for its elements. However, the approach used to define the class of tree structured data structures can be modified to apply to other data types in a form that matches the concept of function clusters defined above. Thus an abstract data type will consist of a set of objects and a finite set of functions that define the operations of the data type. Formally, we represent an abstract data type as {D, I, O, F} where D is the set of objects, and F is the set of functions. The components I and O contain sets which are auxiliary input and output domains, respectively, of the functions in F. Each element f_i of F is either a query function with functionality

$$f_i: \quad D \times I_i \rightarrow O_i$$

or an update function with functionality

$$f_i: \quad D \times I_i \rightarrow D$$

where I_i is a direct product of domains in I and O_i is a domain in O. Although some operations on data types (such as the

next-record function of a cluster for sequential files) perform both modification or update, and an observation, such combined functions can be defined in terms of separate update and query functions.

The definition of an abstract data type is completed by a set of recursive equations which assert relationships between the query and update functions. The effect of an update function on an object is specified by the changes that are observed in subsequent applications of functions to that object. The equations always include specification of the result of applying each query function to each of a set C of distinguished objects in D. The result of each query function for objects not in C can usually be specified by equations which specify the result of applying each query function to an update of an arbitrary member of D. This approach is similar to but more xx general than the paradigm recently exposited by Parnas [90].

As an example, we show how a data type integer set would be defined. Let D be the set of integer sets, I be the set of integers, O be the set {true, false}, and F be {insert, remove, has}. The query function is

$$\text{has}: \quad D \times I \rightarrow O$$

and the update functions are

$$\text{insert}: \quad D \times I \rightarrow D$$
$$\text{remove}: \quad D \times I \rightarrow D$$

The defining equations for these functions make use of a single distinguished object, the empty set $\emptyset$, and are as follows:

R_1: has(insert(S, i), j) $\equiv$ if i = j then T
else has(S, j)

R_2: has(remove(S, i), j) $\equiv$ if i = j then F
else has (S, j)

R_3: has($\emptyset$, j) $\equiv$ F for all j $\in$ I

The equations (R_1, R_2, R_3) complete the definition of the type integer set. The similarity in style between the equations and the axioms for a data system should be apparent.

The next phase of this research is to develop a proof technique for showing the correctness of representations of abstract data types that are defined in the manner outlined above. In conjunction with this work, the appropriate use of protection mechanisms for data will be demonstrated.

15. A Highly Parallel Processor

Highly parallel computers such as the Illiac IV and the CDC Star achieve their processing speed by imposing constraints on the structure of the data being processed. Both of these machines are organized to perform very well for data represented as vectors. To realize its potential, computation using the

Illiac IV must be organized to exploit the machine's ability to execute simultaneously the same operation for many sets of operands. In the case of the CDC Star, the maximum processing rates of the pipelined processing unit is approached only if the computation is organized to make effective use of streaming operations on very long vectors. In both machines the programmer is forced to use unusual and intricate data representations if highly parallel execution is to be achieved. Thus these machines are developments contrary to what is generally seen as one of the most important issues in contemporary computer practice -- the difficulty of developing correct programs. Even such an important notion as the use of subroutines is inadequately supported in these machines.

The Computation Structures Group is developing a computer organization that can yield highly concurrent operation with no sacrifice in the generality of programming that can be supported. The ultimate goal of this work is to develop a machine organization that can directly support programs expressed in terms of the base language also under development by the Group. A general idea of the concepts expected to be employed in the design of such a machine has been given by Dennis [63]. Unusual features include a memory system designed to support tree-structured objects as the basic means for data and program representation, and instruction sequencing based on the availability of the required operands.

A fundamental problem in the design of a highly parallel computer is to devise a means of efficiently moving instructions and their operands from the memory system to appropriate functional units for execution. In response to an interesting application requiring highly parallel, signal-processing computations, Jack Dennis and David Misunas have developed an elegant solution to the memory/processor interconnection problem for this specialized class of programs. These programs correspond to data flow schemas in which each actor is an operator. In computations specified by such programs all operators are executed equally many times. Digital signal processing computations involving waveform generation, summing, modulation, and filtering are well suited to specification as programs in this class. The design of a highly parallel processor for these programs is described briefly in the following paragraphs. In future work we expect to further develop these ideas and develop principles for highly parallel machines that execute more general classes of programs based on data flow models of computation.

So operation of the proposed machine may be readily understood we shall use the data flow schema in Figure 27 as the basis for an illustrative program. This schema represents the computation required for a second-order recursive digital filter

$$y(t) = Ax(t) - By(t - 1) + Cy(t - 2)$$

where $x(t)$ and $y(t)$ denote input and output samples for time t. In this schema operators 1, 2 and 3 are unary operators

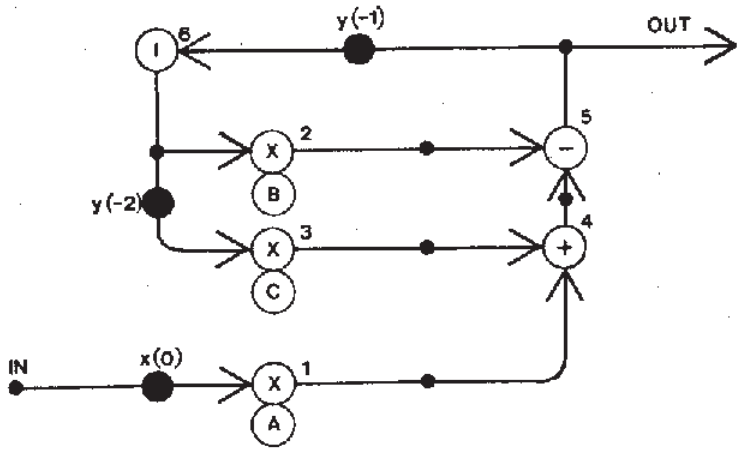


Figure 27

Data flow schema for second order filter

that multiply by the fixed parameters A, B and C. Operators 4 and 5 are binary operators that perform addition and subtraction, and operator 6 is an identity operator that transmits its input values unchanged. The solid dots show the presence of values at certain input arcs of operators and define the initial configuration of the schema. An operator with values present on each of its input arcs is ready to act, and does so by removing values from its input arcs and placing the result of its application on the input arc of each successor operator.

The general organization of the proposed processor is shown in Figure 28; there are four major sections:

memory section
arbitration network
functional units
distribution network

The design is conceived to make advantageous application of our knowledge of asynchronous modular systems [91, 92]. There are no signals or circuits having timing as their sole function. Each connection between sections of the machine (and most connections between smaller units as well) are independently coordinated using an acknowledge signal for each unit of data sent over the connection.

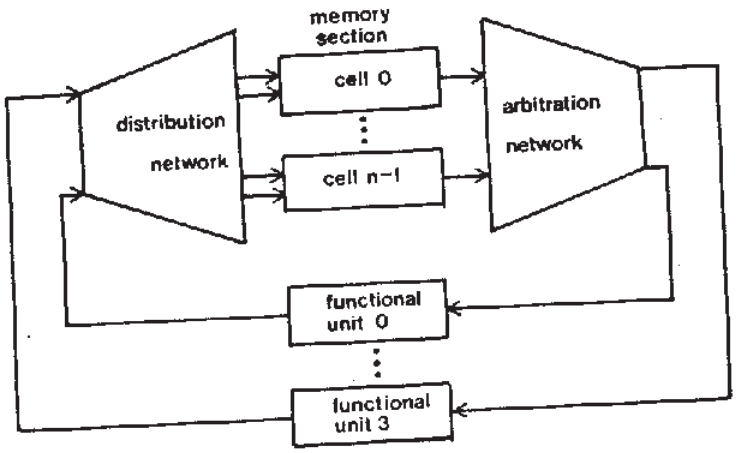


Figure 28

General organization of the processor

The information units transmitted through the arbitration network from the memory section to the functional units are instruction packets; each instruction packet specifies one unit of work for the functional unit to which it is directed. The information units sent through the distribution network from functional units to the memory section are result packets; each result packet delivers a newly computed value to a specific value holding register in the memory section.

The memory section of the processor holds a representation of the program and holds computed values awaiting use. The memory is a collection of cells; each operator in the program is associated with some cell of the memory. Each cell (Figure 29) contains three registers -- one register to hold an instruction which encodes the type of operator and its connections to other operators of the program, and two registers that receive operand values for use in the next execution of the instruction. Each operand register may be set to behave as a constant or as a variable. If set to act as a variable, an operand register expects to receive a result packet containing the operand value through the distribution network; if set to act as a constant, an operand register retains the value delivered to it when the program was loaded into the memory. When both operand registers contain values, the cell is said to be enabled and the contents of the three registers (instruction and two operand values) form an instruction packet which is presented to the arbitration network.

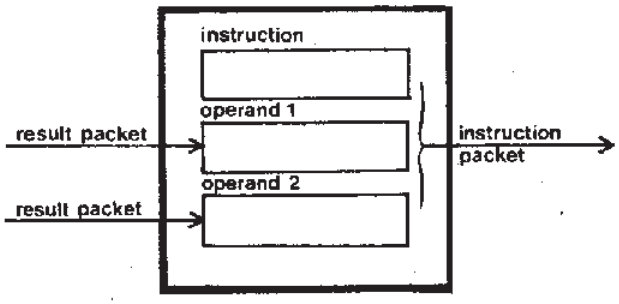


Figure 29

Registers of a memory cell

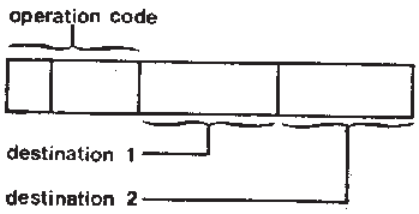


Figure 30

Instruction format

Figure 30 shows the instruction format. The operation code has a field (of two bits, say) which specifies the functional unit required (out of four, say), and a field that indicates which capability of the functional unit is to be used. Each destination field specifies a memory cell and which of its operand registers is to receive one copy of each result generated by the instruction. The initial contents of memory cells for the digital filter example is shown in Figure 31.

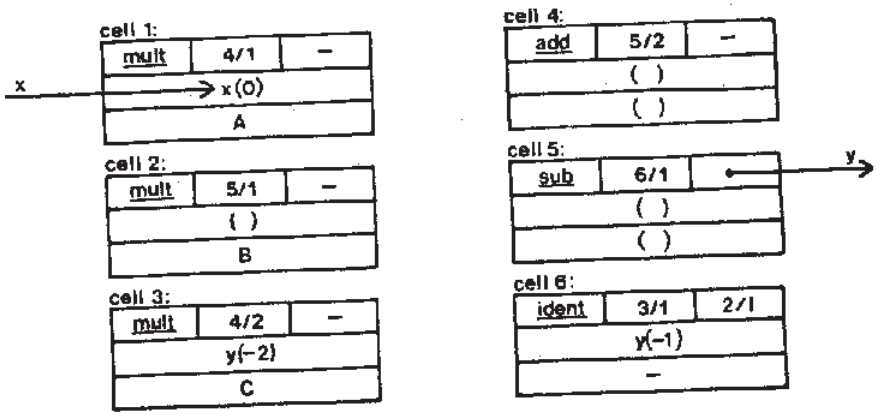


Figure 31

Initial contents of memory for the second order filter computation

Empty parentheses indicate an operand register waiting to receive a value. In Figure 31 cell 1 is enabled and presents the instruction packet

$$\begin{Bmatrix} \underline{\text{mult}},\ 4/1,\ - \\ x(0) \\ A \end{Bmatrix}$$

to the arbitration network. Some functional unit will compute

$$z = A\ x(0)$$

and send the result packet

$$\begin{Bmatrix} 4/1 \\ z \end{Bmatrix}$$

through the distribution network and operand register 1 of cell 4 will receive the value z.

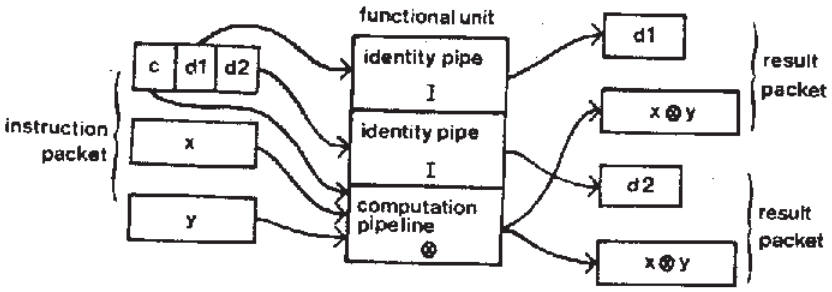


Figure 32

Operation of a functional unit

As illustrated in Figure 32, each functional unit receives from the arbitration network all instruction packets directed to it by their operation codes, and, in general, delivers two result packets to the distribution network. To realize maximum throughput, each functional unit is constructed as three pipelines: one pipeline performs the computation of the result value $z = x \otimes y$ where x and y are the operands from the instruction packet. The second and third pipelines carry the destination codes d1 and d2 so these may be associated with the result z when it emerges from the computational pipeline.

Since the data flow form of a program expresses most of the possibilities for concurrent execution of instructions, we can expect that many cells in the memory section of the processor will be enabled at once. As the functional units have high potential throughput, we must show how the arbitration and distribution networks can be organized to handle many packets concurrently so all sections of the processor are effectively utilized. The arbitration network is designed so many instruction packets may flow into it concurrently from cells of the memory system, and merge into four streams of packets -- one for each functional unit. The network is built of the four types of modules shown in Figure 33.

The arbitration module passes to its output port c packets arriving at input ports a and b, one-at-a-time, using a round-robin discipline to resolve any ambiguity about which packet should be sent next. The switch module assigns packets arriving at port a to ports b or c according to some property of the packet. In the arbitration network switch modules separate instruction packets into four categories, one for each functional unit, by testing the operation codes of the instructions they contain. Figure 34 shows how arbitration and switch modules might be arranged into an arbitration network. This

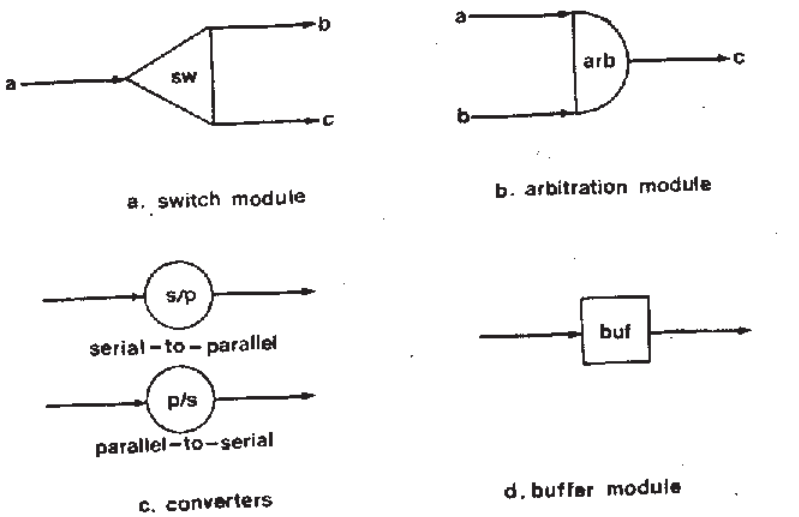


Figure 33

Module for the arbitration and distribution networks

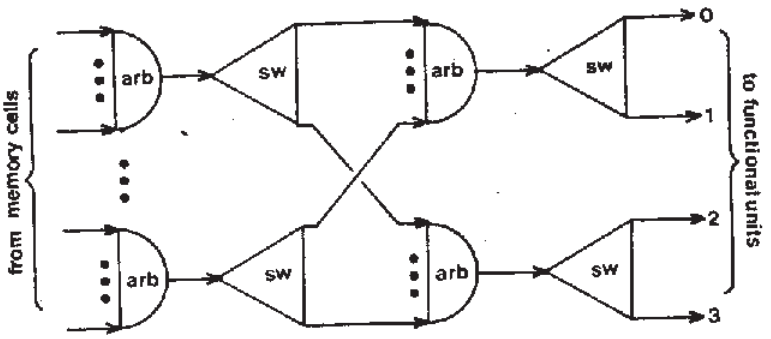


Figure 34

Primitive arbitration network

network contains a path for instruction packets from each memory cell to each functional unit.

Since the arbitration net has many input ports and only four output ports, the rate of packet flow will be much greater at the output ports. Thus a serial representation of packets is appropriate at the input ports to minimize the number of connections to the memory section, but a more parallel representation is required at the output ports so a high throughput may be achieved. Evidently, serial-to-parallel conversion is required within the arbitration network, and conversion modules (Figure 33c) must be included. In addition, a packet emerging from a conversion module must be prevented from engaging a subsequent arbitration module until the serial packet has been completely absorbed by the conversion module. Thus buffer storage modules are needed at the output of each converter. Figure 35 shows an improved arbitration network including conversion modules and buffers.

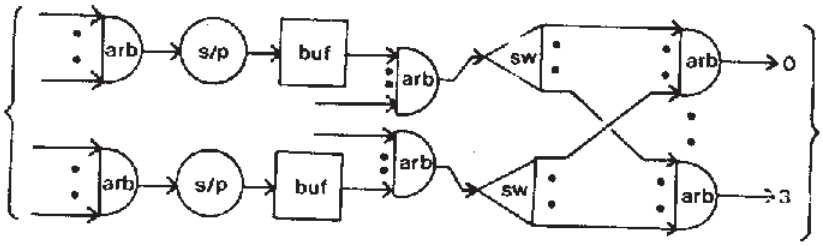


Figure 35

Improved arbitration network

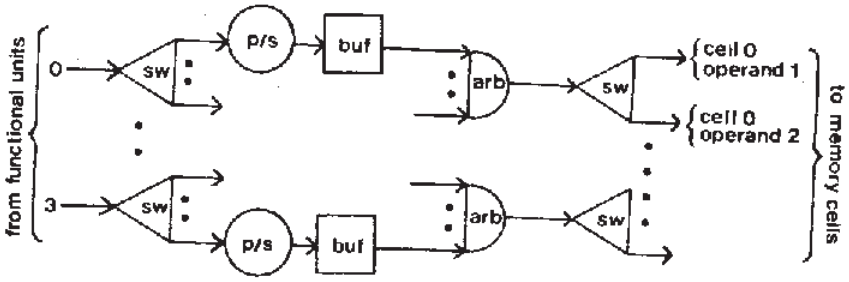
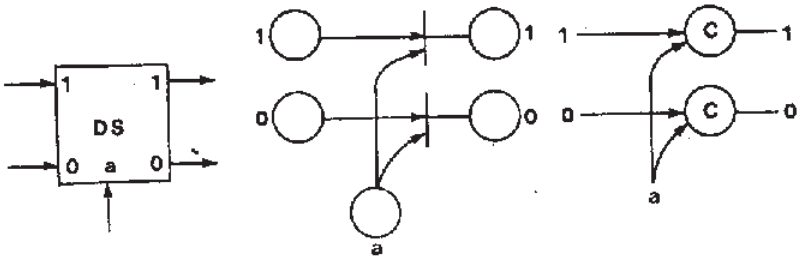


Figure 36

Distribution network

The distribution network is similarly organized. As shown in Figure 36, many switch modules route result packets to the operand registers specified by their destination codes. A few arbitration modules are required so result packets with different destination codes may enter the distribution network concurrently and share use of the second rank of switch modules.

We have found this new concept of computer organizations a very attractive application for our work on speed-independent logic design. David Misunas has developed preliminary designs for many parts of the proposed machine [93], and his paper on this work [94] tied for second place in the 1972 ACM student paper competition. The use of Petri nets as a formal notation for specifying the behavior of asynchronous modules has been very important to the success of this work. Some examples of useful asynchronous modules are shown in Figures 37 through 39. In these modules reset signalling [92] is used; that is, transitions from 0 to 1 denote significant events and must be followed by transitions from 1 to 0 called reset events. The Muller C-element is discussed in [92]. The data switch allows events representing bits of data to be forwarded to a particular destination under control of events occurring on wire a. Data switches are also used in combination as in Figure 38. The values of control bits arriving at control port d determine which of data output ports b and c is to receive each data bit arriving at data input port a. This circuit is speed independent in that arbitrary delays may be inserted in any external connection without interfering with its correct operation. Internally, the circuit is insensitive to the speed of operation of its active elements. Figure 38 shows the bit pipeline which is the speed independent counterpart of the conventional shift register and has a natural application in the registers of the memory system.



a. module b. Petri net c. circuit

Figure 37

Data switch module

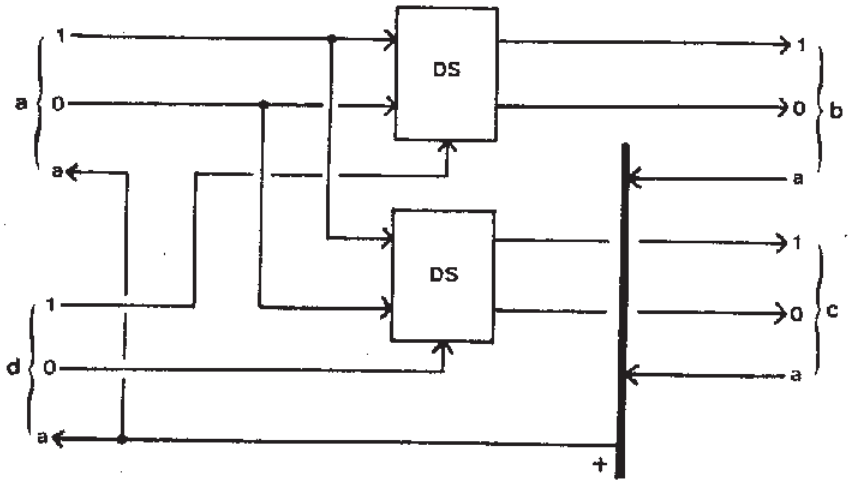


Figure 38

Speed independant module built from data switches

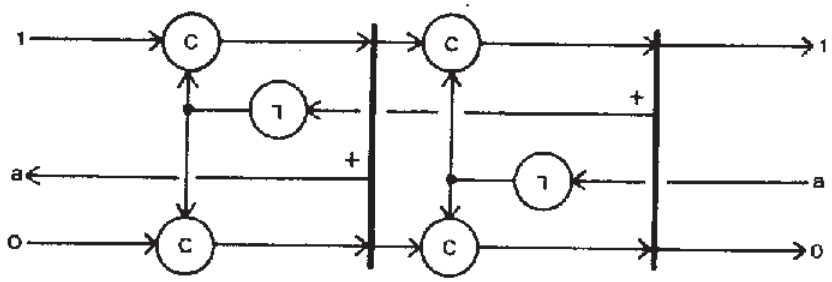


Figure 39

A section of bit pipeline

References

52. Project MAC Progress Report IX, July 1971 to July 1972, Massachusetts Institute of Technology, Cambridge, Mass., pp. 17-21.

53. Plummer, W. W., Asynchronous arbiters, IEEE Trans. on Computers, Vol. C-21, No. 1 (January 1972), pp. 37-42.

54. Patil, S. S., Synchronizers and arbiters, paper in preparation.

55. Lester, B. P., The Balance Property of Parallel Computation, Ph.D. Thesis, Department of Electrical Engineering, M.I.T., Cambridge, Mass. (in preparation).

56. Project MAC Progress Report IX, July 1971 to July 1972, Massachusetts Institute of Technology, Cambridge, Mass., pp. 9-16.

57. Ramachandani, C., On the computation rate of asynchronous computation systems, Proceedings of the Seventh Annual Princeton Conference on Information Sciences and Systems, 1973, pp. 276-280.

58. Ramachandani, C., Analysis of Asynchronous Systems by Timed Petri Nets, Ph.D. Thesis, Department of Electrical Engineering, M.I.T., Cambridge, Mass., September 1973.

59. Karp, R. M. and Miller, R. E., Properties of a model for parallel computations: determinancy, termination, queueing, SIAM Journal of Applied Mathematics, Vol. 14, No. 6 (November 1966), pp. 1390-1411.

60. Hack, M., Extended State-Machine Allocatable Nets (ESMA), an Extension of Free Choice Petri Net Results, Computation Structures Group Memo 78, Project MAC, Massachusetts Institute of Technology, Cambridge, Mass., May 1973.

61. Baker, H., Equivalence Problems of Petri Nets, S.B. Thesis. Department of Electrical Engineering, M.I.T., Cambridge, Mass., June 1973.

62. Keller, R. M., Vector Replacement Systems: A Formalism for Modeling Asynchronous Systems, Technical Report 117, Princeton University, Computer Science Laboratory, Princeton, New Jersey, December 1972.

63. Dennis, J. B., Programming generality, parallelism and computer architecture, Information Processing '68, North-Ho Holland Publishing Co., Amsterdam, 1969, 484-492.

64. Dennis, J. B., On the design and specification of a common base language, Proceedings of the Symposium on Computers and Automata, Polytechnic Press of the Polytechnic Institute of Brooklyn, New York, 1971, pp. 47-74.

65. Project MAC Progress Report VIII, July 1970 to July 1971, Massachusetts Institute of Technology, Cambridge, Mass., pp. 26-51.

66. Amerasinghe, S. N., The Handling of Procedure Variables in a Base Language, S.M. Thesis, Department of Electrical Engineering, M.I.T., Cambridge, Mass., September 1972.

67. Dennis, J. B. and Fosseen, J. B., Introduction to Data Flow Schemas (manuscript to be submitted for publication).

68. Linderman, J., Productivity in Parallel Computation Schemata, Ph.D. Thesis, Department of Electrical Engineering, M.I.T., Cambridge, Mass., June 1973.

69. Karp, R. M. and Miller, R. E., Parallel program schemata, J. of Computer and System Sciences, Vol. 3, No. 2, May 1969, pp. 147-195.

70. Project MAC Progress Report IX, July 1971 to July 1972, Massachusetts Institute of Technology, Cambridge, Mass., pp. 21-28.

71. Luckham, D. C., Park, D. M. R. and Paterson, M. S., On formalised computer programs, J. of Computer and System Sciences, Vol. 4, No. 3, June 1970, pp. 220-249.

72. Kfoury, D. and Park, D. M. R., Programs terminate because of second order reasons, Proceedings of the Logic Conference and Meeting of the Association for Symbolic Logic, Orleans, France, September 1972.

73. Hawryszkiewycz, I. T., Semantics of Data Base Systems, Ph.D. Thesis, Department of Electrical Engineering, M.I.T., Cambridge, Mass., September 1973.

74. Codd, E. F., A relational model of data for large shared data banks, Comm. of the ACM, Vol. 13, No. 9, June 1970, pp. 377-387.

75. Codd, E. F., Normalized data base structure: a brief tutorial, Proceedings of the 1971 ACM-SIGFIDET Workshop on Data Description, Access and Control.

76. Dennis, J. B., Modularity, Advanced Course on Software Engineering, Lecture Notes in Economics and Mathematical Systems, Springer-Verlag, 1973, pp. 128-182.

77. Dijkstra, E. W., Go to statement considered harmful, Comm. of the ACM, Vol. 11, No. 3, March 1968, pp. 147-148.

78. Mills, H. D., Top down programming in large systems, Debugging Techniques in Large Systems, R. Rustin (Ed.), Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1971, pp. 41-55.

79. Dijkstra, E. W., Notes on Structured Programming, Technische Hogeschool, Eindhoven, Netherlands, August 1969.

80. Wirth, N., Program development by stepwise refinement, Comm. of the ACM, Vol. 14, No. 4, April 1971, pp. 221-227.

81. Dijkstra, E. W., The humble programmer, Comm. of the ACM, Vol. 15, No. 10, October 1972, pp. 859-866.

82. Wulf, W. and Shaw, M., Global variable considered harmful, SIGPLAN Notices, Vol. 8, No. 2, February 1973, pp. 28-34.

83. Liskov, B. H., A design methodology for reliable software systems, AFIPS Conference Proceedings, Vol. 41 (FJCC 1972), pp. 191-199.

84. Wegbreit, B., Brosgol, B., Holloway, G., Prenner, C., and Spitzen, J., ECL Programmer's Manual, Center for Research in Computing Technology, Harvard University, Cambridge, Mass., September 1972.

85. Wirth, N., The programming language PASCAL, Acta Informatica, Vol. 1, 1971, pp. 35-63.

86. Dahl, O-J. and Nygaard, K., SIMULA -- an ALGOL-based simulation language, Comm. of the ACM, Vol. 9, No. 9, September 1966, pp. 671-678.

87. Dahl, O-J. and Hoare, C. A. R., Hierarchical program structures, Structured Programming, Academic Press, New York, 1972, pp. 175-220.

88. Standish, T., Unpublished manuscript.

89. Lucas, P., Lauer, P., and Stigleitner, H., Method and Notation for the Formal Definition of Programming Languages, IBM Technical Report TR.25.087, IBM Laboratory, Vienna, 1968.

90. Parnas, D., A technique for software module specification with examples, Comm. of the ACM, Vol. 15, No. 5, May 1972, pp. 330-336.

91. Dennis, J. B. and Patil, S. S., Speed independent asynchronous circuits, Proceedings of the Fourth Hawaii International Conference on System Sciences, Western Periodicals Co., North Hollywood, Calif., 1971, pp. 55-58.

92. Patil, S. S. and Dennis, J. B., The description and realization of digital systems, Sixth Annual IEEE Computer Society International Conference, Digest of Papers 1972: Innovative Architecture, IEEE, New York, N. Y., pp. 223-226.

93. Misunas, D. P., Petri Nets and Speed Independent Design, S.B. Thesis, Department of Electrical Engineering, M.I.T., Cambridge, Mass., July 1972.

94. Misunas, D. P., Petri nets and speed independent design, Comm. of the ACM, Vol. 16, No. 8, August 1973, pp. 474-481.