MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

Computation Structures Group Memo 77

Computation Structures Group
Progress Report 1971-72

This research was done at Project MAC, MIT, and was supported in part by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr N00014-70-A-0362-0001, and in part by the National Science Foundation under grant GJ-432.

April 1973

COMPUTATION STRUCTURES

Prof. J. B. Dennis

Academic Staff

Prof. R. M. Fano

Prof. S. S. Patil

Instructors, Research Associates, Research Assistants and Others

N. Amerasinghe
H. G. Baker
R. Barquin
P. B. Bishop
R. Cohen
J. Fosseen
P. J. Fox
F. Furtek
T. M. Gearing
I. Greif
M. Hack
I. T. Hawryszkiewycz
D. A. Henderson
B. Lester
J. P. Linderman
J. B. Lotspiech
J. A. Meldman
J. E. Qualitz
C. Ramchandani
L. J. Rotenberg
J. E. Rumbaugh
R. J. Steiger
S. R. Umarji

Undergraduate Students

G. G. Bajoria
H. J. Kim
C. K. Leung
D. Misunas
J. Phillips
S. Sadeq
J. C. Schaffert
R. Swift

DSR Staff

J. Nievergelt

Support Staff

B. A. Morneault

A. Rubin

COMPUTATION STRUCTURES

The Computation Structures Group is concerned with the analysis of fundamental issues arising in the design and construction of general-purpose computer systems through the formulation and study of appropriate abstract models. The past year has seen new developments in the theory and application of Petri nets as a model of systems of interacting parts, improved techniques for realizing digital systems with assurance of correct operation, development of the theory of data flow schemata, and contributions to the study of program correctness and programming generality.

A. Petri Nets

Our research relating to Petri nets is concerned with the theory of Petri nets, the relation of nets to logic circuits and asynchronous modular systems, and the use of Petri nets as a model for the behavior of systems of interacting parts, including systems within and outside the domain of computer science.

Timed Petri Nets

Chander Ramchandani is investigating the use of Petri net models in the performance analysis of systems. Petri nets (8, 5) are an attractive model for studies of system performance because the important interactions between system parts are easily represented. Petri nets represent the ordering relationship of events in a system that mark the initiation and termination of activities, but do not represent the timing of events or durations of activities. For performance analysis the Petri-net model of a system must be augmented with timing information.

In a Petri net (Figure 1), the firing of a transition may represent an interval of activity by some system part. If the transition is enabled (at least one token in each of its input places) it means that activity of the system part may begin. We associate initiation of activity with picking up one token from each input place, and termination of activity with adding one token to each output place. This corresponds to considering the transition to be two transitions and a place p as in Figure 2.

Figure 3 shows a timed Petri net obtained by associating time parameters with certain transitions of the net in Figure 1. In a timed net transitions without time parameters represent sequencing constraints on activities as in a conventional Petri net. Action of a timed transition may be explained in terms of Figure 2, where the time parameter r(t) is associated with place p. Transition t' may fire immediately when enabled or any time later (providing it remains enabled). Then transition t" becomes enabled and fires exactly r(t) time units after the firing of t'. Thus the firing of transitions t' and t" represents initiation and termination of one instance of the activity represented by transition t. It is possible for a transition t in a timed net to be re-enabled before a previously initiated instance of the associated activity has terminated. In fact, many instances of the activity may be in

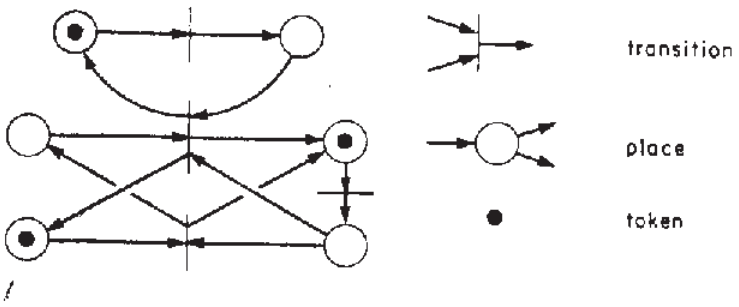


Figure 1. A Petri net.

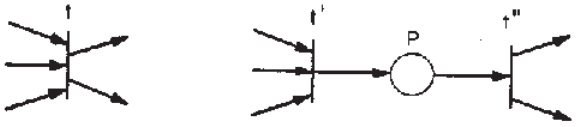


Figure 2. Meaning of a timed transition.

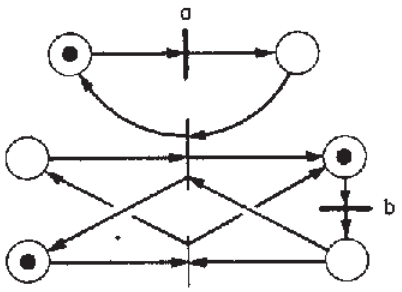
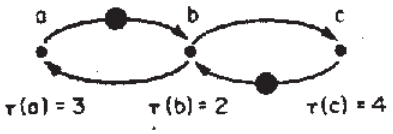


Figure 3. A timed Petri net.

(a) a timed marked graph



(b) periodic schedule

transition a: 2-5, 8-11, 14-17, ...
b: 0-2, 6-8, 12-14, ...
c: 2-6, 8-12, 14-18, ...

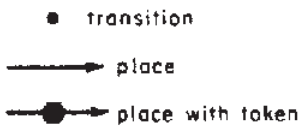


Figure 4. Periodic schedule for a timed marked graph.

progress simultaneously, as we shall see in later examples. The number of tokens in place p is the current number of simultaneous instances of the activity.

A schedule for a timed Petri net is a set of sequences of initiation and termination times for the timed transitions of the net. A schedule is feasible if the timed net can exhibit the behavior specified by the schedule. A schedule is not feasible if it calls for initiation of an activity earlier than allowed by terminations of other activities. A feasible schedule is said to be prompt if each activity always initiates as early as possible. Here are examples of feasible and prompt schedules for the timed net and initial token distribution shown in Figure 3:

(a) a feasible schedule
transition a: 0-5, 8-13, 13-18
b: 2-4, 4-6, 9-11, 11-13, 15-17

(b) a prompt schedule
transition a: 0-5, 5-10, 10-15
b: 0-2, 2-4, 5-7, 7-9, 10-12

Every timed net for which the underlying Petri net is persistent (no transition ceases being enabled except by firing) has a unique prompt schedule.

We have studied the class of Petri nets known as marked graphs. In a marked graph, each place is an input place of at most one transition, and an output place of at most one transition. All transitions of a marked graph fire equally many times in any behavior that returns the net to its original configuration. In consequence, a prompt schedule for a timed graph is periodic in that each timed transition initiates at regular intervals. The example in Figure 4 has a periodic prompt schedule with period six. In this case, the rate of firing is determined by the circuit containing transitions b and c.

Figure 5 illustrates a situation where several instances of an activity represented by transition b may proceed concurrently. Instances of the activity represented by transition a are forced to occur strictly in sequence by the one-token self loop. The prompt schedule shown has a period of eight.

The computation rate of a timed marked graph is the average rate of firing for any transition of the graph in a prompt schedule. For the example in Figure 4 the rate is 1/6; for Figure 5, the rate is 1/4.

There is a simple algorithm for determining the computation rate of a timed marked graph. Let the vertices (transitions) and arcs (places) of a strongly connected marked graph be

$$V = \{v_1, \ldots, v_\ell\}$$

$$A = \{a_1, \ldots, a_p\}$$

where an arc $a_m = (v_i, v_j)$ is directed from transition v_i to v_j, and let τ_i be the time associated with transition v_i ($\tau_i = 0$ if v_i is not a timed transition). For any strongly connected marked graph one can find a set of simple circuits $C_1, \ldots, C_m$ that cover all arcs of the graph (5). Let M_{ij} be the number of tokens on arc (v_i, v_j) in the initial marking of the net. Then the computation rate ρ of the timed marked graph is given by

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

where

$$T_k = \sum_{v_i \in C_k} \tau_i$$

is the sum of the times associated with transitions of circuit C_k and

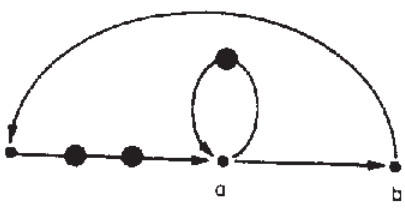
$$N_k = \sum_{(v_i, v_j) \in C_k} M_{ij}$$

is the number of tokens on arcs of circuit C_k.

Figure 6 shows a "PERT" chart with activities a,b,c,d,e and the corresponding timed marked graph. Application of the foregoing procedure shows that the computation rate is 1/8, the reciprocal of the time for the critical path. We may ask what happens to the computation rate if N_p processors are permitted to perform activities concurrently. The corresponding marked graph is shown in Figure 7, where it is assumed that only N_R instances of activity e are permitted at one time, but arbitrarily many instances are possible for the other activities. The figure gives the computation rates for several values of N_p and N_R.

Work is continuing on performance analysis of systems represented by more general classes of Petri nets. Also, the properties of Petri nets having time bounds or statistical

(a)

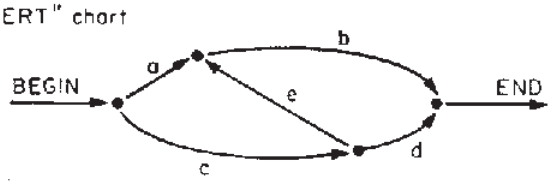


(b)

transition a: 0-2, 2-4, 8-10, 10-12,...
b: 2-8, 4-10, 10-16, 12-18,...

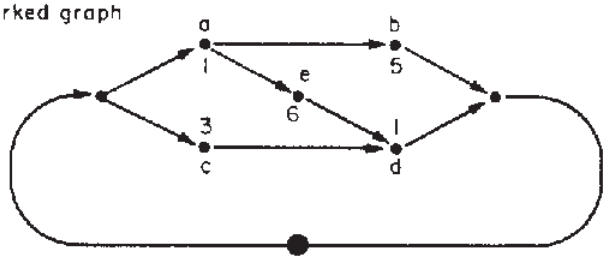
Figure 5. Marked graph with concurrent instances of an activity.

(a) "PERT" chart



$\tau(a) = 1$
$\tau(b) = 5$
$\tau(c) = 3$
$\tau(d) = 1$
$\tau(e) = 6$

(b) marked graph



circuit	N_k	T_k
ab	1	6
cd	1	4
aed	1	8

Figure 6. Computation rate of a timed marked graph

distributions associated with transitions are being studied.

Canonic Forms for Petri Nets

We have begun investigation of notions of equivalence and canonic forms for Petri nets. For the special case of marked graphs, Henry Baker (2) has shown how to reduce any marked graph to a simple form which is the same for all marked graphs equivalent to the given marked graph.

Suppose G is a marked graph and N is some subset of the transitions of G. Then if ω is a firing sequence of G, the corresponding derived firing sequence ω_N is obtained from ω by erasing all elements that are not members of N. Let G and G' be marked graphs and let $N = \{t_1, \ldots, t_n\}$ be a set of n transitions that appear in both G and G'. We say that G and G' are equivalent with respect to N if for each firing sequence ω of G there is a firing sequence ω' of G' such that ω_N and ω'_N are identical, and vice versa. The two marked graphs in Figure 8 are equivalent with respect to N = {a,b} since in each case the set of derived firing sequences is $(ab \cup ba)^*$.

First we give two rules which when applied to any marked graph will give a simpler marked graph equivalent to the original with respect to all of its transitions:

Rule 1: If an arc originates and terminates on the same transition, and has at least one token, it may be deleted.

Rule 2: Let a and b be any two distinct transitions, and let x be an arc from a to b. If the number of tokens on arc x is greater than or equal to the total number of tokens on the arcs of any other simple, directed path from a to b, then arc x may be deleted.

Use of the two rules is illustrated in Figure 9. Rule 1 is used to remove arc 1, and rule 2 is used to delete arcs 2, 3 and 4. For each of the three marked graphs, the firing sequences are all prefixes of the infinite string $(abc)^\infty$. A marked graph for which no applications of the two rules are possible is called a minimal-arc marked graph.

The minimal arc form of a marked graph always has the same set of firing sequences as the original marked graph. Furthermore, any pair of marked graphs that are equivalent with respect to a one-to-one correspondence of their transitions have the same minimal-arc form. Thus the minimal arc form is canonic for these marked graphs.

Now suppose N is a set of n transitions common to two marked graphs G and G'. How can we tell whether G and G' are equivalent with respect to N? It turns out that if G is a live marked graph, it may be reduced to an n-transition marked graph equivalent to G with respect to N. This is done by carrying

out the steps below for each transition t of G that is not a member of N:

Step 1: Delete any arcs that originate and terminate at transition t. If any such arc has no token, the marked graph is not live.

Step 2: Let $X = \{x_1, \ldots, x_m\}$ be the set of input arcs and $Y = \{y_1, \ldots, y_n\}$ the set of output arcs of transition t. Let M_i be the number of tokens on arc x_i and let N_j be the number of tokens on arc y_j.

Step 3: Replace transition t and the arcs in $X \cup Y$ with the arcs

$$\{z_{ij} | i = 1, \ldots, m; \quad j = 1, \ldots, n\}$$

where z_{ij} originates on the same transition as x_i and terminates on the same transition as y_j. Put $M_i + N_j$ tokens on arc z_{ij}.

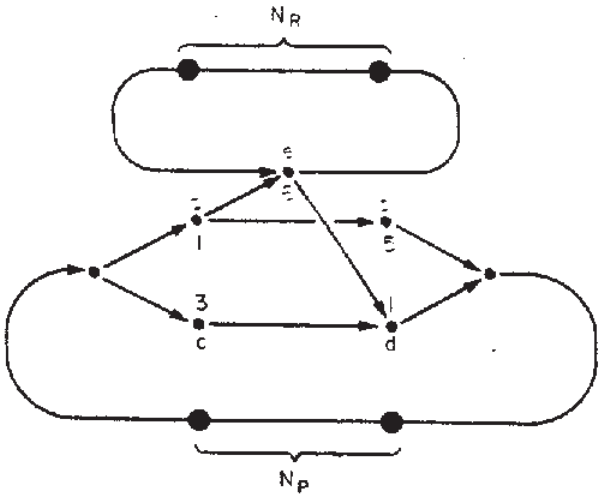
Applying this procedure to either marked graph in Figure 8 gives the canonic form in Figure 10. This example shows that the canonical form for a safe marked graph (5) is not necessarily safe.

B. Arbiters

Arbiters are fundamental units of digital systems that are required whenever two or more asynchronous activities compete for access to a shared unit or resource. A basic form of arbiter known as an elementary arbiter is illustrated in Figure 11. It controls access to a shared resource by two users -- user 1 and user 2. A 0-to-1 transition on either one of the request wires is a signal that the corresponding user desires access to the shared resource. In the absence of a competing request from the other user the arbiter must promptly produce a 0-to-1 transition on the corresponding grant wire. The user signals completion of his use of the resource by a 1-to-0 transition on the request wire, whereupon the arbiter must respond with a 1-to-0 transition on the grant wire. If requests arrive nearly simultaneously from both users, the arbiter must promptly and unambiguously grant either one of the requests and delay granting the second request until the resource is freed. Correct operation of an elementary arbiter must satisfy these conditions:

1. It must never occur that both grant wires are simultaneously at level 1.

2. If both grant wires are at 0 and at least one of the request wires is at 1, the arbiter must grant one of requests.



N_P

N_R	1	2	3	4
1	1/8	1/6	1/6	1/6
2	1/8	1/4	1/3	1/3
3	1/8	1/4	3/8	1/2

Figure 7. Timed marked graph representing several processors and limited throughput of one activity.

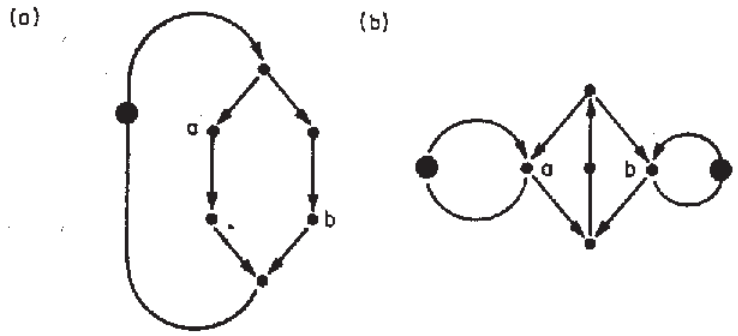
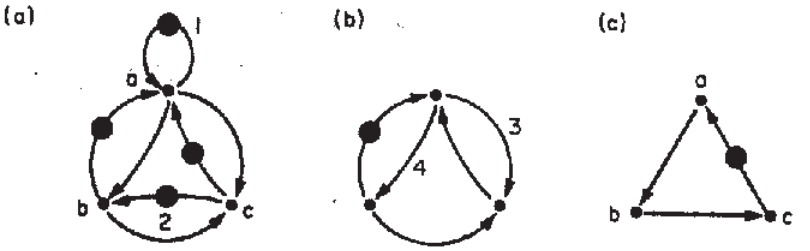


Figure 8. Two equivalent marked graphs.



firing sequences : $(abc)^* (\lambda \cup a \cup ab)$

Figure 9. Simplification of a marked graph.

Figure 10. Canonic form for the marked graphs in Figure 8.

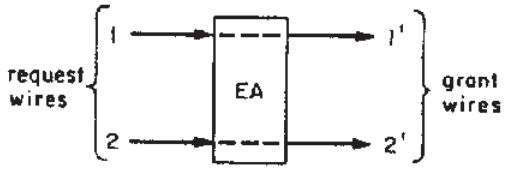


Figure 11. The elementary arbiter.

We have found that any requirement for arbitration in asynchronous digital systems can be met by a modular subsystem using elementary arbiters. For example, an arbiter that oversees sharing of a resource by n users can be built using a binary tree of elementary arbiters (14). The case of n users and m servers has been studied thoroughly by Patil, and he has recently devised an improved solution based on n-user and m-user arbiters (13).

Designing an elementary arbiter that functions correctly and always acts within a specified time interval is a difficult problem. When the two request wires make 0-to-1 transitions nearly simultaneously, the arbiter may make an arbitrary choice, but it must do so without hesitation, and without the appearance of spurious signals on the grant wires.

Suhas Patil has devised an elegant scheme for building an elementary arbiter that will operate correctly in a fixed time with extremely small probability of error. This scheme makes use of a subunit called finite resolution arbiter (FRA) and illustrated in Figure 12. An FRA can fail to operate correctly only if two request signals arrive with a separation of δ time units or less. If an FRA fails, the result is that both grant wires switch to 1.

Now consider a pair of FRA's connected in cascade as in Figure 13a. If two requests arrive at FRA-1 separated by more than δ time units, only one of the request signals will reach FRA-2 and operation will be completed correctly. If requests arrive at FRA-1 with less than δ time units separation, then FRA-1 will transmit both grant signals. Assume for the moment that the two request signals are delayed equally by FRA-1. Then, so long as $\Delta > 2\delta$, the requests arriving at FRA-2 will be separated by more than δ time units and FRA-2 will grant one and only one of the requests.

One of several possible cricuits for a finite resolution arbiter is shown in Figure 14. Each pair of NAND gates forms a set-reset flip flop which is forced into its 1 state by the presence of a request on the associated input wire. The setting of one flip flop prevents the other flip flop from being set, thereby blocking its associated request. If two requests arrive at nearly the same instant, both flip flops will be set since neither will be fast enough to block the other.

The time interval δ is the time separation of request signals such that a request signal and a block signal arrive simultaneously at one of the flip flops. In this circumstance the flip flop may be placed in a metastable state in which it may remain for an arbitrarily long time (with decreasing probability). The existence of metastable states, and the certainty that failures caused by circuits persisting in metastable states have been problematic in computer systems has been nicely explained by Ornstein (4).

It is reasonable to model flip flop behavior for critical input timing as follows: If the set and reset inputs become 1 with time separation less than some small fixed interval ε, the flop flop enters its metastable state, and the probability of commitment to one of the stable states during an interval dt after elapsed time τ is $P(\tau)dt = (1/T)e^{-\tau/T}dt$ where T is a characteristic time of the flip flop. An exponential density function $P(\tau)$ is used because we expect that the probability of commitment during any interval, given that commitment has not occurred earlier, is independent of the elapsed time.

From Figure 15 we see that the cascade of two FRA's is not perfect; it can fail if request 2 and a block signal generated by request 1 occur simultaneously. The probability of failure is very small if $\Delta >> 2\delta$ and decreases exponentially as Δ is made larger. Moreover, the probability of failure may be mas as small as desired by adding further FRA's in cascade, as in Figure 13b.

Note that, while it appears impossible to design a perfect elementary arbiter that always operates within a fixed time, one can modify the FRA circuit so that each flip flop will respond with grant and block signals only when it is committed to a stable state. In this way a perfect elementary arbiter may be constructed which may require an arbitrarily long time (exponentially distributed) to respond.

C. Computation Schemata

Our research in the theory of computation schemata has the goal of reaching a better understanding of good representations for algorithms -- representations in terms of which determinacy of an algorithm may be readily determined or guaranteed; forms suitable for deriving optimum machine code, or for identifying concurrently executable parts; schemes of representation for which the meaning is readily apparent to the programmer.

We have studied two sorts of parallel computation schemata that model programs and systems involving concurrent transformations and tests on unstructured values. On one hand we have developed a refinement of the parallel program schemata of Karp and Miller (9) and have investigated issues raised by the refined model. In this model the flow of data in a program or system is modeled separately from the sequencing or control. On the other hand are models like the program graphs studied by Rodriguez (16) in which the data flow and control specifications are combined in a single graph. Further development of the ideas of Rodriguez has led to the study of data flow schemata. Recent results from these two directions of research on parallel schemata are reviewed below.

Productivity in Parallel Schemata

In a computation schema it may be that certain actions occurring during a computation have no effect on any output value produced by the computation. In this case we say that

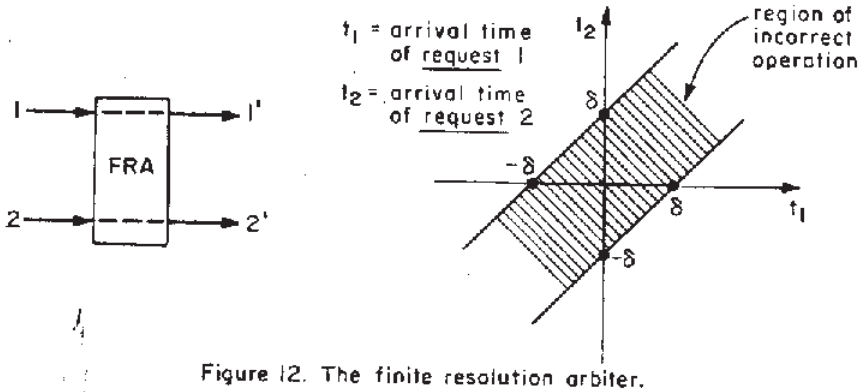
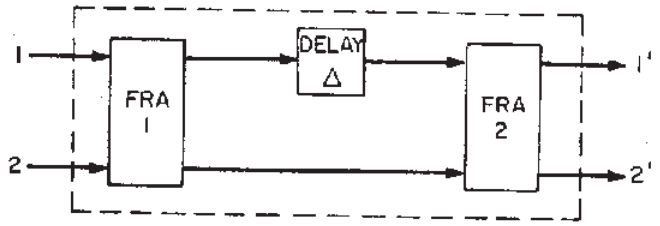


Figure 12. The finite resolution arbiter.

(a)



(b)

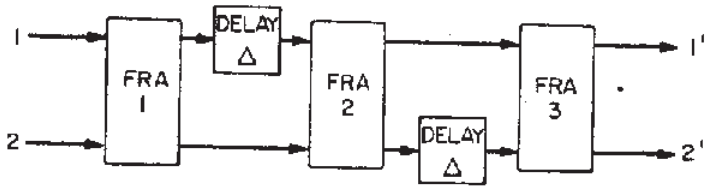


Figure 13. Finite resolution arbiters connected in cascade.

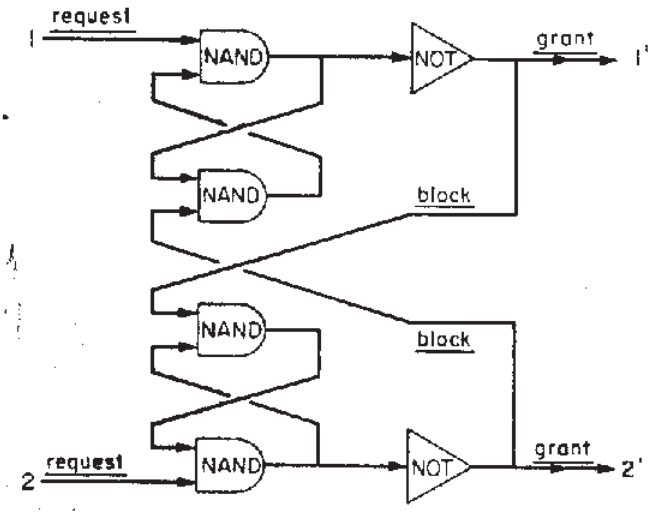


Figure 14. Circuit for finite resolution arbiter.

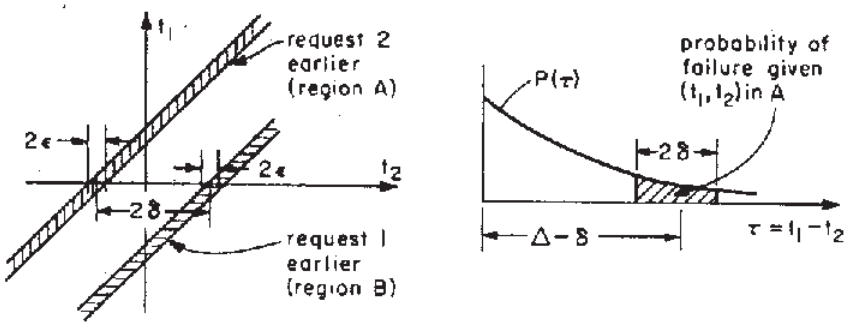


Figure 15. Failure analysis for two FRA's in cascade.

these actions are not productive. We have found there is a trade-off in parallel schemata between productivity and degree of concurrency. That is, to achieve maximum parallelism, it is necessary that the possibility of nonproductive actions be introduced. John Linderman has studied this matter for a class of computation schemata closely related to the parallel program schemata of Karp and Miller (9), and the flow-graph schemata of Slutz (17).

These schemata have separate parts to represent the communication paths for data and the sequencing of actions by operators and decision elements. Since the distinction between "transformations" and "tests" is so pervasive in programming, we feel they should be modeled as different fundamental actions in computation schemata. For this reason, our data flow graphs contain both operators, which model elements that transform values, and deciders, which perform tests with true/false outcomes. Associated with each operator is a function letter, and with each decider a predicate letter. Specific functions and predicates are assigned to the function and predicate letters by an interpretation of the schema. In this way, several operators may be required to perform the same transformation -- or several deciders, the same predicate -- in any interpretation of the schema. This departure from the Karp-Miller model permits treatment of determinacy and equivalence for a broader range of programs and systems.

Each operator and decider has associated initiation and termination events. When an operator or decider initiates, values are read from its ordered set of input memory cells and this vector of values is, in effect, entered into a first-in-first-out queue. Thus multiple initiations of an operator or decider may occur without intervening terminations. When an operator terminates, it writes into its output memory cells the values obtained by applying the function denoted by its function letter to the vector of values taken from the head of the queue. For each decider there are two termination events corresponding to the true and false outcomes of applying the predicate denoted by its predicate letter to the vector of values at the head of its queue.

When and if these events can happen is specified by the control of the schema. A variety of explicit mechanisms have been used to represent the control, including finite state machines, precedence graphs, and Petri nets. These mechanisms share the property that they specify which sequences of events are allowed and which are not allowed as possible behaviors of a schema. The allowed sequences of events are called the control sequences of the schema. Study of various control mechanisms has shown that certain properties of control sequences -- persistence, commutativity, conflict freedom, and repetition freedom -- are central to the study of equivalence, determinacy, parallelism and productivity, regardless of the mechanism used to specify the set of control sequences. For this reason we have studied these properties of schemata without regard to the mechanism used to specify the set of control sequences.

Consider the program below in which w and x are input variables and y and z are output variables:

```
begin
     y := f(g(w))
     if p(w,x) then z := g(f(w)) else z := h(f(w))
end
```

Two schemata for this program are shown in Figure 16. To be definitive, the control sets have been specified by Petri nets. Examples of control sequences for S_1 include

$$\bar{a}\ \bar{t}\ \underline{a}\ \bar{c}\ \bar{b}\ t^T\ \underline{b}\ \bar{d}\ \underline{d}\ \underline{c}$$

$$\bar{t}\ \bar{b}\ t^F\ \underline{b}\ \bar{a}\ \underline{b}\ \underline{a}\ \bar{c}\ \bar{e}\ \underline{e}\ \underline{c}$$

in which overbars and underbars indicate initiation and termination, and the superscript T or F refers to the outcome of a decider.

We identify certain memory cells of a schema as an ordered set of input cells and an ordered set of output cells. Then we may discuss equivalence of two schemata in terms of producing the same output values when given identical inputs. In Figure 16, w and x are the input cells and y and z the output cells of both S_1 and S_2. It is easy to see that, in either schema, any allowed sequence will assign the same values to cells y and z as are produced by the program. Hence both schemata are "functionally determinate" and are equivalent with respect to the specified input and output cells.

In these schemata, an issue arises that is not present when every termination event puts a value in some memory cell and all cell histories affect the question of equivalence, as in the Karp-Miller theory. It is now possible for operators and deciders to be invloved in "useless activity." For example, if y were not an output cell of schema S_1 or S_2, operators a and c would not be productive. Similarly if the same sequence of actions followed either outcome of a decider, then that action of the decider would not be productive.

The precise formulation of this notion of productivity requires formaliams we do not wish to develop here, but the central idea is fairly straightforward. A use of an operator in a control sequence is productive if subsequent actions by operators "carry its result" to a schema output cell or to a productive decider. Since an action by a decider does not directly affect contents of memory cells, determining its productivity is not as easy. We consider a use of a decider to be productive if the schema has two control sequences that define inequivalent computations, and are in "disagreement about decider outcomes" only at the given decider use. For example, consider the program

(a) schema S_1 :

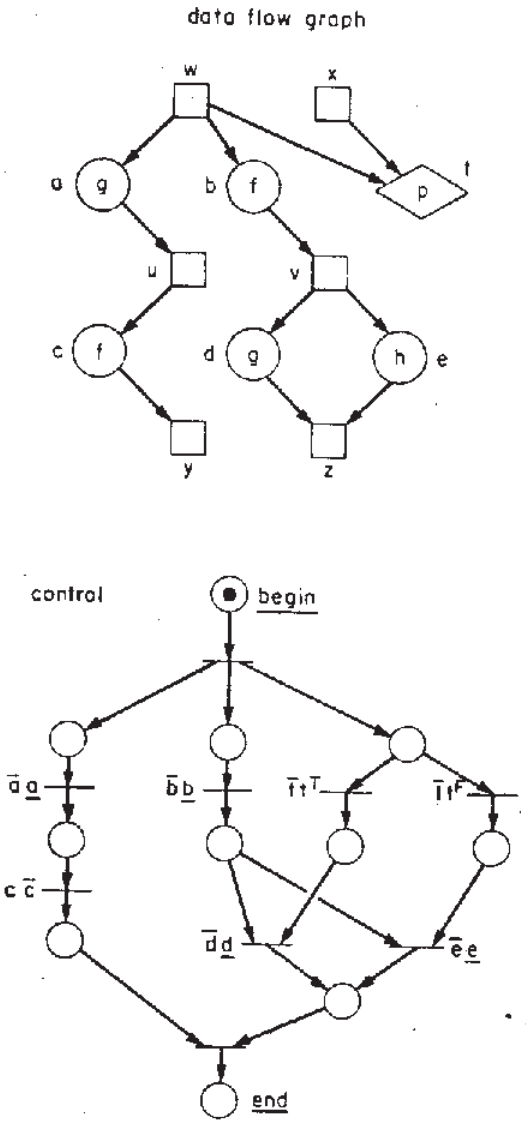


Figure 16. Two equivalent computation schemata.

(b) schemo S_2 :

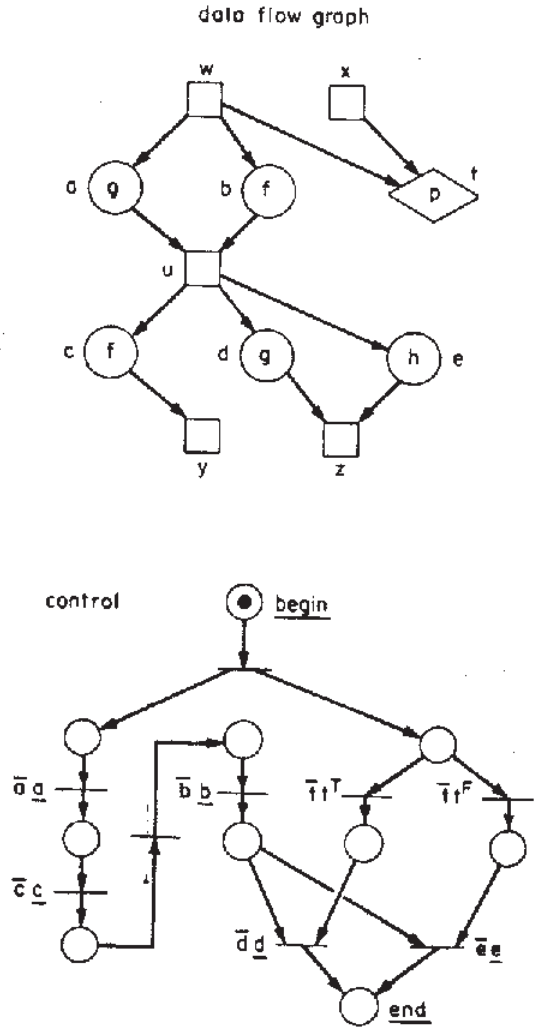


Figure 16. (Continued).

```
begin
    if p(x) then
        if q(x) then y := f(x) else y := g(x)
    else
        if q(x) then y := f(x) else y := g(x)
end
```

Since output y may be set to f(x) if p(x) is false, and g(x) if p(x) is true, one might conclude that this use of p is productive. However, both possibilities exist in either case, the choice being determined by q(x); hence p(x) is not really productive, in agreement with our definition.

Much of this research has been directed toward identifying the most appropriate definitions for "productive control sequences". A seemingly desirable condition is that every use of an operator or decider in a control sequence be productive. Unfortunately, this strong productivity condition limits the degree of parallelism that can be realized. Suppose a sequence must be performed if either of two tests produces true as a result:

```
begin
    if p(x) or q(x) then y := f(x) else y := g(x)
end
```

As soon as either p(x) or q(x) is found to be true, evaluation of the other is unproductive. Thus parallel evaluation of p(x) and q(x) will violate the strong productivity condition. We are studying a weaker form of productivity which does not clash with parallelism.

Data Flow Schemata

An example of a data flow schema is shown in Figure 17. It is a directed graph having two kinds of nodes: actor nodes and link nodes. The arcs of a data flow schema are paths through which data and control values flow from actor nodes to link nodes and from link nodes to actor nodes. Link nodes serve to distribute values to several actor nodes, and are of two kinds -- data links drawn as small solid circles for data values, and control links drawn as small open circles for control values. Certain data link nodes are the input nodes of the schema, and certain data link nodes are the output nodes of the schema. Each link node, except the input nodes, has exactly one incident arc, and all but the output nodes have at least one emanating arc.

There are five kinds of actor nodes:

operator	square box with a function letter written inside.
decider	diamond box with a predicate letter written inside.
true gate/false gate	circle with T or F written inside.
merge	ellipse with T and F written inside.
Boolean	square box with one of the symbols $\wedge$, $\vee$, $\neg$ written inside.

Each arc leaving a link node acts like a first-in-first-out queue for values waiting for use by the actor on which the arc terminates. A value arriving at a link node is replicated as required and entered in the queues of the emanating arcs. In most cases, each queue will either be empty or hold one value. However, permitting unbounded queues permits operation of a data flow schemata to achieve a kind of maximum parallelism we shall illustrate by a later example.

Given a data flow schema and an interpretation of its function and predicate letters, computations by the schema are described by sequences of actions by the actor nodes, analogous to the firing sequences of a Petri net. An operator, decider, or Boolean node is enabled to act when at least one value is available from each of its input arcs. When enabled, one of these actors may "fire" by removing one value from each input queue, applying the specified function, predicate or Boolean operator, and sending the results to its output data or control link. A true gate is enabled by the availability of a data value and a control value from its input arcs. The gate fires by removing these values from their queues. Then, if the control value is true the data value is sent to the output data link; if the control value is false no further action takes place. The false gate acts in an analogous manner. A merge node acts by transmitting a value from its F-input arc if the control input value is false, or a value from its T-input arc if the control value is true. The filled-in arrows on certain control links indicate that a false value is entered in their queues in the initial configuration of the schema. This arrangement is needed to initiate action by a portion of a data flow schema that performs an iteration.

According to these rules of behavior, every actor of a data flow schema is persistent: once enabled an actor becomes not enabled only by firing. From this fact and the discipline by which actor and link nodes interact, a result of Patil (12) shows that any data flow schema is a determinate system.

Study of the schema in Figure 17 reveals that it is equivalent to the following "while schema":

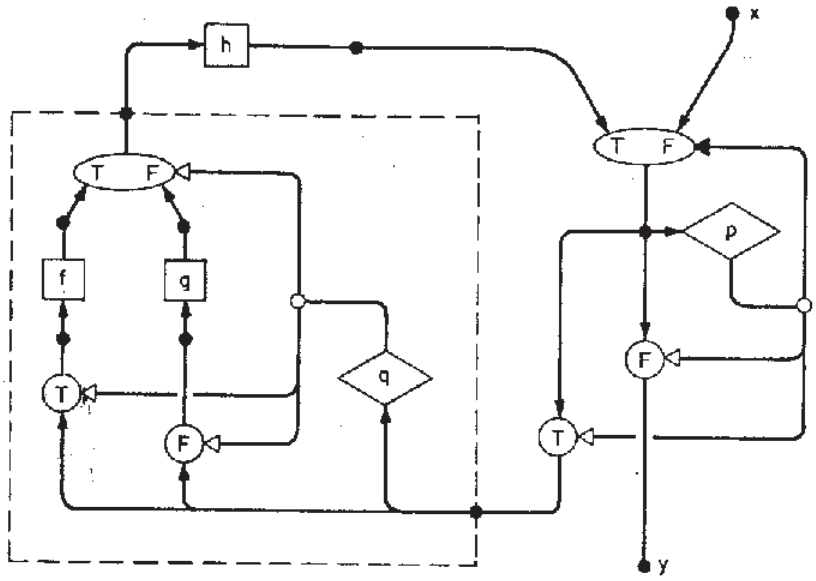


Figure 17. A well-formed data flow schema.

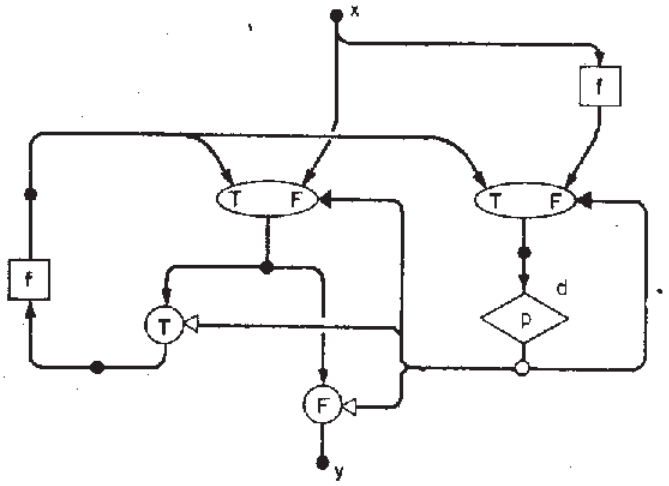


Figure 18. A data flow schema that is not free.

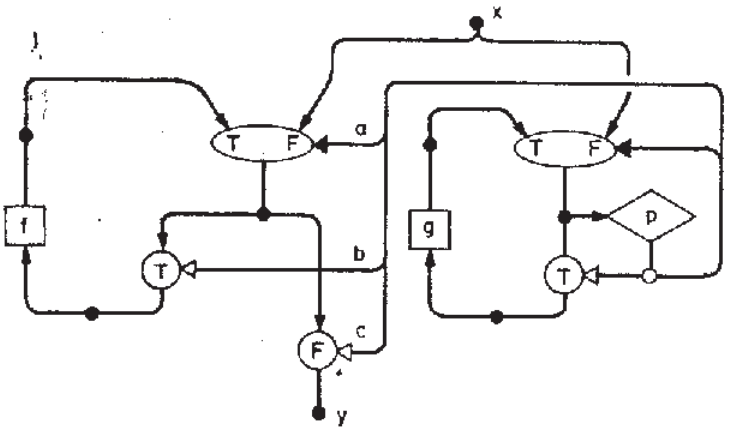


Figure 19. A data flow schema requiring unbounded queues.

```
begin
while p(x) do
    if q(x) then w := f(x) else w := g(x)
    x := h(w)
    end
y := x
end
```

Just as in a while schema, the data flow schema has a nested structure indicated by the dashed lines, and uses specific configurations of gate, merge and decider nodes to form conditional and iteration subschemas. A data flow schema having this structure is said to be well formed. Any well formed data flow schema will generate exactly one value at each output node for each set of values presented at the input nodes. Because it is determinate, any well-formed data flow schema determines a functional dependence of output values on input values. We consider two schemas to be equivalent if both define the same functional dependence of outputs on inputs, and this is true regardless of the interpretation chosen for the function and predicate letters.

On the basis of work by Ashcroft and Manna (1) one can construct a well-formed data flow schema equivalent to any "goto program" or any program schema of the type studied by Paterson (11). Hence the general equivalence problem for data flow schemata is unsolvable.

It has been found that the theory of "free" schemata is more rewarding in terms of positive results than the study of unrestricted schemata. A data-flow schema is said to be free if no two actions by deciders apply the same predicate to the same value. Figure 18 illustrates a schema that is not free because the first two uses of decider d both apply predicate p to the result of applying f to the schema input value. Hence there is no way for the iteration subschema to perform exactly one execution of its body.

John Fosseen (6) has found it possible to transform free data flow schemata in such a way that any pair of data arcs may be tested for equivalence. (Two arcs are equivalent if they pass the same sequence of data values in any computation.) We hope the concepts developed to obtain this result will provide further insight into the equivalence problem for free data flow schemata.

We remarked earlier that treating the input arcs of actors as unbounded queues permits greater concurrency. The data flow schema in Figure 19 illustrates such a case and is based on an example of Keller (10). The right-hand portion of the schema may run arbitrarily ahead of the left-hand portion, a true value being entered in the queues of arcs a, b, and c for each cycle.

The left-hand part may operate as fast as it can until the queues are emptied, whereupon (to be strongly productive) operation must wait for further decisions to be made.

Any data flow schema is inherently maximally parallel in the sense that each operator and decider is at work whenever values are available for some productive use of the operator or decider.

Weakly Productive Computations

In a data flow schema, actions are initiated when the required input values are present and the action (in most cases) is known to be productive. As an interesting exploratory study, we have studied properties of parallel computations in which every operation is initiated as soon as its input values have been computed, so long as some possible continuation of the computation makes productive use of the result. Consider the data flow schema in Figure 20, which represents the following program with input variable x and output variable y:

```
begin
while p(x) do
    if q(x) then x := f(x) else x := g(x)
y := x
end
```

If execution of this schema is performed according to the rules given earlier, then every action by the operators (a and b) and deciders (d and e) is productive. Let us consider what happens if we allow all weakly productive actions to initiate. Suppose termination of the first uses of deciders d and e is arbitrarily delayed. Since the first uses of operators a and b require only the initial value of x, these uses are immediately initiated. Their terminations produce values that are inputs to further weakly productive uses of operators a and b, and so on. These actions define the unbounded tree of values illustrated in Figure 21a; the tree has a node for each value any computation by the schema could generate. As outcomes of decider actions become known, portions of the tree of values become useless and may be deleted, since the operator uses that produce these values become known to be nonproductive. For example, if the first use of decider d yields false, the tree of possibly useful values is as in Figure 21b, and if deciders d and e have successive outcomes F,T and T,T,F, respectively, the tree becomes that in Figure 21c, and represents a completed computation.

Joseph Qualitz (15) has studied the bookeeping requirements for weakly productive computations, and has devised execution structures in terms of which the detailed progress of such computations may be studied. Clearly it is necessary to tag each value produced by a schema operator with the assumptions made

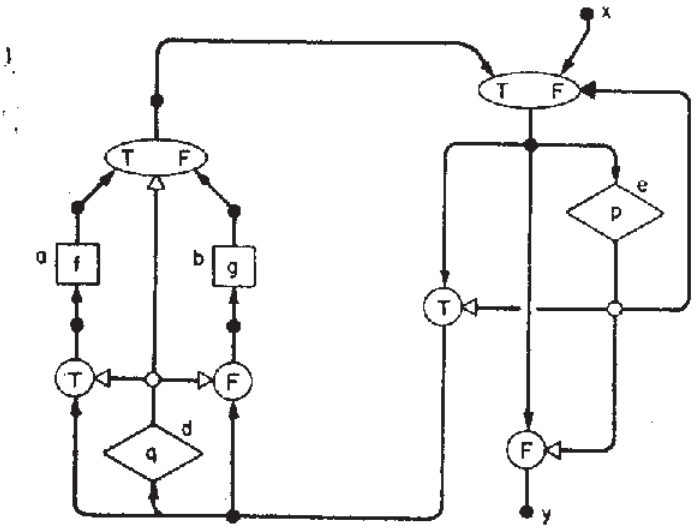


Figure 20. Data flow schema.

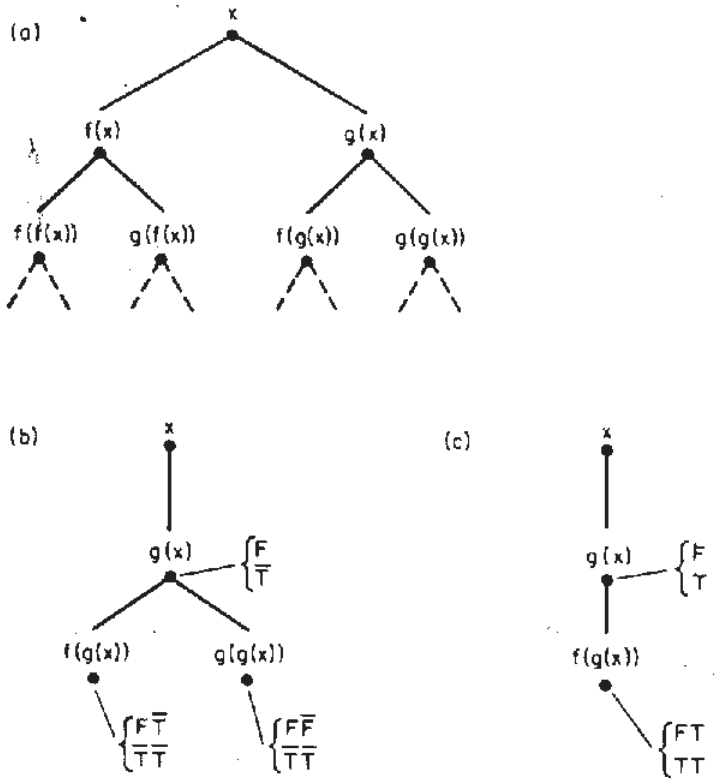


Figure 21. Value trees from a weakly productive computation.

about decider outcomes. We let each value carry a color which is a set of sequences of the symbols $\{T, F, \overline{T}, \overline{F}\}$, one sequence for each decider of the schema. The letters without overbars denote known outcomes, whereas letters with overbars denote assumed outcomes. In Figures 21b, 21c, colors are shown for each value.

At any stage in a weakly productive computation, many values may be associated with certain value nodes of a schema. It is not useful to order these sets of values because, unlike normal execution of a data flow schema, the order in which values arrive is not necessarily the order in which they are used. Instead, each value node is regarded as holding a pool of values, each tagged with the appropriate color, and available for use. Therefore, when an operator or decider has several input value nodes, some means must be provided for identifying the combinations of values to which a function or predicate should be applied. This is done by associating with each value an index that is distinct for each cycle of any loop in the schema. Finally, when a decision is made, certain values become useless and further initiation of actions that use these values must be inhibited.

We have devised rules of execution for weakly productive computations and have shown that these rules correctly simulate the computations of any well-behaved data flow schema.

D. Inductive Proofs of Program Properties

One of the purposes of studying schemata or simplified programming languages is to isolate aspects of programs which must be encompassed by any approach to the construction of formal proofs about the functions computed by programs. Recursion is one such property. To prove equivalence or correctness results about recursive programs, some form of argument by induction must be made. This has been recognized by many people and several of them have formulated induction rules to be used for particular classes of programs. Generally, a program can be viewed as falling in several of these classes. By examining a single program and proofs about it from different viewpoints we have been able to clarify the relationships among these various proof techniques. By means of a simple example we shall illustrate the work of Irene Greif (7) on relating the different ways of interpreting a recursive definition and the corresponding proof techniques.

Consider the following definition of a function f over the nonnegative integers:

$$f(m,n) \equiv \underline{\text{if}}\ n = 0\ \underline{\text{then}}\ m\ \underline{\text{else}}\ f(m + 1, n - 1)$$

(The reader should convince himself that $f(m, n) = m + n$.) The first and most obvious interpretation of the definition is that it describes an algorithm for computing f. The algorithm is to test for $n = 0$; if $n = 0$ then $f(m, n) = m$; otherwise apply the same algorithm in computing $f(m + 1, n - 1)$ to obtain the result. A second interpretation depends on the existence of an ordering on the domain of the function. In this case

the pairs of integers (m,n) can be ordered as follows:

$$(m_1, n_1) \textcircled{<} (m_2, n_2)$$

if and only if

$$n_1 < n_2.$$

Then the definition of f is an inductive definition. The base of the definition is:

For all m $\quad f(m, 0) = m.$

The induction step is:

$$f(m, n) = f(m + 1, n - 1).$$

The third interpretation of f is as the minimal fixpoint of the following functional:

$$\mathcal{C}(X) \equiv \lambda m.\lambda n.\ \underline{\text{if}}\ n \neq 0\ \underline{\text{then}}\ m\ \underline{\text{else}}\ X(m + 1, n - 1)$$

It can be shown that the minimal fixpoint of $\mathcal{C}$ is $\bigcup_{i=0}^{\infty} \mathcal{C}^i(\Omega)$ where Ω is the function that is everywhere undefined and $^{i}(X)$ means the function produced by i applications of $\mathcal{C}$ to X. Notice that $\mathcal{C}(\Omega) = \lambda m.\lambda n.\ \underline{\text{if}}\ n \neq 0\ \underline{\text{then}}\ m\ \underline{\text{else}}\ \Omega(m + 1, n - 1)$ is the function which is m for (m, 0) and undefined for all other ordered pairs. $\mathcal{C}^2(\Omega)$ has the value m for the ordered pair (m, 0) and m + 1 for the ordered pair (m, 1) and is otherwise undefined. Proceeding in this manner, the function f which we are expecting will be generated.

The last interpretation is that the function f represents the agreement of its "truncations." These truncations are the partial functions defined as follows:

$$f_i(m, n) = \underline{\text{if}}\ n = 0\ \underline{\text{then}}\ m\ \underline{\text{else}}\ f_{i-1}(m + 1, n - 1).$$

The reader should note that in this case

$$f_i(m, n) = \mathcal{C}^i(\Omega)\ (m, n).$$

Now we will give four different proofs of the following simple fact:

$$f(m + 1, n) = f(m, n) + 1.$$

The first, by recursive induction, corresponds to the notion of definition by algorithm. We show that f(m + 1, n) and f(m, n)+1 can be computed by exactly the same algorithm by showing that they can be expressed in the same form, namely:

$X(m, n) = \underline{\text{if}}\ n = 0\ \underline{\text{then}}\ m + 1\ \underline{\text{else}}\ X(m + 1, n - 1)$

1. $g_1(m, n) = f(m + 1, n)$

$= \underline{\text{if}}\ n = 0\ \underline{\text{then}}\ m + 1\ \underline{\text{else}}\ f(m + 2, n - 1)$

$= \underline{\text{if}}\ n = 0\ \underline{\text{then}}\ m + 1\ \underline{\text{else}}\ g_1(m + 1, n - 1)$

2. $g_2(m, n) = f(m, n) + 1$

$= (\underline{\text{if}}\ n = 0\ \underline{\text{then}}\ m\ \underline{\text{else}}\ f(m + 1, n - 1)\ ) + 1$

$= \underline{\text{if}}\ n = 0\ \underline{\text{then}}\ m + 1\ \underline{\text{else}}\ f(m + 1, n - 1) + 1$

$= \underline{\text{if}}\ n = 0\ \underline{\text{then}}\ m + 1\ \underline{\text{else}}\ g_2(m + 1, n - 1)$

This shows that $g_1 = g_2$ on the domain of X. If we are trying to prove $g_1 = g_2$ for the pairs of nonnegative integers, a separate proof about the domain of X will be required.

Another proof can be written, utilizing the partial ordering on the domain of these functions, and the inductive definition. The basis of this proof by structural induction is:

for all m

$f(m + 1, 0) = \underline{\text{if}}\ 0 = 0\ \underline{\text{then}}\ m + 1\ \underline{\text{else}}\ f(m + 2, 0 - 1)$

$= m + 1$

$f(m, 0) + 1 = \underline{\text{if}}\ 0 = 0\ \underline{\text{then}}\ m + 1\ \underline{\text{else}}\ f(m+1, 0-1) + 1$

$= m + 1$

Therefore, for the minimal element in the domain, $f(m + 1, n) = f(m, n) + 1$. The induction step, for (m, n), $n \neq 0$ is:

Assume for (m, n), $n < N$ that $f(m + 1, n) = f(m, n) + 1$

1. $f(m + 1, N) = f(m + 2, N - 1)$

2. $f(m, N) + 1 = f(m + 1, N - 1) + 1$

$= f(m + 2, N - 1)$ by induction since $N-1 < N$.

The initial assumption, based on the means of definition of the function is that f is total on the ordered pairs, partially ordered by $(<)$. From this fact and the above proof, we know that $f(m + 1, n) = f(m, n) + 1$ for all pairs of nonnegative integers.

The third proof is actually simple induction on the depth of recursion of a computation. In terms of the definition of the minimal fixpoint

$$f = \bigcup_{i=\infty} \mathcal{C}^i(\Omega),$$

computational (or μ-rule) induction is simple induction on i.

1. for i = 0 we must show

   $\Omega(m + 1, n) = \Omega(m, n) + 1$

   Obviously both are totally undefined.

2. Assume $X(m + 1, n) = X(m, n) + 1$

   then prove $\mathcal{C}(X)(m + 1, N) = \mathcal{C}(X)(m, n) + 1$

$$\begin{aligned}
\mathcal{C}(X)(m + 1, n) &= \underline{\text{if}}\ n = 0\ \underline{\text{then}}\ m+1\ \underline{\text{else}}\ X(m+2, n-1) \\
&= \underline{\text{if}}\ n = 0\ \underline{\text{then}}\ m+1\ \underline{\text{else}}\ X(m+1, n-1) + 1 \\
&\qquad \text{(by induction)} \\
&= (\underline{\text{if}}\ n = 0\ \underline{\text{then}}\ m+1\ \underline{\text{else}}\ X(m+1, n-1)\ )+1 \\
&= \mathcal{C}(X)(m, n) + 1
\end{aligned}$$

This proves that f(m, n) = f(m, n) + 1 are totally equivalent, i.e., either both are undefined or both are defined and have the same value.

A separate argument can easily be given to show that both functions are defined for all pairs of nonnegative integers.

The last proof technique is very similar to computational induction, being course-of-values induction on the index i of the truncations of a function. This amounts to doing course-of values induction on the depth of recursion. For our particular example, in which equivalence depends only one one step in the computation of the fixpoint, the difference between the two proofs is strictly a matter of formalism. We prove that for $i = 0$, $f_0(m + 1, n) = f_0(m, n) + 1$.

Then for $i \neq 0$:

assume for $j < i$, $f_j(m + 1, n) = f_j(m, n) + 1$

$$\begin{aligned}
f_i(m + 1, n) &= \underline{\text{if}}\ n = 0\ \underline{\text{then}}\ m+1\ \underline{\text{else}}\ f_{i-1}(m+2, n-1) \\
&= \underline{\text{if}}\ n = 0\ \underline{\text{then}}\ m+1\ \underline{\text{else}}\ f_{i-1}(m+1, n-1)+1 \\
&= (\underline{\text{if}}\ n = 0\ \underline{\text{then}}\ m+1\ \underline{\text{else}}\ f_{i-1}(m+1, n-1)\ )+1 \\
&= f_i(m, n) + 1
\end{aligned}$$

As in the last proof, this shows strong equivalence, this time by truncation induction.

Generally, any method can be used for a proof. If the programmer had one of these interpretations in mind in writing his program, then the corresponding proof technique will probably seem most natural. Ideally an automatic program verifier would be flexible with respect to choice of induction rule. It is unlikely, however, that all of these will be equivalently useful in mechanical proofs, even though there seems to be no real difference in scope of application among them.

E. A Computer for General Data Types

One goal in the design of programming systems is to retain the generality of an algorithm when it is encoded into the language of the programming system. A serious limitation on the generality readily achieved in contemporary computer systems is imposed by the fixed word length and finite size of computer memories.

In preparing a program for execution by a computer system, the programmer first imagines the abstract function the program is to implement. Simple examples might be to implement the scalar product of any two n-component vectors of real numbers, or to obtain the greatest common divisor of two integers. As in these examples, the abstract function almost always has an infinite domain. Then the programmer conceives of an algorithm for the function -- a step-by-step process for obtaining the value of the function through the use of idealized primitive operations such as the arithmetic operations on integers and reals. The next step is to express the algorithm in the language of some practical programming system. Usually the actual data types of the programming system have their idealized counterparts, and, if the language is suited to the needs of the algorithm, the algorithm may be converted into a program with little difficulty. Our problem of generality would be solved if the task of the programmer were completed at this point. However, he must now check whether the word size and finite memory size of the computer system, as reflected in defects of the primitive operations of the programming language, may prevent correct operation of his program. In many cases, the program will operate correctly for a large (but finite) number of points in the domain of the abstract function, and will fail (often without any hint to the user) for the remaining (infinite) set of domain points. In other cases the programmer will find that the number of cases for which the program will work correctly is too small to be of interest and a new approach, using a language less suited to expressing the algorithm, or less efficient in execution, must be adopted.

The ability of a programming system to correctly execute programs expressed in terms of idealized data types is called generality with respect to domain. Most programming systems fail to be general with respect to domain by limiting the amount of storage that may be allocated to one data value to less than the available memory of the computer on which the

programming system runs. For instance, integers are usually limited in value by the number of bits in one memory word, and the maximum range of an array subscript must often be specified at the moment the array is created.

Since the memory of any practical computer system is, in fact, finite we cannot expect any program to obtain the value of the programmer's abstract function for any point in its domain. However, we should expect a programming system to produce the correct result unless the computer system runs out of memory in trying. (If the computer system runs out of memory, should one blame the program for the absence of sufficient memory to compute the function ?) This consideration is the basis for the following definitions:

Definition: A program p, with input variables $\underline{x} = (x_1, \ldots, x_m)$ and output variables $\underline{y} = (y_1, \ldots, y_n)$, computes a function f over domain D if and only if for each point $\underline{x}$ in D either

1. program p produces output $\underline{y}$ from input $\underline{x}$ where $y = f(x)$, or
2. for input $\underline{x}$ program p fails to complete due to an unsatisfied request for additional storage.

Thus a program that computes a function must obtain the correct result whenever it is given sufficient resources to operate.

Definition: A programming system that implements a language L is general with respect to domain if and only if for any algorithm that defines a function f on domain D, the corresponding program in L computes f on D.

The heart of the problem of implementing programming systems having generality with respect to domain is machine instructions which themselves are programs not general with respect to domain. The basic arithmetic instructions, for example, usually operate on representations that occupy a single register. Since conventional programmed multiple length arithmetic introduces a high cost in time consumed, even for quantities that require only single-length representation, achieving generality for these data types in a conventional computer system is unattractive.

Peter Bishop (3) has designed an abstract computer in which generality with respect to domain is achieved for a large class of data types including integers, floating point numbers, strings and arrays, as well as more elaborate structures. In the abstract computer, each data value is represented by a pointer-linked tree structure having as many elements as necessary to represent the value. The representation of any quantity may expand arbitrarily as required until available memory is exhausted.

References

1. Ashcroft, E., and Manna, Z., The translation of 'go to' programs to 'while' programs. Information Processing 71, Ljubljana, 1971.

2. Baker, H., Petri Nets and Languages. Computation Structures Group Memo 68, Project MAC, M.I.T., May 1972.

3. Bishop, P. B., Data Types for Programming Generality. S.B. and S.M. Thesis, Department of Electrical Engineering, M.I.T., June 1972.

4. Chaney, T. J., Ornstein, S. M. and Littlefield, W. J., Beware the synchronizer. Proceedings of the Sixth Annual IEEE Computer Society International Conference, San Francisco, September 1972, pp 317-319.

5. Commoner, F., Holt, A. W., Even, S., and Pnueli, A. Marked directed graphs. J. of Computer and System Sciences, Vol. 5 (1971), pp 511-523.

6. Fosseen, J. B., Representation of Algorithms by Maximally Parallel Schemata. S.M. Thesis, Department of Electrical Engineering, M.I.T., June 1972.

7. Greif, I. G., Induction in Proofs About Programs. S.M. Thesis, Department of Electrical Engineering, M.I.T., December 1971.

8. Holt, A. W. and Commoner, F. Events and Conditions. (In three parts), Applied Data Research, New York 1970. (Chapters I, II, IV and VI appear in Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York 1970, pp 3-52.)

9. 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.

10. Keller, R. M., On maximally parallel schemata. IEEE Conference Record. Eleventh Annual Symposium on Switching and Automata Theory, 1970, pp 32-50.

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

12. Patil, S. S., Closure properties of interconnections of determinate systems. Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York 1970, pp 107-116.

References (continued)

13. Patil, S. S., Forward Acting n x m Arbiter. Computation Structures Group Memo 67, Project MAC, M.I.T., June 1972.

14. Plummer, W. W., Asynchronous Arbiters. IEEE Trans. on Computers, Vol. C-21, No. 1 (January 1972)

15. Qualitz, J. E., Weakly Productive Computation Schemata. S.B. and S.M. Thesis, Department of Electrical Engineering, M.I.T., May 1972.

16. Rodriguez, J. E., A Graph Model for Parallel Computations. Technical Report TR-64, Project MAC, M.I.T., September 1969.

17. Slutz, D. R., The Flow Graph Schemata Model of Parallel Computation. Technical Report TR-53, Project MAC, M.I.T., September 1968.

COMPUTATION STRUCTURES

Publications

1. Baker, H., "Petri Nets and Languages", Computation Structures Group Memo 68, May 1972.

2. Dennis, J. B., "Management of Names in a Computer System", Computation Structures Group Memo 63, November 1971.

3. Dennis, J. B., "Concurrency in Software Systems", Computation Structures Group Memo 65-1, June 1972.

4. Dennis, J. B., "The Design and Construction of Software Systems", Computation Structures Group Memo 69, June 1972.

5. Dennis, J. B., "Modularity", Computation Structures Group Memo 70, June 1972.

6. Dennis, J. B., On the Design and Specification of a Common Base Language, Project MAC, M.I.T., MAC TR-101, June 1972, AD 744-207.

7. Fano, R. M., "On the Number of Bits Required to Implement an Associative Memory", Computation Structures Group Memo 61, August 1971.

8. Flinker, E., "Translation of a Block Structured Language Into the Common Base Language", Computation Structures Group Memo 66, January 1972.

9. Fox, P. J., "A Look at 'The Controlled Execution of Parallel Programs Operating on Structured Data' by Ian Campbell-Grant", Computation Structures Group Memo 62, October 1971.

10. Greif, I. G., Induction in Proofs about Programs, Project MAC, M.I.T., MAC TR-93, February 1972, AD 737-701.

11. Hack, M. H. T., Analysis of Production Schemata by Petri Nets, Project MAC, M.I.T., MAC TR-94, February 1972, AD 740-320.

12. Lester, B. P., Cost Analysis of Debugging Systems, Project MAC, M.I.T., MAC TR-90, September 1971, AD 730-521.

13. Patil, S. S., "Forward Acting n x m Arbiter", Computation Structures Group Memo 67, June 1972.

Talks

1. Dennis, J. B., and S. S. Patil, "Systematic Realization of Asynchronous Systems", IEEE Seminar, Boston, September 8, 1971.

2. Dennis, J. B., "Design of a Common Base Language"; and "A Data Flow Model of Computation", Tutorial Symposium on Semantic Models of Computation, New Mexico State University, Las Cruces, New Mexico, January 3-5, 1972.

3. Dennis, J. B., "The Design and Construction of Software Systems"; "Modularity"; and "Concurrency in Software Systems", Advanced Course on Software Engineering, Technical University of Munich, Munich, Germany, February 21 - March 4, 1972.

4. Dennis, J. B., "On the Design and Specification of a Common Base Language", talk at the General Electric Co., Schenectady, New York, July 15, 1972.

Theses Completed

1. Bishop, P. B., Data Types for Programming Generality, S.B. and S.M. Thesis, Department of Electrical Engineering, M.I.T., June 1972.

2. Fosseen, J. B., Representation of Algorithms by Maximally Parallel Schemata, S.M. Thesis, Department of Electrical Engineering, M.I.T., June 1972.

3. Furtek, F. C., Modular Implementation of Petri Nets, S.M. Thesis, Department of Electrical Engineering, M.I.T., September 1971.

4. Greif, I. G., Induction in Proofs about Programs, S.M. Thesis, Department of Electrical Engineering, M.I.T., February 1972.

5. Hack, M., Analysis of Production Schemata by Petri Nets, S.M. Thesis, Department of Electrical Engineering, M.I.T., February 1972.

6. Qualitz, J. E., Weakly Productive Computation Schemata, S.M. Thesis, Department of Electrical Engineering, M.I.T., May 1972.

Theses in Progress

1. Amerasinghe, S. N., "The Handling of Procedure Variables in a Base Language", S.M. Thesis.

2. Fox, P. J., "Representation of Parallel Computation on Data Structures", S.M. and E.E. Thesis.

3. Hawryszkiewycz, I. T., "The Semantics of Data Base Systems", Ph.D. Thesis.

Talks

1. Dennis, J. B., and S. S. Patil, "Systematic Realization of Asynchronous Systems", IEEE Seminar, Boston, September 8, 1971.

2. Dennis, J. B., "Design of a Common Base Language"; and "A Data Flow Model of Computation", Tutorial Symposium on Semantic Models of Computation, New Mexico State University, Las Cruces, New Mexico, January 3-5, 1972.

3. Dennis, J. B., "The Design and Construction of Software Systems"; "Modularity"; and "Concurrency in Software Systems", Advanced Course on Software Engineering, Technical University of Munich, Munich, Germany, February 21 - March 4, 1972.

4. Dennis, J. B., "On the Design and Specification of a Common Base Language", talk at the General Electric Co., Schenectady, New York, July 15, 1972.

Theses Completed

1. Bishop, P. B., Data Types for Programming Generality, S.B. and S.M. Thesis, Department of Electrical Engineering, M.I.T., June 1972.

2. Fosseen, J. B., Representation of Algorithms by Maximally Parallel Schemata, S.M. Thesis, Department of Electrical Engineering, M.I.T., June 1972.

3. Furtek, F. C., Modular Implementation of Petri Nets, S.M. Thesis, Department of Electrical Engineering, M.I.T., September 1971.

4. Greif, I. G., Induction in Proofs about Programs, S.M. Thesis, Department of Electrical Engineering, M.I.T., February 1972.

5. Hack, M., Analysis of Production Schemata by Petri Nets, S.M. Thesis, Department of Electrical Engineering, M.I.T., February 1972.

6. Qualitz, J. E., Weakly Productive Computation Schemata, S.M. Thesis, Department of Electrical Engineering, M.I.T., May 1972.

Theses in Progress

1. Amerasinghe, S. N., "The Handling of Procedure Variables in a Base Language", S.M. Thesis.

2. Fox, P. J., "Representation of Parallel Computation on Data Structures", S.M. and E.E. Thesis.

3. Hawryszkiewycz, I. T., "The Semantics of Data Base Systems", Ph.D. Thesis.