MAC TR-111

PRODUCTIVITY IN PARALLEL COMPUTATION SCHEMATA

John P. Linderman

December 1973

This research was supported by the National Science Foundation in part under research grant GJ-432, and in part under research grant GJ-34671.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

PRODUCTIVITY IN PARALLEL COMPUTATION SCHEMATA

BY

John Parent Linderman

Submitted to the Department of Electrical Engineering
on May 21, 1973, in partial fulfillment of
the requirements for the Degree of Doctor of Philosophy

ABSTRACT

A general model for parallel computation is developed in three parts. One part, the data flow graph, describes how actors which transform and test values are connected to the locations in a finite memory. Another part, an interpretation, supplies information about the contents of memory and the detailed nature of the transformations and tests.

The third part specifies how initiations and terminations of the actors are allowed to occur. We define this in a general way, using a set of sequences of initiation and termination events to model control. This allows us to prove results which apply to a broad class of control mechanisms.

Our major results are analogous to a theorem of Karp and Miller. Their theorem defines a class of schemata for which conflict-freeness is necessary and sufficient for determinacy. We use a weaker notion of determinacy which depends only upon the final contents of a subset of the memory locations. To establish necessity, we introduce the property of productivity which expresses whether individual transformations and tests contribute to the final results of a computation.

THESIS SUPERVISOR: Jack B. Dennis
TITLE: Professor of Electrical Engineering

ACKNOWLEDGEMENTS

I would like to thank my thesis supervisor, Jack Dennis, for introducing me to schemata, guiding my research, and carefully reading drafts of this dissertation. I appreciate his interest and editorial comments.

My thanks to my readers, Carl Hewitt and Al Meyer, for taking the time to go over drafts of dubious literary merit.

A collective 'Thank You' to the people at Project MAC and particularly the members of the Computational Structures Group. They contributed in subtle but important ways to make the pursuit of this research a genuine pleasure.

June, 1973 John P. Linderman

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF THEOREMS AND LEMMAS

1.1.1 A Note on Mathematical Style

It is conventional to use single letters to identify mathematical objects such as functions, sets and relations. The advantage is primarily one of conciseness. Unfortunately, this may come at the expense of understandability, and almost certainly with a loss of readability. The number of formal objects which will be introduced in the course of this dissertation would quickly exhaust the alphabets commonly available on typewriters. Worse still, the mnemonic value would be vanishingly small.

We will therefore abandon the single letter convention and freely use words or even phrases to identify formal objects. As often as not, the name we choose for a set will simply be the plural form of the elements it comprises. Thus, we will not feel obliged to mention that SCHEMA INPUTS is a set whose elements are, in fact, schema inputs. As a means of avoiding unintended confusion, we will capitalize formal names. We will use these conventions throughout with the hope that they will enlighten rather than confuse.

When mathematical precision conflicts with clarity or perspective in our definitions or proofs, we will favor the latter. For example, we will talk about "edges labeled with numbers" rather than formalize a "function from the set of edges into the natural numbers." In all cases, it should be clear how the proofs and definitions could be made more rigorous if so desired.

A glossary can be found at the end of this dissertation for the reader's convenience.

1.2.1 Introduction and Background

This dissertation is, in effect, a discussion of a model for parallel computation. Everything a mathematician needs to know about the model can be found in the subsequent chapters. However, the model is more than a collection of abstract definitions. Its form was influenced by a number of areas in computer science, and a knowledge of these may help the reader to push through some of the abstractions and to feel comfortable with the model.

1.3.1 Schematology

Schemata are models for computation, as are Turing machines, finite state machines, programs written in a programming language, and so forth. The feature which distinguishes schemata is the lack of interpretation for primitive functions and predicates. To logicians, this is a familiar concept. For the rest of us, a common analogue is fortunately available.

Suppose an ordinary compiler runs across a statement such as

$$u \leftarrow f(v,w,x,\cdots,y,z)$$

Assume function f is externally defined. Although the compiler cannot know exactly what f does, it is not entirely in the dark. The compiler at least knows where f will get its inputs and where it will store its result. This may seem to be very little knowledge indeed, but, with a few more assumptions, it allows the compiler to perform transformations such as calling f with <u>copies</u> of the inputs or using u as a temporary work area until f has returned its result. For example, it could substitute something like

$u \leftarrow v$

$u \leftarrow f(u,w,x,\cdots,y,z)$

for the original statement, and not alter the meaning of the program.

The point to be noticed here is that the transformation is valid no matter how f is defined (subject to certain assumptions we will presently discuss.) Schematology is concerned with making statements about computation which remain valid regardless of the definition of the functions and predicates used. From the above discussion, it should be clear that schematology has immediate application in compiler theory.

Although the "external routine" analogue may be useful to those whose background tends more towards computers than towards formal logic, it must not be pushed too far. There are assumptions we make about functions and predicates that are not always true for arbitrary programs.

Stated simply, we assume that all functions and predicates implement total mathematical functions. To emphasize what is <u>not</u> allowed as a function or predicate, some elaboration is in order. A function or predicate name may appear many times in a schema, but it must always take the same number of arguments. Thus, although one could argue for the desirability of a generalized addition function which returns the sum of an arbitrary number of arguments, such constructs will not be allowed in schemata. Functions and predicates are presumed to be defined for all inputs. Thus, no function could precisely model division which is not defined for divisor 0. Given the same set of inputs, functions and predicates must always return the same results. This precludes certain routines with "memory" such as "clock functions" or predicates which are true only the first time they are used. Functions and predicates do not alter their inputs, and have no "side-effects" whatever.

From this discussion, the reader can surmise that the schemata defined herein are not intended to model every computation that might be possible using conventional programming languages. We are, instead, focusing on a limited, but very rich subset of such computations.

In addition to compiler theory, there are other justifications for adopting a schematized model. The Turing machine is one of the best known models for interpreted computation. Hopelessly inefficient as a practical computing device, Turing machines have been most valuable in identifying problems which cannot be systematically solved. For example, we know there is no procedure for determining if an arbitrary Turing machine halts when started on a particular input, or whether two Turing machines compute the same partial function. We cannot find the "fastest" Turing machine which implements an arbitrary function because, in some cases, there is always a faster machine, and one still faster and so on *ad infinitum*. In the face of such undecidability results, we must lower our sights somewhat. Schematalogy sacrifices a certain amount of "relevance" for the possibility of answering questions analogous to the above.

It would seem to be a better approach to go only half-way towards schemata and allow both interpreted and uninterpreted operations. After all, this is precisely the problem a compiler must handle. Unfortunately, the amount of interpretation which can be tolerated without introducing undecidability problems is very small. For example two counters with unlimited capacity are sufficient to mimic Turing machines[10].

1.3.2 Schematology - History

Ianov is generally credited with the first schematized model for computation[15] His model treats all of storage as a monolithic entity which is transformed by operations. The operation to be applied is determined by the outcome of a number of predicates testing the store. Ianov showed that equivalence of his schemata was decidable. Rutledge[15] later pointed out a correspondence between Ianov's schemata and a class of finite state machines.

Luckham, Park and Paterson[9] developed a more familiar schematized model based on flow charts. Operator instructions in the flow charts are of the form

label: $x_i \leftarrow f(x_{j_1},x_{j_2},\cdots,x_{j_m})$

with branching in the charts accomodated by transfer instructions such as

label: $p(x_i)$,tlabel,flabel

By establishing a correspondence with a class of automata, they were able to show that the equivalence problem for schemata using two or more memory locations is not decidable. However, certain restricted classes of flow chart schemata were identified for which equivalence is decidable.

Schemata have also been used by Hewitt and Paterson to allow meaningful comparison of various control mechanisms such as iteration, recursion, and recursion with limited parallelism[12].

1.4.1 Parallelism

Parallel processing as it exists today generally refers to concurrent execution of several processes which are themselves strictly sequential. Although this is an important trend in the efficient use of computer systems, it is not the level of parallelism in which we are interested. We will be focusing on parallelism within a program at the instruction level. Therefore, we are discussing hardware which is not now common and probably will not be common for a generation or so.

There is more than a theoretical difference between concurrent execution of one thousand programs and concurrent use of one thousand processors by a single program. An astronaut does not care if ground control can recompute a thousand courses in an hour. What he requires is one course recomputed in seconds. The number of such real time applications is constantly expanding. With speeds of individual processors approaching limits imposed by the speed of light and the laws of thermodynamics, low level parallelism will take on increasing importance.

In addition to such practical applications, one can make an argument for parallelism on theoretical grounds. A conventional algorithm may impose sequencing constraints which are arbitrary and have nothing to do with the function being implemented. A parallel specification can help to focus on this function by stripping away arbitrary sequencing and leaving only that which is essential.

1.4.2 Parallel Schematology - History

Karp and Miller[7], Slutz[16], and Keller[8] have worked with schemata which model parallel computation. Using a hybrid operation which combines the rôles of transformation and testing, they were able to identify certain classes of parallel schemata for which determinacy and equivalence are decidable. Some of the major results of this dissertation are extensions of the work of Karp and Miller to a model with more conventional operators and a weaker form of equivalence.

Slutz's *maximum parallel form*[16] which maximized the number of possible computations for a given interpretation helped to motivate our notion of *productivity*. With our weaker form of equivalence, the possibility of useless operations arises. A workable analogue to Slutz's maximum parallel form would have to restrict computations to some level of productivity. Lacking this, unproductive activity could be added indefinitely without truly increasing parallelism.

Parallel schemata in which the arrival of data triggers activity have been investigated by Rodriguez[13], Dennis[3] and Fosseen[5]. A desirable feature of these schemata is their inherent determinacy. The equivalence problem remains undecidable for general data flow schemata. However, both Rodriguez and Fosseen discuss decidable questions about equivalence of data links within a schema.

1.5.1 Modularity

Modularity is a term finding great favor both in computer science and on Madison Avenue. Informally, we might say that a modular system is the interconnection of a small number of components whose inputs, outputs and input/output behavior are explicitly defined. Although we have restricted modular systems to comprise only a few components, there is no real restriction on system size. Each component may be a modular system whose components are themselves systems and so forth. We are therefore talking about a style of systems design rather than a class of systems.

Modular systems have the virtue of being easy to debug and easy to modify. If such a system malfunctions, the components can be "unplugged" and tested to see that they meet their input/output specifications. Since this is, by definition, their only rôle in the system, operational components need not be checked in greater detail. This allows quick isolation of problems. Furthermore, any component can be replaced by another with the same input/output behavior. This allows us to take advantage of developments which make modules cheaper, faster, more reliable, or whatever. The value of such capabilities in large systems can scarcely be overestimated.

The definition of equivalent schemata reflects our input/output orientation. We view schemata much as a programmer views subroutines. When given inputs, they either run on forever or they halt and produce some outputs. Equivalence will amount to halting on the same inputs, yielding the same results. Intermediate values, storage used, amount of parallelism exploited and the like will be of no consequence as far as equivalence is concerned.

1.6.1 Speed Independence

Programmers on some early machines observed that they could initiate data input operations before they were finished with the data in the area to be overwritten. They knew there would be ample time to access the data before the relatively slow input devices could effect the transfer. This technique was actually an early form of parallel processing, but a decidedly dangerous one. For example, if the I/O device had been improved or the program simulated rather than executed, chaos could have resulted.

The peculiarities of particular machines and operating systems now make any assumptions about relative speeds highly suspect. Our model will not assume that all operations take the same amount of time or even that a particular operation always terminates after a fixed delay after it has initiated. In fact, our model will make no mention of delays and timing. Instead, each action will have distinct, explicit initiation and termination events. All timing considerations will be modeled by the sequencing of these events. The model will therfore possess enough flexiblity to describe systems where the amount of time needed to complete an operation may be highly variable.

2.1.1 Data Flow Graphs - Introduction

If schemata are a model for computation as we suggested in the introduction, then we should begin with a broad-brush approximation to just what a computation is. Let us informally define a computation as a process which performs transformations and tests on a set of values. This is obviously not to be taken too seriously, but it will serve as a foundation upon which to build.

For example, the definition suggests that a model must have agents capable of transforming values and agents capable of testing values. We refer to these agents as *operators* and *deciders* respectively, and call them *actors* collectively. As we noted in the introduction, with our schematized approach we do not define exactly what these actors do or even the set of values they act upon. However, by associating function names with operators and predicate names with deciders, we can constrain distinct actors to do the same thing, whatever that may be, by giving them the same function name or predicate name.

If an operator performs a transformation of m values, we will depict the operator as a circle with its associated function name inside and with input arcs labeled 1 through m. An operator having no inputs is allowable: it allows us to model constants. Deciders are similar but they will be diamond-shaped and we insist that they have at least one input: they must have something upon which to base a decision. Of course, all actors with which a given name is associated must have the same number of inputs.

We assume that all operators have exactly one output. There is no real loss of generality, since we could always model a transformation with k outputs by k single-output operators. Deciders, on the other hand, have no outputs. They have outcomes, either *true* or *false*. Again, binary deciders can be put together in such a way as to model k-way tests. Figure 2.1 shows an operator and a decider.

In our model, the values input to actors and produced by operators are presumed to reside in memory locations. Which location supplies which input is depicted by attaching the arcs on actors to memory locations. The same location may supply more than one input, and may also be the output location of an operator to which it supplies an input. Some actor/memory interconnections are shown in Figure 2.2.

Labeling the arcs on such interconnections can be tedious and unenlightening. We will adopt the convention that arcs appear in order of increasing index starting from the bottom of the actor and proceding clockwise around it unless explicitly labeled otherwise. Thus, all arc labeling in Figure 2.2 is unnecessary.

Presumably, we institute the series of transformations and tests which we called a computation because we are interested in some results. In general, we expect these results to depend in some way on an initial set of values we provide to the computation. In our actor/memory interconnections, we identify two ordered subsets of the memory locations. One set, the schema inputs, are presumed to hold the initial values when the computation begins. The other set, the schema outputs, will contain the results if and when the computation terminates. The actor/memory interconnection and schema inputs and outputs determine what we will call a *data flow graph*.

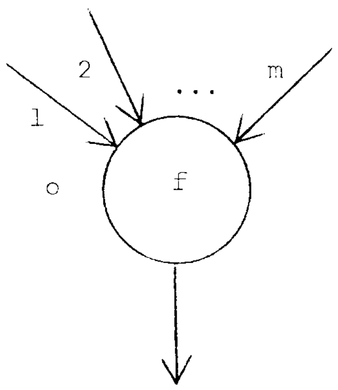


Figure 2.1a An operator o with m-ary function name f

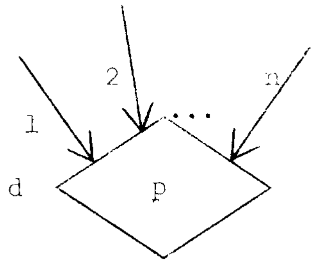


Figure 2.1b A decider d with n-ary predicate name p

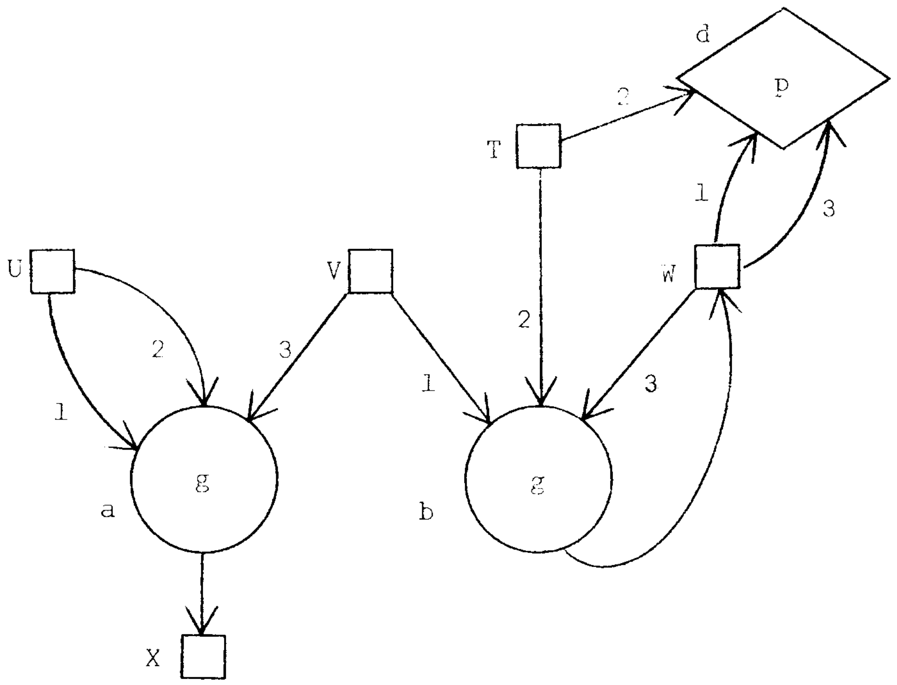


Figure 2.2 An interconnection of actors and memory locations

The class of computations we will be investigating have finite data flow graphs which are not altered during the course of computation. We cannot allocate additional memory locations, change the function name associated with an actor, or move an actor around in the graph. We do not claim to have a satisfactory way of representing structured data such as arrays, lists or stacks. These features are <u>not</u> deemed unimportant, but rather lie outside of the scope of this dissertation.

If the reader experiments with producing a data flow graph corresponding to a familiar computation (as we shall do in Section 1.3), he will probably discover that simple assignment occurs. That is, we often want to simply copy the contents of one location into another. This is one function which is always meaningful, independent of the domain of values with which we are dealing. We therefore allow *identity operators* to be used in data flow graphs. We will reserve the function name ":=" for these operators.

2.1.2 Data Flow Graphs - Formal Definition

A *data flow graph* is a directed, labeled, bipartite graph. One set of nodes is a finite number of *locations* collectively referred to as MEMORY. An ordered (possibly empty) subset of MEMORY is distinguished as the SCHEMA INPUTS, and another ordered (nonempty) subset is distinguished as the SCHEMA OUTPUTS. The other set of nodes are a finite collection of ACTORS, partitioned into OPERATORS and DECIDERS. Associated with each operator is one of a set of FUNCTION NAMES. For simplicity, we will assume that the number of its inputs (≥ 0) is implicit in the function name. If f is an m-ary function name, then any operator o with which f is associated has input arcs labeled 1 through m originating on (not necessarily distinct) locations in MEMORY. We call these locations the *input locations of o*. In particular, we call the location on the arc labeled i the *i^{th} input location of o*. Each operator o has a single output arc terminating on some location in MEMORY (not necessarily disjoint from the input locations of o.) We refer to this as the *output location of o*. Associated with each decider is one of a set of PREDICATE NAMES. As with function names, we will assume that the number of inputs (>0) is implicit in the predicate name. If p is an n-ary predicate name, then any decider d with which p is associated has input arcs labeled 1 through n originating on (not necessarily distinct) locations in MEMORY, the *input locations of d*. Deciders have no output arcs.

2.1.3 Data Flow Graphs - Example

It might be instructive to take a familiar model for computation, a program, and consider how a corresponding data flow graph could be defined. The program in Figure 2.3 mimics integer division. That is, given a numerator N and a denominator D, it will determine a quotient Q and remainder R such that $N=Q\times D+R$, the magnitude of R is less than the magnitude of D, and if R is nonzero, it has the same sign as Q. The example is not profound, but the reader should look it over since we will refer to it in several later sections.

An obvious first step is to equip our data flow graph with a memory location for each variable in the program. Of course, we may need additional locations for temporary results and the like. The choice of actors is not quite so clear. For example, in line 6 we wish to increase Q by 1. This could be accomplished by adding the constant 1 to Q via the normal addition function, or by using a special add-1-to-the-argument unary function. This obviously makes no difference in the program, but will lead to quite different data flow graphs. A similar choice occurs for the comparisons with constant 0 on lines 9 and 13.

A complete data flow graph is shown in Figure 2.4. We could have used function names like < or - but these names are hard to dissociate from their conventional interpretations. Thinking of c as complement, ℓ as less than, m as magnitude, s as subtract, t as tally (add 1), and z as zero, the data flow graph should be quite easy to analyze. The reader is encouraged to verify line by line that the data flow of the program can be duplicated. The identity copies 0 for use by p and q.

```
input ( N , D )
Q ← 0
R ← |N|
T ← |D|
do until R < T
    Q ← Q+1
    R ← R-T
    end
if N < 0
then do R ← -R
        Q ← -Q
        end
if D < 0
then Q ← -Q
output ( Q , R )
```

Figure 2.3 A sample program for integer division

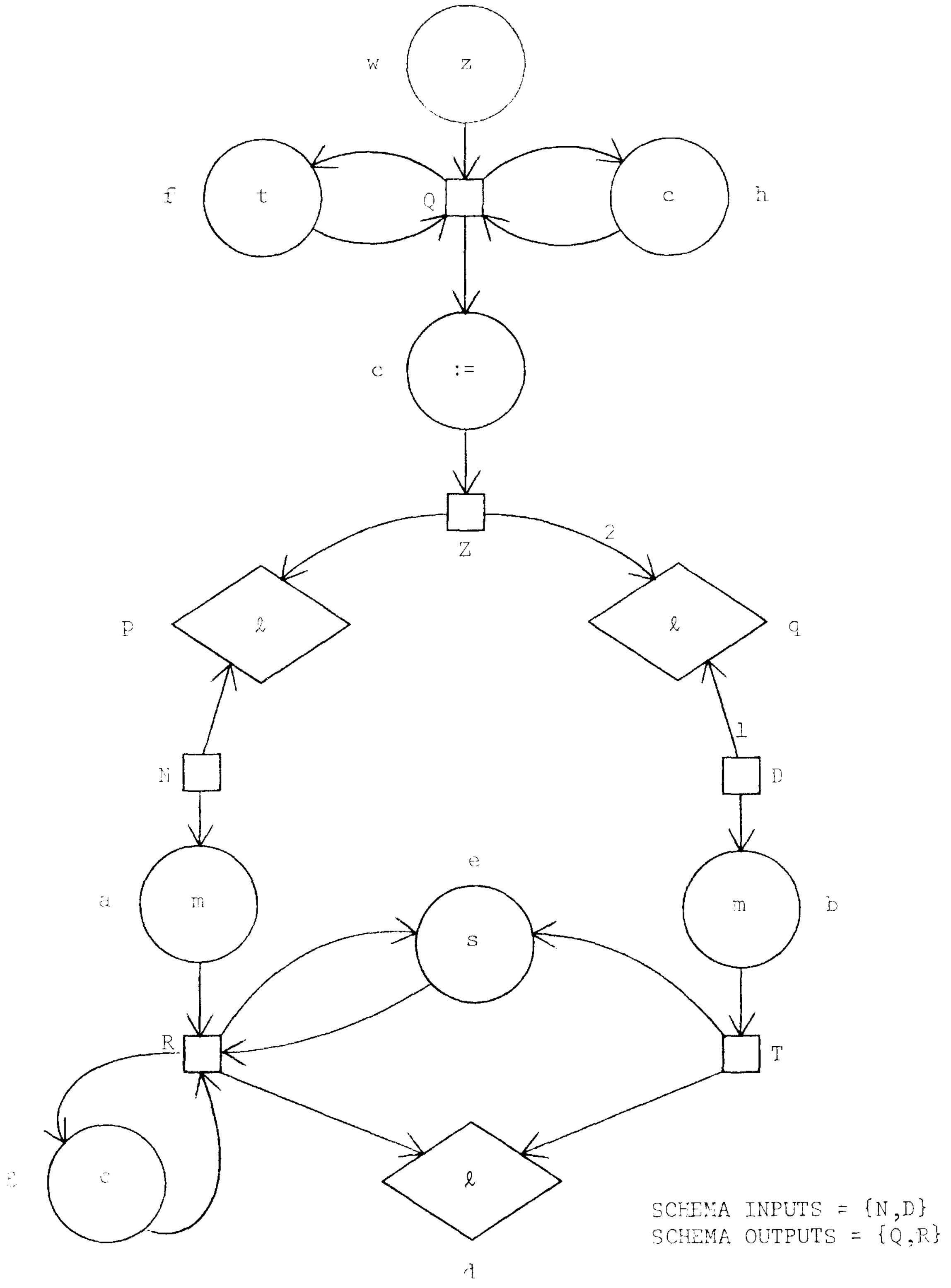


Figure 2.4 A data flow graph for the program of Figure 2.3

2.2.1 Events - Introduction

Although actors are the atomic elements by which computation can be carried out, we want a description of their behavior which is more detailed than "actor a happens." This level would suffice for sequential computations, in which there is only one site of activity. In a parallel computation, however, we need a mechanism capable of describing concurrent, asynchronous activity.

To this end, we associate with each operator in a data flow graph, an initiation event and a termination event. With each decider, we associate an initiation event and two termination events. In essence, an initiation event and corresponding termination event bracket what we referred to earlier as "actor a happening." However, it is not necessary for initiations and terminations of an actor to occur in strict alternation. By postulating a means of keeping track of initiations for which the corresponding termination has not yet taken place, we can allow several initiations to occur before a termination occurs.

When we formally describe events in the next section, we do so by assuming that each actor has an associated queue and processor which computes the function or predicate named. It should be emphasized that these processors and queues are merely vehicles for defining the intended behavior of actors. Any implementation which gives rise to the same behavior would be equally acceptable.

We now turn to a formalization of the semantics of events.

2.2.2 Events - Formal Description

For each operator o with associated n-ary function name f, we define two *events*, the *initiation* of o, denoted $\bar{o}$, and the *termination* of o, denoted $\underline{o}$. When o initiates, a value is associated with each input arc of o by (non-destructively) reading the corresponding input location of o. This n-tuple of values can be thought of as being enqueued on an f-processor associated with o. The termination of o is defined only if this fifo queue is non-empty. (It should be noted that a queue containing 0-tuples is most certainly not empty.) In this case, the first n-tuple in the queue is removed, the f-processor is applied to the n-tuple, and the result is written into the output location of o, destroying any previous contents.

For each decider d with associated n-ary predicate name q, we define three events, the *initiation* of d, denoted $\bar{d}$, and the *true* and *false terminations* of d, denoted d_T and d_F, respectively. The intended semantics are similar to those for operators. Upon termination, there is no output location to overwrite, of course, but the outcome of the predicate named by p on the n-tuple of values is reflected in the choice of termination events.

These rules define, for each data flow graph, an alphabet Σ of events associated with the actors. Since there are only finitely many actors, Σ is always finite.

2.2.3 Events - Example

Suppose, referring back to the data flow graph of Figure 2.4, we consider the sequences of events $\bar{w}\underline{w}\bar{f}\underline{f}\bar{f}\underline{f}$ and $\bar{w}\underline{w}\bar{f}\bar{f}\underline{f}\underline{f}$ and their effect upon the contents of location Q. When we begin, the contents of Q are undefined. When w initiates, the contents of Q remain undefined, but now there is a 0-tuple enqueued on w's z-processor. When w terminates, a value which we can symbolically refer to as z() is written into Q. At this point, all queues are again empty. When f initiates, the 1-tuple z() is enqueued on f's t-processor. Up until now, both of sequences we have been considering have caused the same behavior.

If f now terminates immediately, t(z()) will be written into Q, destroying its former contents, z(). After another initiation and termination of f, Q will contain t(t(z())) and all queues will be empty.

On the other hand, if f reinitiates before terminating, z() is placed on the t-processor's queue after the other 1-tuple (which also happens to be z().) The first termination of f leads to t(z()) being written into Q. The second termination then overwrites this value with the very same thing, t(z()). Again, all queues have been emptied.

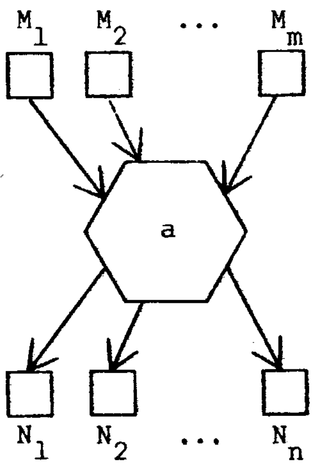
For the data flow graph shown, $\Sigma=\{\bar{w},\underline{w},\bar{a},\underline{a},\bar{b},\underline{b},\bar{c},\underline{c},\bar{e},\underline{e},\bar{f},\underline{f},\bar{g},\underline{g},\bar{h},\underline{h},$ $\bar{d},d_T,d_F,\bar{p},p_T,p_F,\bar{q},q_T,q_F\}$.

2.3.1 Data Flow Graphs - Summary

Data flow graphs provide a formalism for describing the structural aspects of computation. They are not the only such mechanism one can devise. Karp and Miller[7], Slutz[16], and Keller[8] have used a model in which actors are of a single, hybrid type. These "operations" combine the features of our operators and deciders by having one or more output locations and one or more possible outcomes. A graphical representation of such an actor is shown in Figure 2.5. An equivalent data flow graph structure is shown in Figure 2.6. We mimic the multiple outputs in the standard way. There are numerous methods for associating K(a) outcomes with collections of binary deciders. As shown, we use K(a)-1 deciders, and let the outcome correspond to the least index of a true decider, or K(a) if all are false. Thus, data flow graphs can model the hybrid operations.

A more dramatic difference appears in one variant of the data flow schemata of Dennis[3] and Fosseen[5] Here actors are interconnected by fifo queues instead of memory locations. Data flow graphs can also reproduce the behavior exhibited by data flow schemata. It can be shown that the so-called well-formed schemata can operate with queues of length 1. Such queues are easily modeled by memory locations. Furthermore, queues can be modeled by identity operators between memory locations, using the implied queuing of multiple operator initiations.

Therefore, data flow graphs are a fairly general model for describing the structure of a computation and are not likely to generate much controversy. Indeed, this is one reason why the data flow structure has been separated from the rest of the schema specification.



a has outcomes $\{a_1, a_2, \ldots, a_{K(a)}\}$

Figure 2.5 A Karp-Miller operation

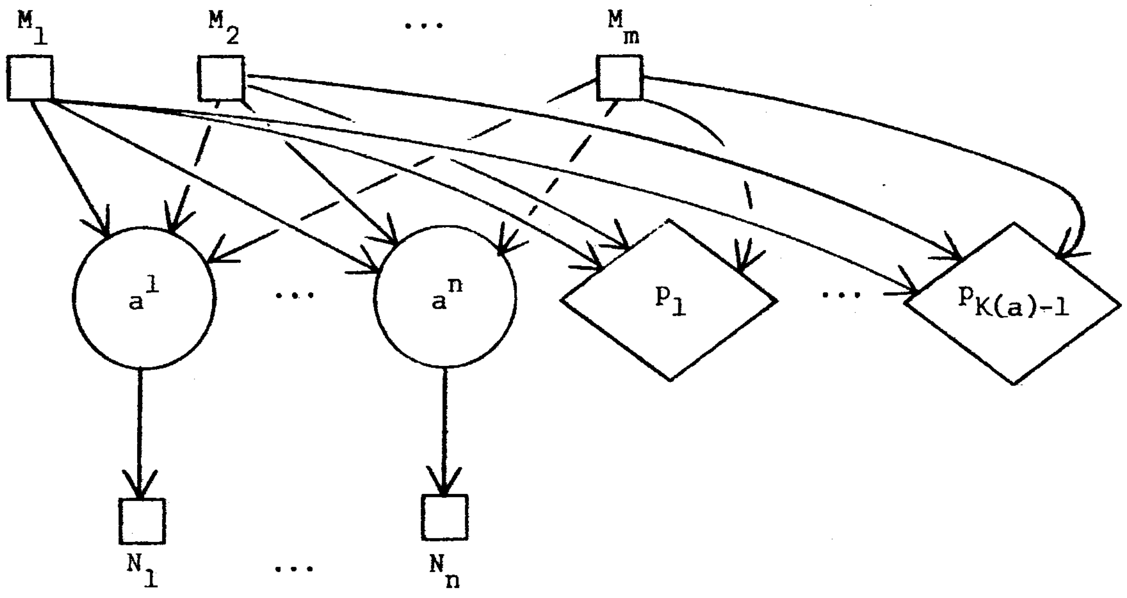


Figure 2.6 A data flow graph model for the operation of Figure 2.5

3.1.1 Control - Introduction

Data flow graphs tell us only part of what we need to model a computation. They specify the structure of memory, transformations, and tests, but do not specify the order in which actors can initiate and terminate. This is the purpose of some sort of control mechanism. What we shall be referring to as a *schema* is a data flow graph and a specification of control. We review some possible control mechanisms before indicating how control is specified for our schemata.

3.2.1 Control - Sequential Models

The most familiar control mechanisms, the ones used by virtually all conventional programming languages, are sequential. There is no concurrent activity, so actor initiations and terminations occur with no intervening events. After each operator termination, there is a unique actor initiation which follows. When a decider terminates, there are two actor initiations which might follow, the choice being determined by the decider outcome. The rules about which actor initiates next are implicit in the semantics of the programming language used.

Flow charts are a general way of specifying the sequencing of actors in a sequential program. Figure 3.1 shows a flow chart for the program in Figure 2.3. One can envisage a single locus of control moving along the arcs of the chart, initiating and terminating actors as it passes. If a program contains go to statements, the topology of such a chart can be quite complicated, a fact used to argue for better behaved sequential control primitives such as do-loops and if-then-else conditional clauses.

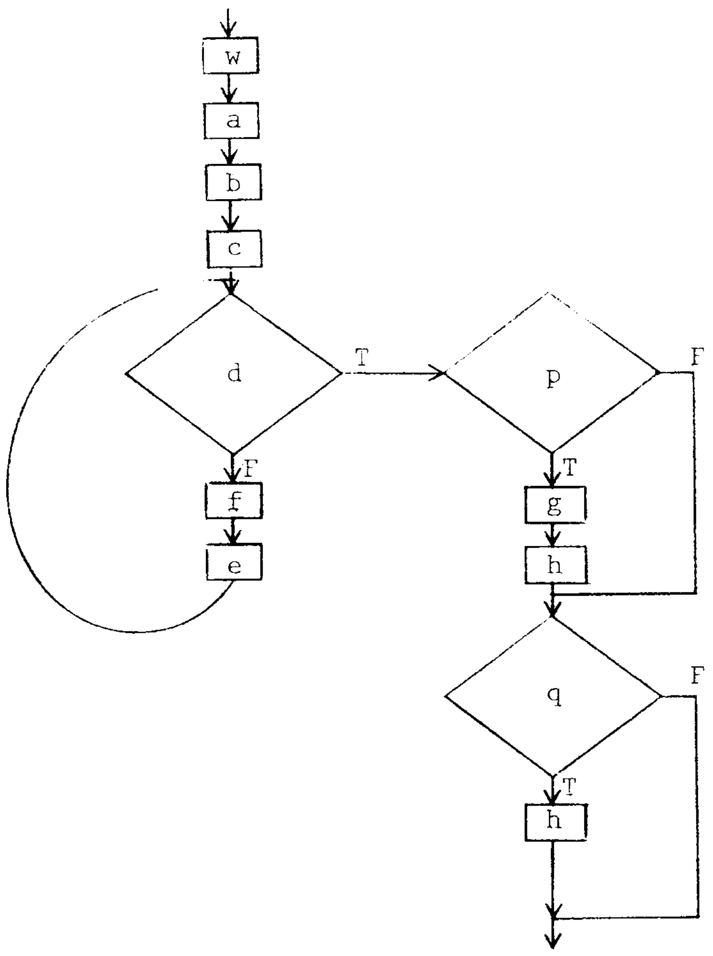


Figure 3.1 A flow chart control specification for the data flow graph of Figure 2.4

3.3.1 Control - Programs with Concurrency

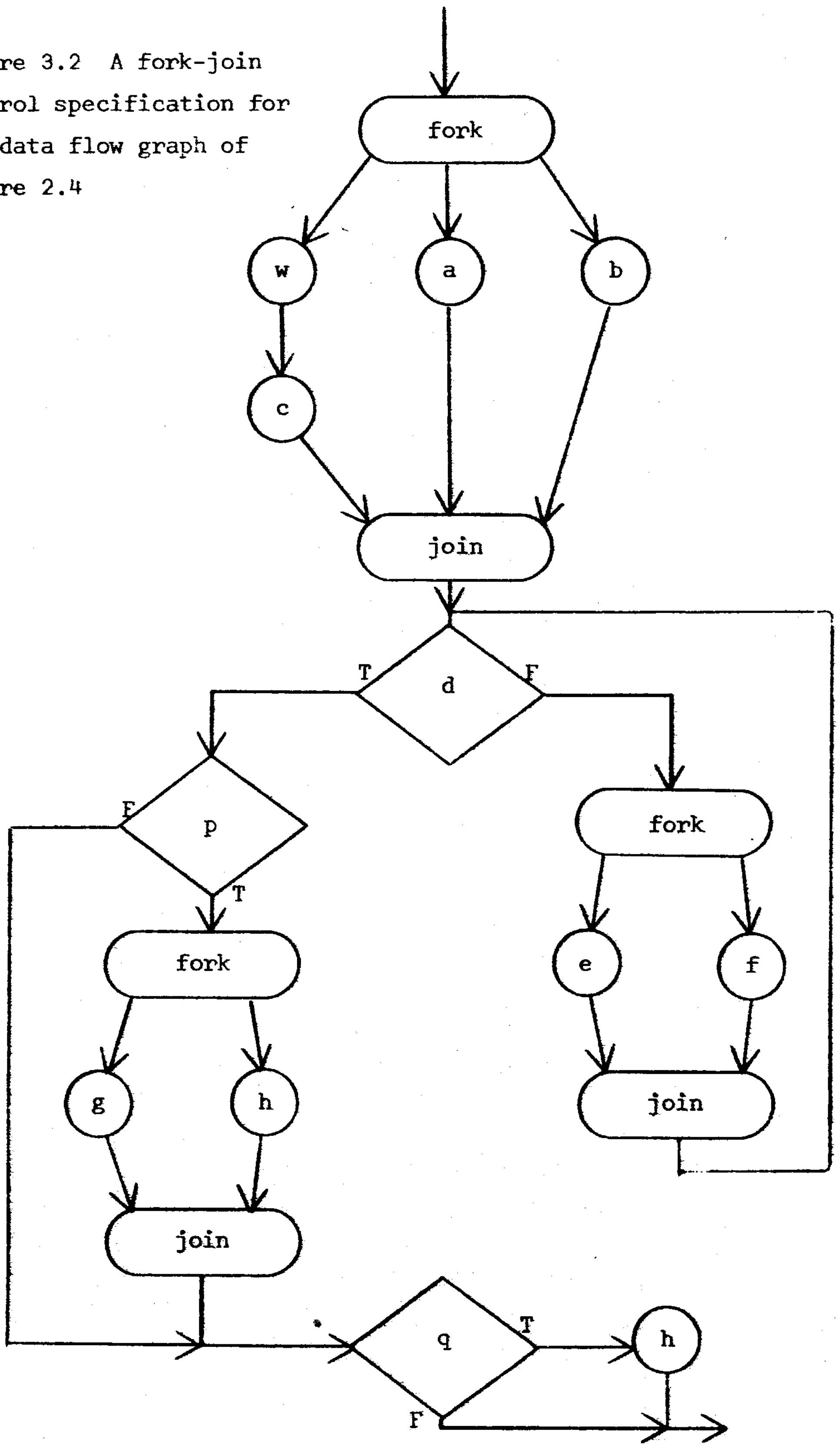
If we wish to maintain a program-like formalism but introduce concurrent activity, there must be a means of adding loci of control. If a program can be broken into completely independent parts, then each part can be assigned a dedicated locus of control. This is essentially what most multiprocessing systems do, assuring independence by working on unrelated programs. Practical programs, however, seldom factor into such independent parts. (If they did, they would be written as separate programs.) In general, then, the loci of control will interact.

Fork and join primitives can be added to programming languages to express this interaction. At a fork, a single locus of control splits into several loci. At a join, a number of loci of control come together and a single locus exits. There have been numerous proposals[1,4] for implementation of such primitives, and we show a graphical representation of a program using these primitives in Figure 3.2.

In inspecting such a graph, it is important to distinguish between the control flow through a fork primitive, in which control flows along <u>all</u> output arcs, and control out of deciders, in which control flows along exactly one of the output arcs.

The resemblence to a precedence graph is quite striking here, with "unordered" actors being capable of concurrent activation. Precedence graphs, embellished to allow conditionals and while-loops have also been used as control mechanisms[2].

Figure 3.2 A fork-join control specification for the data flow graph of Figure 2.4



3.4.1 Control - Petri Nets

Breaking away from program-like formalisms can lead to control structures such as Petri nets, introduced by Holt[6] Petri nets are such a simple yet powerful formalism for modeling concurrent activity that it is worth a short digression to study them.

Petri nets are directed, labeled, bipartite graphs whose nodes are either *states* or *events*. States are drawn as circles and events as bars. States have an associated non-negative integer which is depicted by drawing the appropriate number of *tokens* in the state. This numbering is called the *marking* of the net. Activity in a Petri net is governed by a simple *firing rule*. An event is said to be *enabled* if each state on an arc into the event contains at least one token. An enabled event can *fire* by removing one token from each input state and adding one token to each output state. Tokens need not be "conserved" by the process of firing. In fact, the total number of tokens in the net will change after an event fires unless the number of input states of the event equals the number of output states of the event. We show some Petri nets in Figure 3.3.

The event in Figure 3.3a is not enabled since there is an input state with no tokens. The event in Figure 3.3b is enabled, and, if it should fire, the resulting net would be that of Figure 3.3c.

By associating the events determined by a data flow graph with a subset of the events in a Petri net, we can use the net as a control mechanism. We show such a net in Figure 3.4.

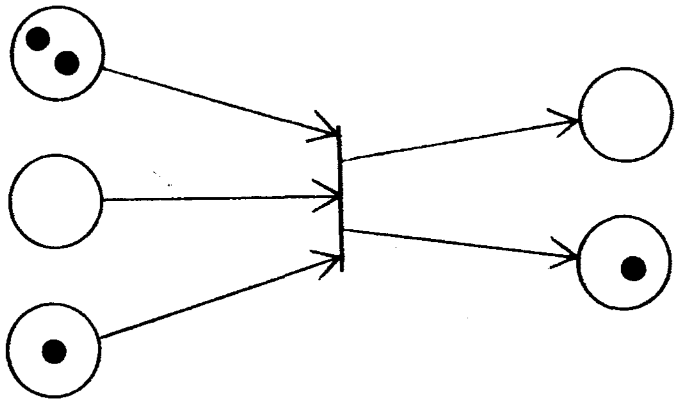


Figure 3.3a An event which is not enabled

Figure 3.3b An event which is enabled

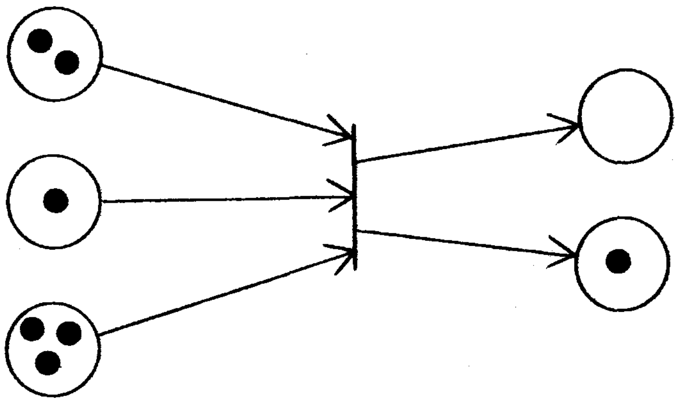
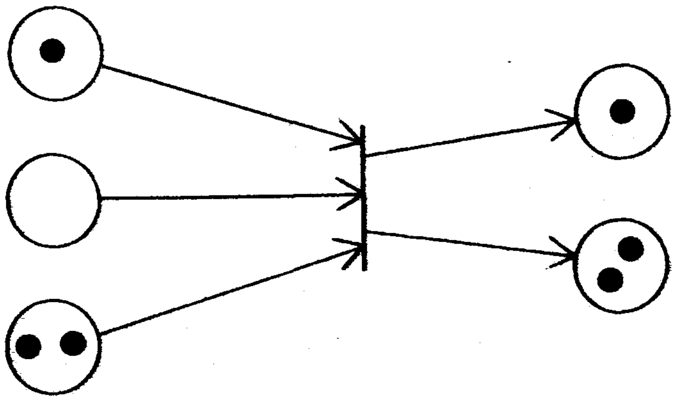


Figure 3.3c The marking which would result if the event of Figure 3.3b fired



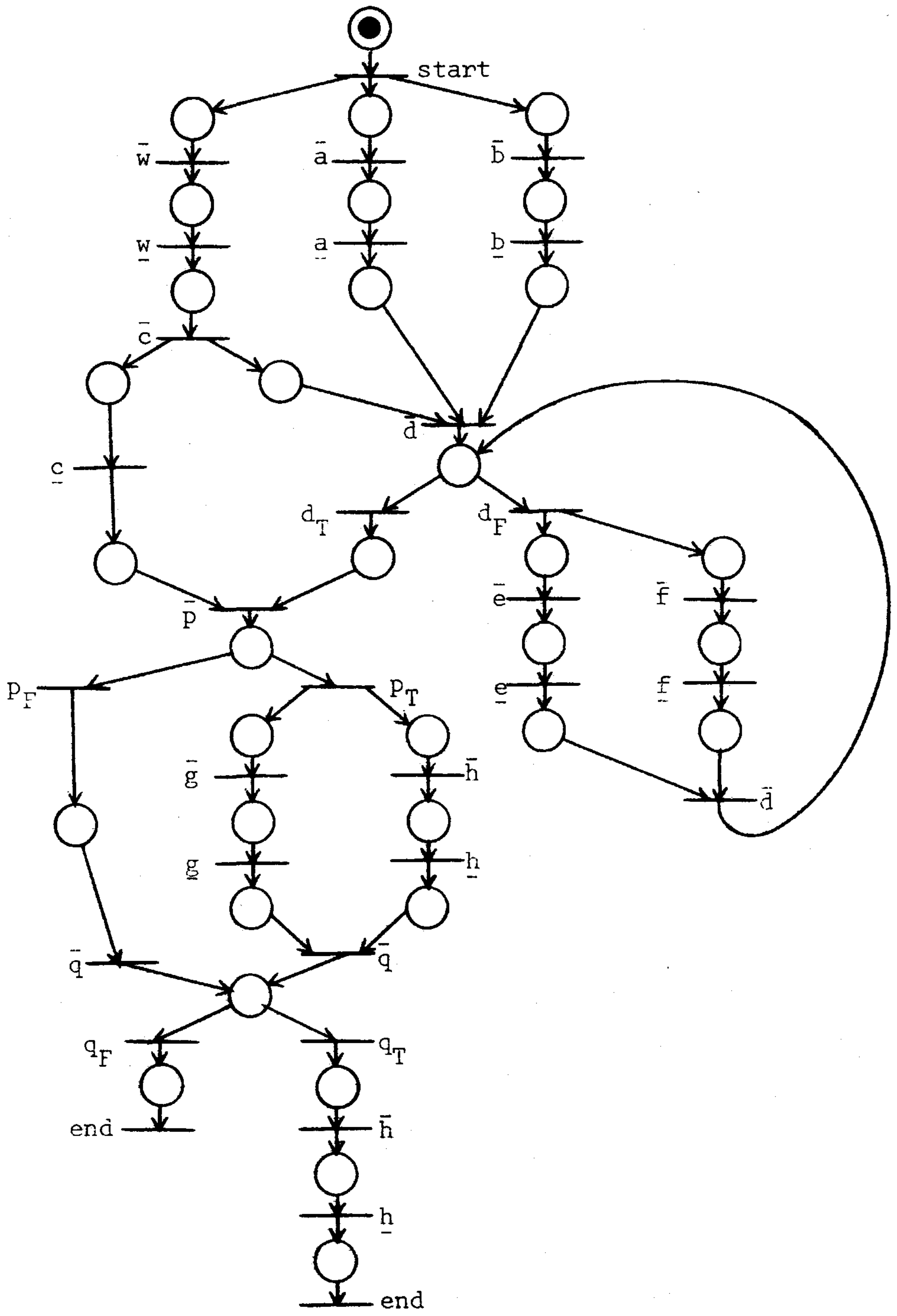


Figure 3.4 A Petri net control specification for the data flow graph of Figure 2.4

One reason that this control mechanism may appear complicated is the introduction of explicit initiation and termination events. Except for this, there is a strong resemblance to the fork/join formulation of Figure 3.2.

Slutz's *flow graph schemata* have a control mechanism which closely parallels Petri nets.[16]

3.5.1 Control - Data Flow Models

It is self-evident that an actor cannot initiate before its input values have been generated. Turning this observation around, Rodriguez,[13] Dennis,[3] and Fosseen[5] have proposed models in which the arrival of values controls the initiation of actors. We will briefly describe one form of data flow schemata studied by Dennis and Fosseen.[3] We call these "Dennis-Fosseen schemata" rather than "data flow schemata" to avoid possible confusion with our own data flow graphs.

A Dennis-Fosseen schema is a directed, bipartite graph in which the nodes are either *actors* or *links*. It is helpful to think of the arcs in a Dennis-Fosseen schema as being able to hold a single value. Operators in Dennis-Fosseen schemata are actors with one or more input arcs, a single output arc, and an associated function name. If a value is present on each input arc and no value is present on the output arc, an operator is enabled to fire. When it fires, an operator removes the values from its input arcs and places on its output arc the value which results from applying the named function to the inputs. Deciders in a Dennis-Fosseen schema have one or more input arcs, a single output arc, and an associated predicate name. As with operators, these actors are enabled to fire when there is a value on each input arc and no value on the output arc. When a decider fires, the values on its input arcs are removed, and the boolean value resulting from the named predicate on the input values is placed on the output arc. Dennis-Fosseen schemata are constructed in such a way that arcs carry either boolean values or data values, never both.

Dennis-Fosseen schemata may contain boolean actors to perform logical operations on boolean values. Boolean actors are enabled to fire when a boolean value is present on each input arc and no value is present on the output arc. When such an actor fires, the input values are removed and the result of the named boolean operation on the input values is placed on the output arc.

There are three types of actors which have both boolean and data input; *true gates, false gates,* and *merges*. A true gate has one input data arc, one input boolean arc, and one output data arc. A true gate is enabled to fire when a value is present on each input arc and no value is present on the output arc. When it fires, the input values are removed. If the boolean value was *true*, the input data value is placed on the output arc. Otherwise, no value is placed on the output arc. False gates are analogous, "passing" the input data value if a *false* boolean value is present and "swallowing" the data value for *true* boolean inputs.

Merge actors have two input data arcs, one boolean input arc, and one output data arc. A merge is enabled to fire when there is a boolean value present, a data value on the input data arc indexed by the boolean value, and no value on the output arc. (The presence or absence of a data value on the other input data arc is irrelevant.) When it fires, a merge removes the boolean value and the indexed data input value, placing the data value on the output arc.

Link nodes serve as connection points and also eliminate the need for explicit identity operators. A link node is enabled when a value is present on its input arc and all output arcs contain no values. When a link node fires, the input value is removed and a copy of the value (data or boolean) is placed on each output arc of the link node.

Figure 3.5 summarizes the elements of Dennis-Fosseen schemata. Each element is shown in enabled status and then, immediately to the right, the element is shown as if it had fired. Arcs which carry boolean values have solid arrowheads while data arcs have open arrowheads.

Figure 3.6 shows how the elements of a Dennis-Fosseen schema can be assembled to model the program of Figure 2.3. This kind of schema is probably foreign to most readers, but with a little experimentation, one can quickly become familiar with its behavior. In particular, gates and merges come in groups fed by the same boolean. When initialized merges are encountered before the gates, a loop structure can be realized. This is the case with the three merges and five gates fed by the decider in the middle of Figure 3.6. When the gates are encountered before the merge, as is true with the gates and merges fed by the deciders at the top of Figure 3.6, conditionals are realized. We recommend that the reader check the behavior of the schema by first assuming that each decider is *false*, then assuming each is *true*. Note, in particular, that the "loop merges" are reinitialized to *true* when the loop is exited.

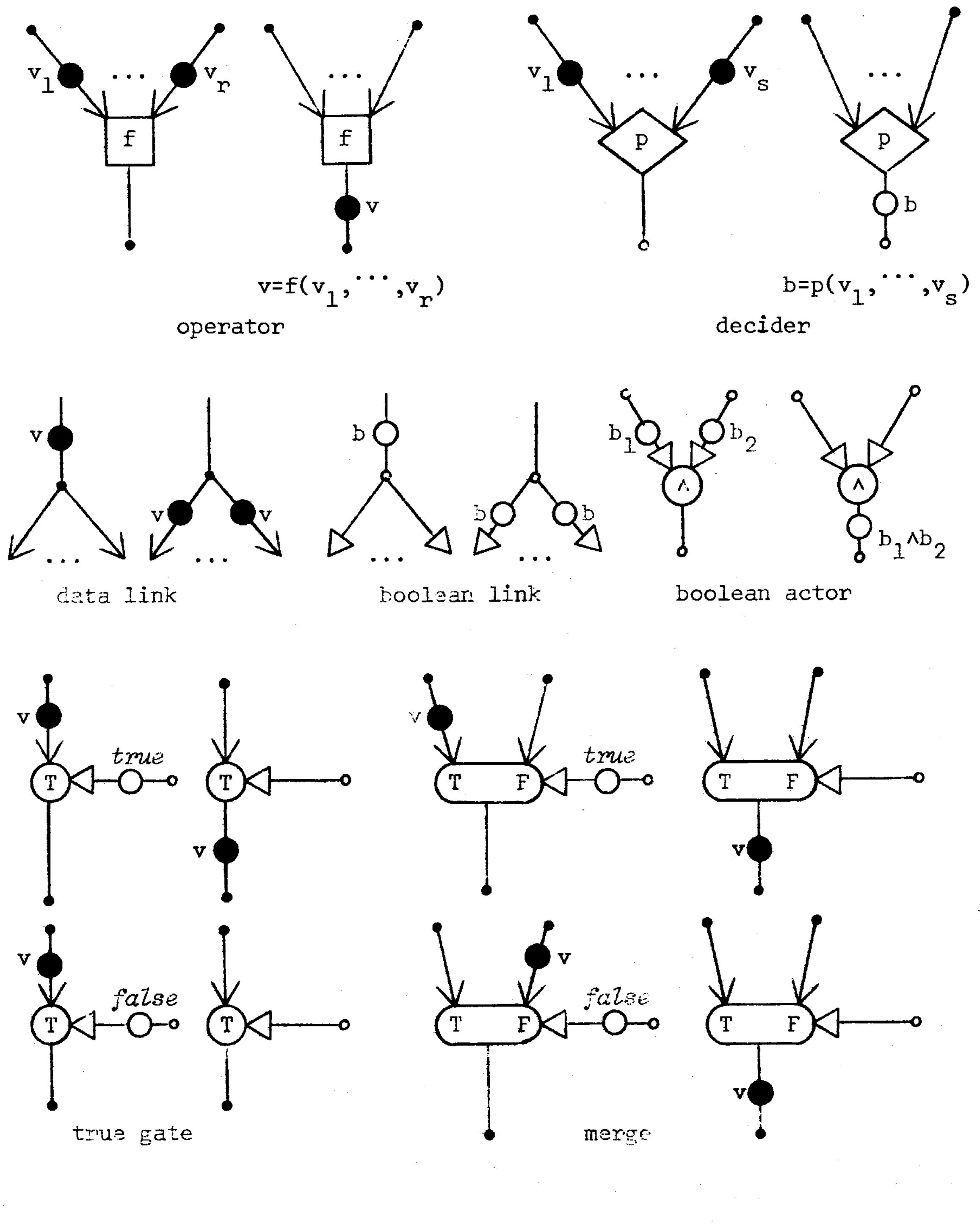
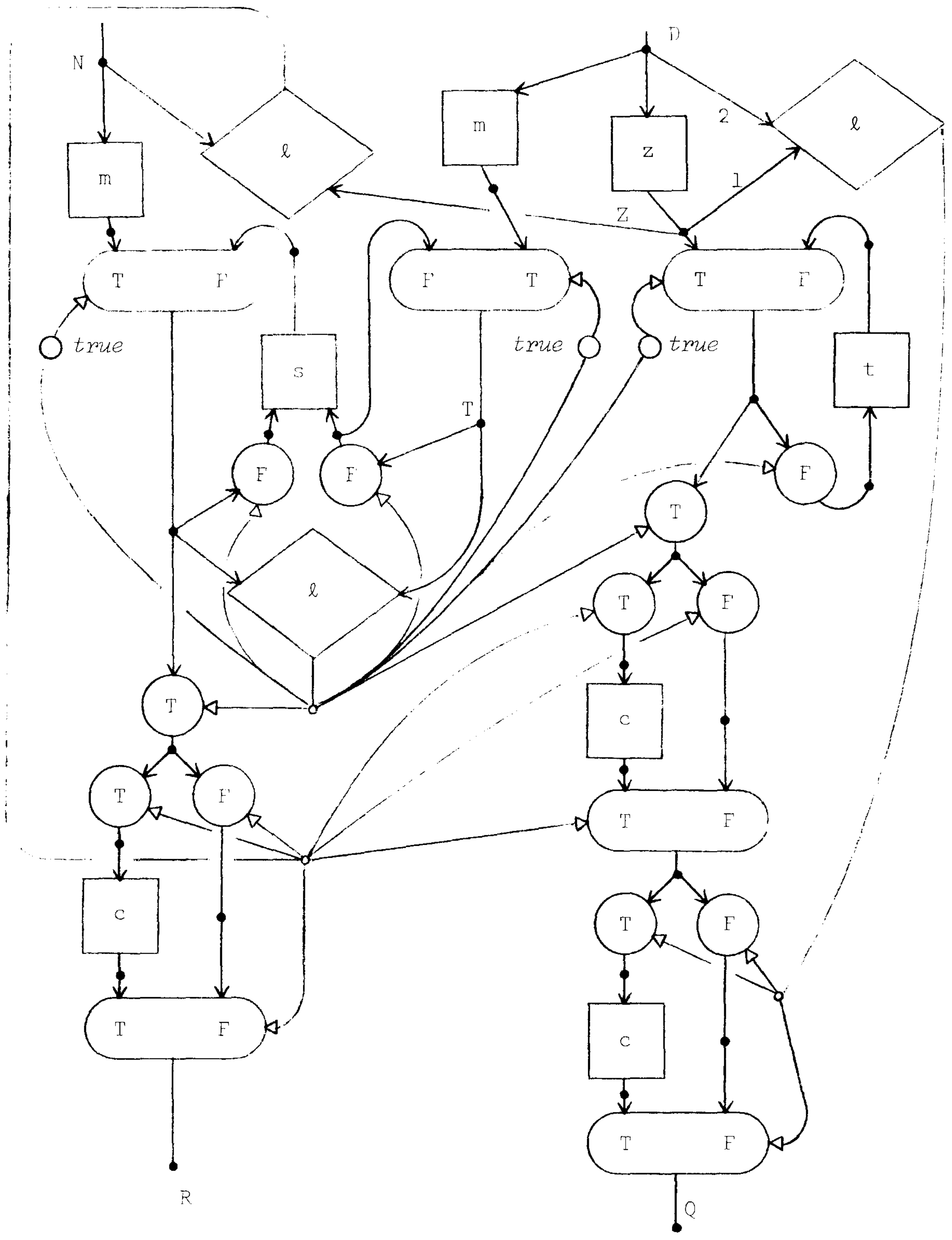


Figure 3.5 Elements of Dennis-Fosseen schemata, enabled and fired



Fi[illegible] [illegible] Dennis-Fosseen schema for the program of Figure 2.3

3.6.1 Control - Automata

Consider a finite state acceptor whose state transitions are labeled with the events associated with a data flow graph. We can view the events on arcs out of a given state as those events which the acceptor allows in that state. Should one of the events occur, the acceptor enters the state to which the corresponding arc leads. We might call those sequences of events which lead from the initial state to some accepting state the sequences which are *acceptable*.

Keller[8], Karp and Miller[7], and Slutz[16] all consider state-machine control mechanisms, treating countably infinite state machines as well as finite state acceptors. Many of the other control mechanisms we have discussed have finite state counterparts which allow exactly the same sequences of events to occur. If, for a given n-state Petri net, there is an integer m such that no state in the net ever contains more than m tokens at a time, then there is a corresponding finite state control mechanism with no more than $(m+1)^n$ states. (A state in the finite state acceptor identifies how many tokens are present in each of the n states of the Petri net.)

As the last example suggests, finite state acceptors may be considerably more complicated than the other models. In Figure 3.7 we show the beginning of a state table for a finite state control mechanism for the data flow graph of Figure 2.4.

One can imagine how pushdown automata or even Turing machines could be employed to control the events in a data flow graph.

q_0	$\bar{w}$	q_1	q_4	$\bar{c}$	q_{10}
	$\bar{a}$	q_2		$\bar{a}$	q_{11}
	$\bar{b}$	q_3		$\bar{b}$	q_{12}
q_1	$\underline{w}$	q_4	q_5	$\underline{w}$	q_{11}
	$\bar{a}$	q_5		$\underline{a}$	q_{13}
	$\bar{b}$	q_6		$\bar{b}$	q_{14}
q_2	$\bar{w}$	q_5	q_6	$\underline{w}$	q_{12}
	$\underline{a}$	q_7		$\bar{a}$	q_{14}
	$\bar{b}$	q_8		$\underline{b}$	q_{15}
q_3	$\bar{w}$	q_6	q_7	$\bar{w}$	q_{13}
	$\bar{a}$	q_8		$\bar{b}$	q_{16}
	$\underline{b}$	q_9			

Figure 3.7 Partial state table of a finite state control specification for the data flow graph of Figure 2.4

3.7.1 Control Sets - Overview

The last few sections serve to indicate the diversity of control mechanisms which have been or could be proposed to direct activity in data flow graphs. Each has advantages, real or imagined, which make it difficult to agree on a "best" mechanism.

It is unfortunate that many of the results in schematology depend upon the particular choice of control mechanism, in ways that become clear only when one attempts to carry the result over to a different model. This works a particular hardship on newcomers to schematology since there is no general theory unifying the various models.

In an effort to begin a more general approach to schematology, we have adopted a specification of control which is not tied to any particular control mechanism. We do this by observing that all the mechanisms just discussed define sets of allowed sequences of events. We call these *control sets*, and we will generally be concerned with their properties as sets, not with the particular mechanism that determines them.

This approach has both good and bad aspects. A control mechanism defines a control set inherently possessing properties which we must explicitly define and justify. Thus, our approach may appear slightly "verbose" and can be difficult to motivate. The advantage we gain in return is to make explicit those properties of control upon which schematalogical results depend. These results then carry over to <u>all</u> mechanisms which can be shown to impose the requisite properties. This may be an easier task than trying to establish the results directly.

3.7.2 Control Sets - Conventions and Formalisms

Let Σ be a finite alphabet. As is conventional, we will use Σ^* to denote the set of all finite strings over Σ. We let λ represent the unique string of length 0, the *empty string*. We use Σ^∞ to denote the set of countably infinite strings over Σ, and we define $\hat{\Sigma}$ to be the union of Σ^* and Σ^∞. We will use the terms 'string' and 'sequence' interchangeably.

We use italicized letters late in the alphabet and possibly subscripted, v, w, x_0, x_1, and so forth, to denote sequences in $\hat{\Sigma}$. If we have sequences $x\epsilon\Sigma^*$ and $y\epsilon\hat{\Sigma}$, then xy represents the sequence in $\hat{\Sigma}$ formed by concatenating x and y. Given $z\epsilon\hat{\Sigma}$ and $x\epsilon\Sigma^*$, x is a *prefix* of z if and only if there exists a sequence $y\epsilon\hat{\Sigma}$ such that $xy=z$.

Suppose Σ is the alphabet of events associated with a data flow graph. A string $y\epsilon\hat{\Sigma}$ is said to be *well-sequenced* if, for every prefix x of y and for every actor a in the data flow graph, the number of occurrences of initiations of a in x is no less than the number of occurrences of terminations of a in x. In simple English, nothing is terminated which has not been initiated. Note that this property does not depend upon the connections in a data flow graph, only on the set of actors.

A sequence $y\epsilon\hat{\Sigma}$ is *well-defined* (for the data flow graph which determines Σ) if it is well-sequenced and for each prefix $x\bar{a}$ of y, and for each input location ℓ of actor a, ℓ is a schema input, or ℓ is the output location of some operator o such that $x = x_1\underline{o}x_2$.

That is, in a well-defined sequence, no actor initiates until the contents of all of its input locations are meaningfully defined. A set of sequences is said to be *well-defined* if every sequence in the set is well-defined.

We define a *control set* (of a data flow graph with alphabet of events Σ), CONTROL, to be any well-defined subset of $\hat{\Sigma}$ with the following property. For each finite sequence $x \in$CONTROL, each schema output m is either the output location of some operator terminating in x, or m is a schema input (or both). Thus each *control sequence*, as we call the elements of CONTROL, which terminates leaves values in all schema output locations.

Finally, and somewhat anticlimactically, we define a *schema* to be a data flow graph and some corresponding control set.

3.7.3 Control Sets - Discussion

The restriction of control sets to well-defined sequences is easy enough to justify. A string which is not well-sequenced, such as $\underline{a}\bar{a}$, simply has no meaning in terms of the behavior of data flow graphs. Similarly, we cannot make any sense out of actors initiating before the contents of their input locations have been established. For example, $\bar{e}\underline{e}$ is well-sequenced for the data flow graph of Figure 2.4, but it is not meaningful because locations R and T have unspecified contents. Finally, if a sequence terminates, we expect it to leave a result in each schema output location. Although we do not exclude anything very meaningful by assuming well-definition, there are noteworthy implications. For example, the finite strings the state-machine of Figure 3.8 accepts are simply $\{\bar{a},\underline{a}\}^*$, but the set of all control sequences contained in $\{\bar{a},\underline{a}\}^*$ cannot be recognized by <u>any</u> finite state acceptor. (Informally, $\bar{a}^k\underline{a}^k$ must be accepted for all integers $k \geq 0$. For k greater than the number of states in a finite state machine, some state must be visited more than once while $\underline{a}^k$ is being read. Then this cycle of length $j>0$ could be repeated so that $\bar{a}^k\underline{a}^{k+j}$ is also accepted. But this sequence is not well-sequenced, so it is not among the control sequences.) Thus, although it is semantically painless to exclude ill-defined sequences, it may be difficult or impossible to do so while maintaining a given control mechanism.

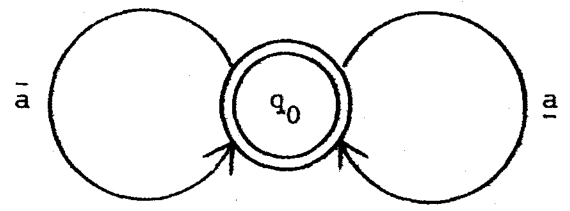


Figure 3.8 A finite state control specification for which CONTROL is not a regular set

4.1.1 Interpretations - Introduction

Schemata, via their data flow graphs and control sets, tell us everything of a structural and operational nature that we need to know about modeling computations. To put the schema model on a par with familiar models, however, we must supply the missing details about the function names and predicate names.

An *interpretation* for a schema specifies the set of elements which the memory locations can contain. For "familiar interpretations", these will include integers, character strings, representations of real numbers, and so forth. Of course, there are also interpretations dealing with trees, or only with integers, or only with the number 6: the choice of domain is virtually unrestricted.

An interpretation also specifies functions and predicates over these elements for the function names and predicate names of the data flow graph. These functions and predicates are total, so "familiar interpretations" would have to be extended to define such things as 1/0 and "cat"<3.14159. Finally, the interpretation specifies the elements initially contained in the schema input locations.

4.1.2 Interpretations - Formal Definition

An *interpretation* I of a schema consists of

1) A (non-empty) set, DOMAIN, of *elements*.
2) For each m-ary function name f, a total function $f^I\text{:DOMAIN}^m \to \text{DOMAIN}$
3) For each m-ary predicate name p, a total predicate $p^I\text{:DOMAIN}^m \to \{T,F\}$
4) For each schema input location ℓ, an *initial element* $d \in \text{DOMAIN}$

4.1.3 Interpretations - Summary

Although we have defined interpretations of schemata, it is evident that the definition depends only upon the data flow graph component. If we have two or more data flow graphs with the same number of schema inputs, we can meaningfully define a single interpretation for all of them. There will be a single DOMAIN, and the set of initial elements will apply to each set of schema inputs. Functions and predicates must be assigned for <u>all</u> function names and predicate names, but since the arity is implicit in the names, this can be done consistently. That is, f cannot be a unary name in one graph and a binary name in another. Thus, a function associated with name f works in all data flow graphs. We will later speak of interpretations for two or more schemata, understanding that this is what is meant.

4.1.4 Interpretations - Example

Suppose we define an interpretation for the data flow graph of Figure 2.4. We can recapture the original intent of the program from which it was derived by the following interpretation.

1) DOMAIN = Z (The integers)

2) $z():=0$ $m(x):=|x|$ $t(x):=x+1$ $c(x):=-x$ $s(x,y):=x-y$

3) $\ell(x,y):=x<y$

4) Initial elements = <1,-1>

Example 4.1 An interpretation for the data flow graph of Figure 2.4

5.1.1 Data Dependence Graphs - Introduction

We have defined schemata as the combination of data flow graphs, a formalism for describing the interconnection of memory locations and actors, and control sets, a formalism for defining how events in a data flow graph are sequenced. We now define a mechanism, the *data dependence graph*, or *dadep graph*, which brings these two formalisms together.

Before proceeding, it should be noted that dadep graphs are derived algorithmically from a data flow graph and associated well-defined sequence. Thus, there is really nothing in a dadep graph we didn't already have, given the data flow graph and sequence. Our justification for introducing this new formalism is one of clarification and convenience.

The virtue of dadep graphs is that they make it easy to overlook some less interesting aspects of schemata such as the choice of names for memory locations and actors, and focus on relevant issues such as transforming and testing values via functions and predicates. As we shall see, it is possible to do this in such a way that we can derive important information about interpretations from the structure of the dadep graph.

Dadep graphs are labeled, bipartite graphs whose nodes we call values and actions. Starting only with the initial values input to the schema, actions are added as actors initiate. One set of labels indicates the function name or predicate name associated with an action. Another set of labels indicates which values are currently contained in the memory locations. This labeling aids in determining which values are inputs to an action, shown by directed, indexed arcs from values to actions.

New values are added when operators terminate. When deciders terminate, another set of labels indicate the outcome for the associated decision. Identity operators do not create values, but rather change the labeling with memory location names to make existing values available under new location names. After formalizing these notions, an example of a dadep graph will be developed in some detail.

5.1.2 Dadep Graphs - Formal Definition

The *dadep graph* determined by a data flow graph and corresponding well-defined sequence is a directed, labeled, bipartite graph. One set of nodes is referred to as VALUES. The other set is referred to as ACTIONS. ACTIONS are partitioned into OPERATIONS and DECISIONS. If the data flow graph has m schema inputs, then there exist m values labeled 1 through m called INITIAL VALUES. Each operation is labeled with a function name and each decision with a predicate name. If an action has an n-ary name, there are arcs labeled 1 through n incident upon the action from n (not necessarily distinct) values. There may be a single arc from an operation to a value. Decisions are either unlabeled or are labeled with a T or an F. A subset of VALUES are labeled with one or more locations in MEMORY.

Suppose we fix a data flow graph. We define inductively the dadep graph determined by a corresponding well-defined sequence. For notational convenience, we will use subscripting to identify the sequence determining a dadep graph. For example, VALUES_x will denote the set of values in the dadep graph determined by sequence x.

For precision in the induction, we state the following induction hypothesis:

Hypothesis 1: Each memory location which is either a schema input or the output location of some operator terminating in x is the label on exactly one value in VALUES_x.

Looking ahead to some further definitions, we also add induction hypotheses:

Hypothesis 2: The dadep graph of sequence x is acyclic.

Hypothesis 3: Each value in VALUES$_x$ is either one of the INITIAL VALUES$_x$ or it has a single input arc originating from an operation in OPERATIONS$_x$ (but not both.)

For the basis of our induction, we consider the dadep graph of the empty sequence, λ. If the data flow graph has m schema inputs, then VALUES$_\lambda$ = $\{v_1, v_2, \ldots, v_m\}$. If the i^{th} schema input is location ℓ, then value v_i is labeled ℓ. Furthermore, v_i is distinguished as the i^{th} initial value. There are no actions and no arcs. It is easy to see that hypotheses 1, 2 and 3 are trivially satisfied.

Suppose that all the components of the well-defined sequence x have been defined, and $x\sigma$, $\sigma \epsilon \Sigma$, is also well-defined. We define the dadep graph of $x\sigma$ by cases as follows:

INITIAL VALUES$_{x\sigma}$ = INITIAL VALUES$_x$ in any case. If σ is not the termination of some decider d, let j be the number of occurrences of σ in $x\sigma$. Otherwise, let j be the number of terminations of d in $x\sigma$.

Case 1: σ is the initiation of an actor c other than an identity operator. Suppose g is the function name or predicate name associated with c. VALUES$_{x\sigma}$ = VALUES$_x$, and the labeling with memory locations is unaltered. We form ACTIONS$_{x\sigma}$ by adding a new action, a, to ACTIONS$_x$. If c is an operator, a is added to OPERATIONS$_{x\sigma}$. Otherwise, it is added to DECISIONS$_{x\sigma}$. Action a is labeled g. Since $x\sigma$ is well-defined, each input location of c is either a schema input, or is the output location of an operator terminating in x. By hypothesis 1, each of these locations is the label of exactly one value in VALUES$_x$. If location ℓ is the i^{th} input location of c, add an arc labeled i from the value in VALUES$_{x\sigma}$ (=VALUES$_x$) with label ℓ to the new action a.

(If c is a 0-ary operator, no arcs are added.) All other arcs remain as in the dadep graph of x. We refer to a as "the action corresponding to the j th occurrence of c.

Hypothesis 1 held for the dadep graph of x, no terminations were added, and the labeling with memory locations was unchanged. Hence hypothesis 1 still holds for the dadep graph of $x\sigma$. Hypothesis 2 still holds since all arcs added were to a node not in the dadep graph of x, so no cycles could be introduced. Hypothesis 3 follows from the fact that no new values were introduced.

Case 2: σ is the initiation of an actor c which is an identity operator. Suppose ℓ is the input location of operator c. As in case 1, there is a value in VALUES_x labeled with ℓ. We will refer to this value as "the value corresponding to the j th occurrence of c." The dadep graph of $x\sigma$ is exactly the same as the dadep graph of x, so all hypotheses hold.

Case 3: σ is the termination of operator c which is not an identity operator. $\text{ACTIONS}_{x\sigma} = \text{ACTIONS}_x$. We form $\text{VALUES}_{x\sigma}$ by adding a new value v to VALUES_x. Since $x\sigma$ is well-defined, c has initiated at least j times in x, so there will be an operation a in ACTIONS_x corresponding to the j th occurrence of c. Add an arc from a to v, leaving all other arcs unchanged. Suppose the output location of c is ℓ. If a value in VALUES_x has label ℓ, remove the label in the labeling of $\text{VALUES}_{x\sigma}$, and, in any case, label v with ℓ.

The (re)labeling procedure guarantees the validity of hypothesis 1. Hypothesis 2 follows because the only new arc is to a node not in the dadep graph of x. It will be seen that no other cases add values or arcs into existing values. Thus hypothesis 3 is satisfied.

Case 4: σ is the termination of a decider c. Since $x\sigma$ is well-defined, c has initiated at least j times in x. Hence, there is a decision a in ACTIONS_x corresponding to the j$^{\text{th}}$ occurrence of c. In the dadep graph of $x\sigma$, a is given label T if $\sigma=c_T$, or label F if $\sigma=c_F$. No other changes are made, so all induction hypotheses remain valid.

Case 5: σ is the termination of identity operator c. Since $x\sigma$ is well-defined, there are at least j initiations of c in x. Hence there is a value v in VALUES_x corresponding to the j$^{\text{th}}$ occurrence of c. If the output location of c is ℓ and ℓ is the label on some value in VALUES_x, remove the label in the labeling of $\text{VALUES}_{x\sigma}$. In any case, label v with ℓ in $\text{VALUES}_{x\sigma}$. No other changes are made.

The (re)labeling preserves hypothesis 1, and since no arcs or nodes are added, hypotheses 2 and 3 also hold.

5.1.3 Dadep Graphs - Example

The formal definition of dadep graphs will be useful in constructing some believable proofs, but it makes dadep graphs appear more formidable than the really are. Stepping through a specific construction should help to eliminate many potential sources of confusion. In Figure 5.1 we do this for the data flow graph of Figure 2.4 and the well-defined sequence $x = \bar{w}\underline{w}\bar{c}\bar{a}\bar{b}\underline{a}\underline{b}\bar{d}d_F\underline{c}\bar{f}\underline{f}\bar{e}\underline{e}\bar{d}d_T\bar{p}p_F\bar{q}q_T\bar{h}\underline{h}$.

Values are drawn as boxes, actions as the same shape as the corresponding actor in a data flow graph. We identify the initial values by the index depicted inside. The memory labels appear alongside values, function and predicate names appear inside the corresponding actions, and decision outcomes appear near the predicate names. We use the same arc labeling conventions introduced for data flow graphs.

As an inspection of the dadep graph of x reveals, dadep graphs display the structural relationships between the values input and those generated by a sequence. This structure can then be used to study what happens under a particular interpretation.

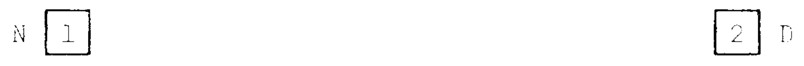


Figure 5.1a Dadep graph of λ

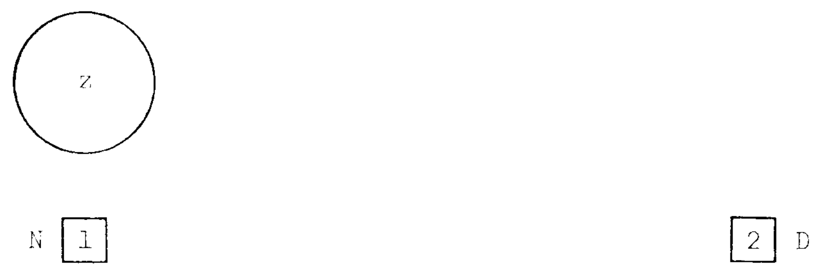


Figure 5.1b Dadep graph of $\bar{w}$

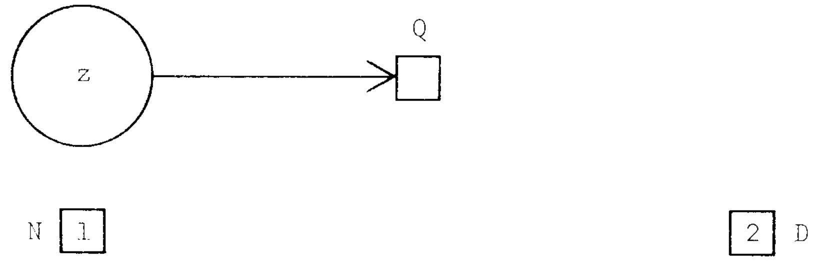


Figure 5.1c Dadep graph of $\bar{w}\underline{w}$

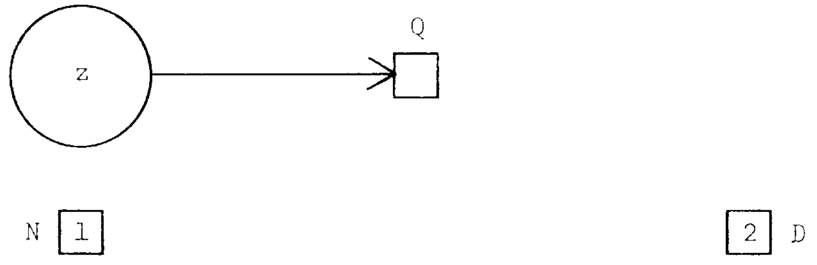


Figure 5.1d Dadep graph of $\bar{w}\underline{w}\bar{c}$

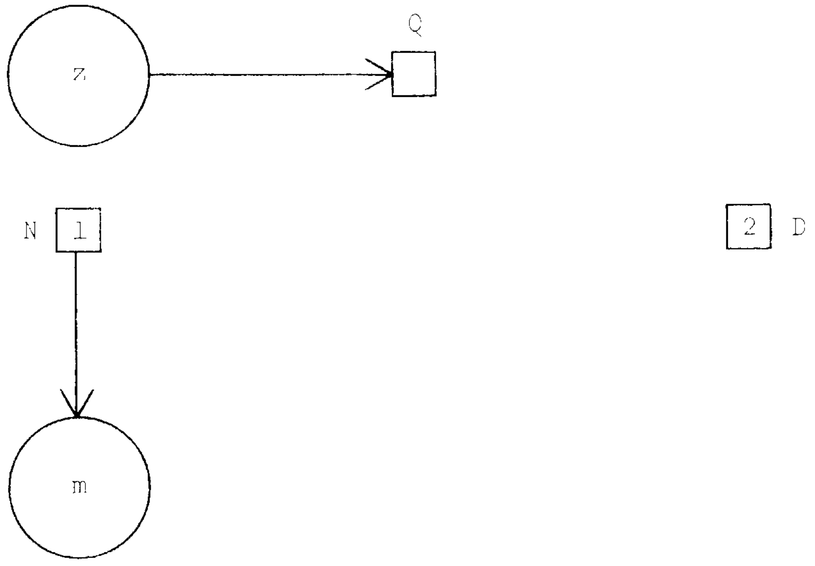


Figure 5.1e Dadep graph of $\bar{w}\underline{w}\bar{c}\bar{a}$

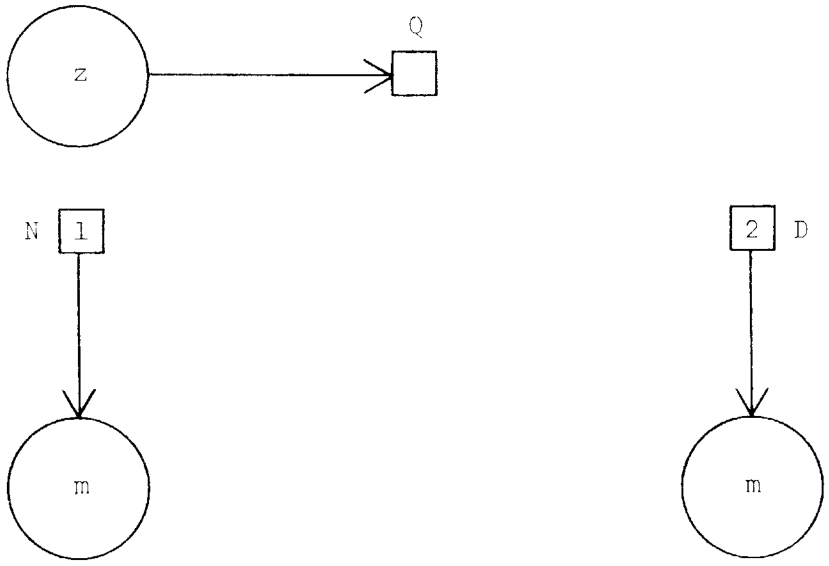


Figure 5.1f Dadep graph of $\bar{w}\underline{w}\bar{c}\bar{a}\bar{b}$

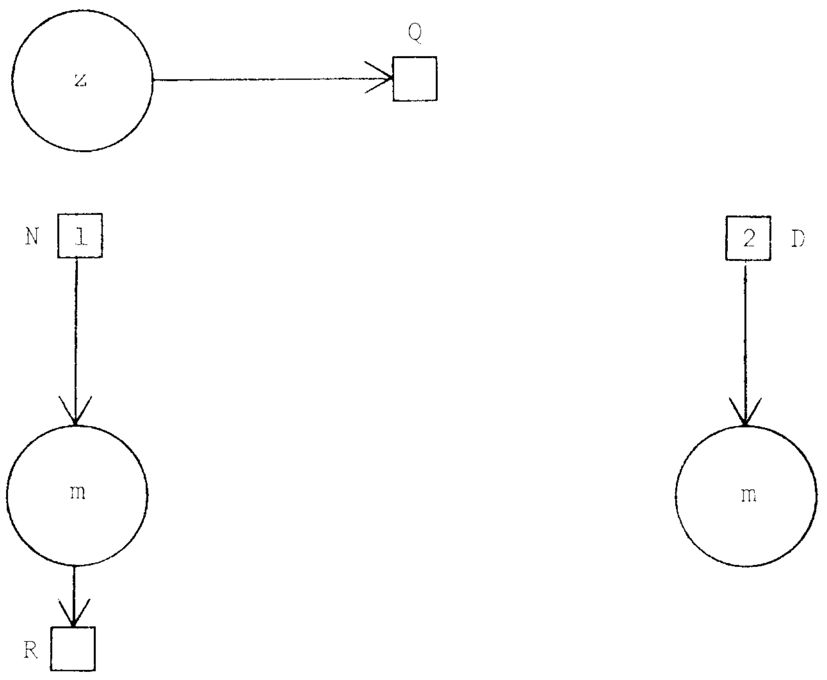


Figure 5.1g Dadep graph of $\overline{w}\underline{w}\overline{c}\overline{a}\overline{b}\underline{a}$

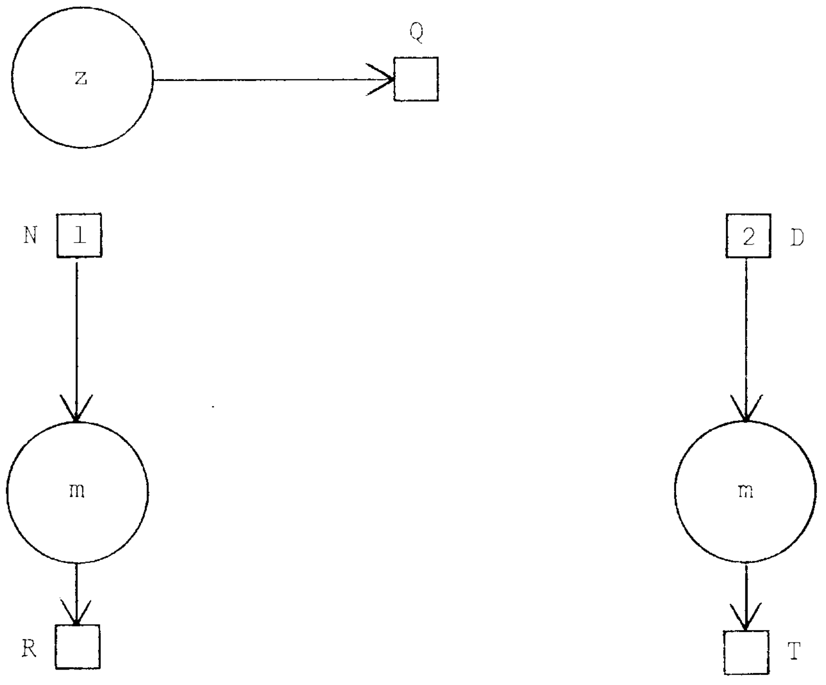


Figure 5.1h Dadep graph of $\overline{w}\underline{w}\overline{c}\overline{a}\overline{b}\underline{a}\underline{b}$

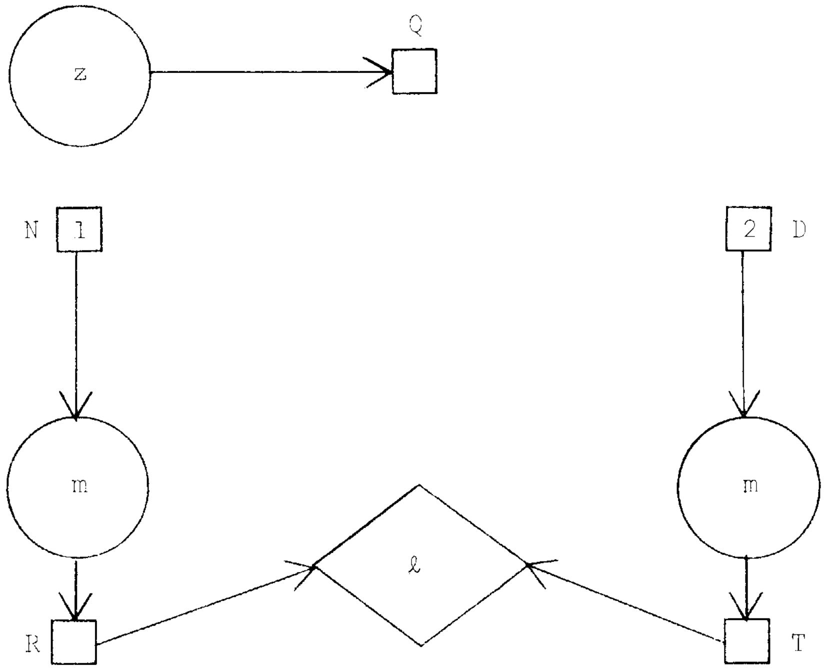


Figure 5.1i Dadep graph of $\bar{w}\underline{w}\bar{c}\bar{a}\bar{b}\underline{a}\underline{b}\bar{d}$

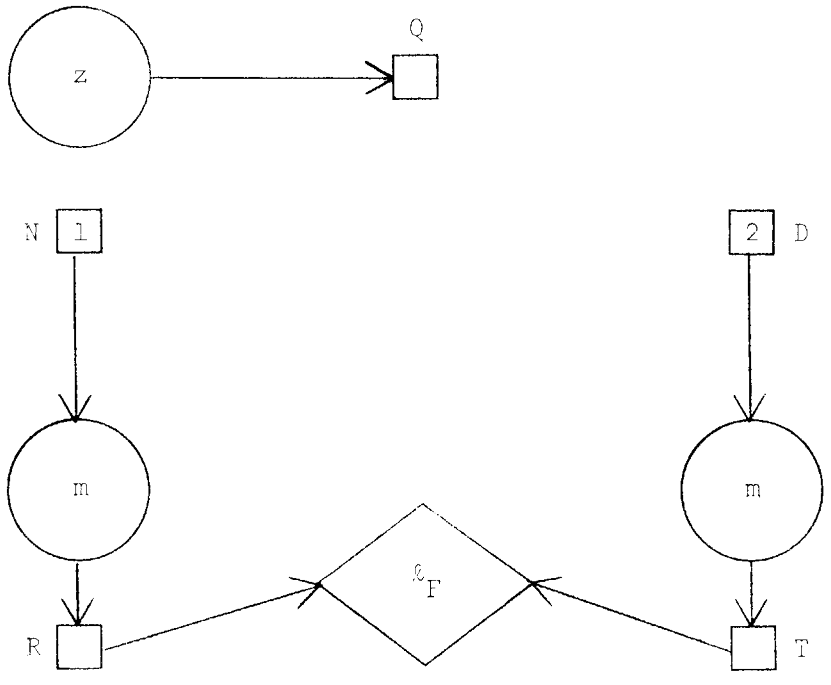


Figure 5.1j Dadep graph of $\bar{w}\underline{w}\bar{c}\bar{a}\bar{b}\underline{a}\underline{b}\bar{d}d_F$

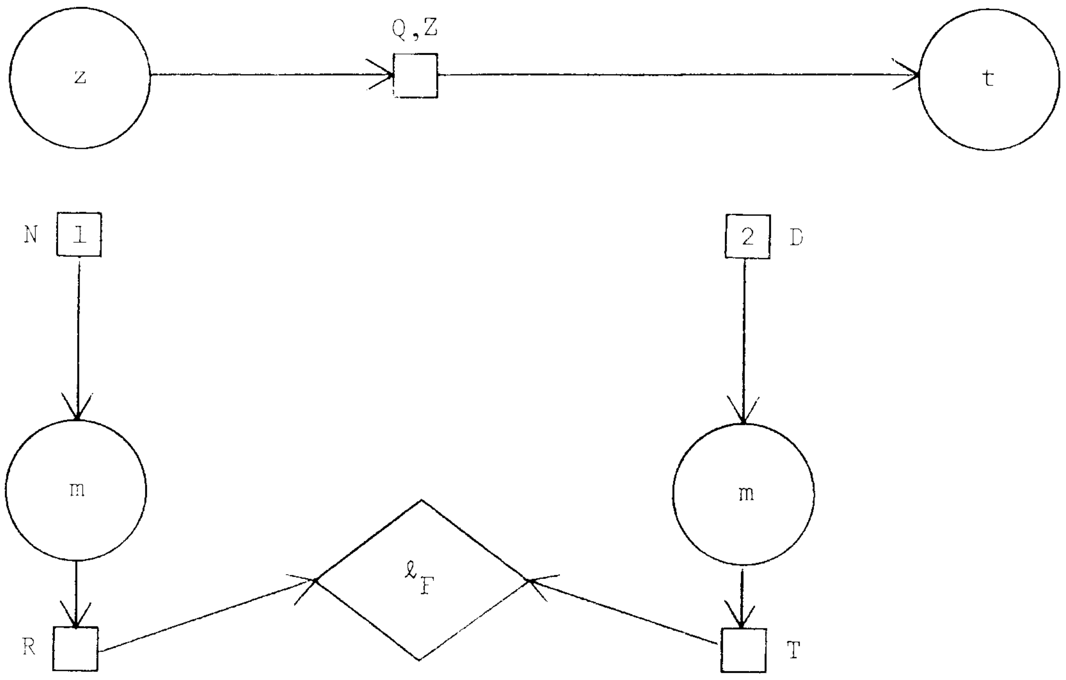


Figure 5.1k Dadep graph of $\bar{w}\underline{w}\bar{c}\bar{a}\bar{b}\underline{a}\underline{b}\bar{d}d_F\underline{c}\bar{f}$

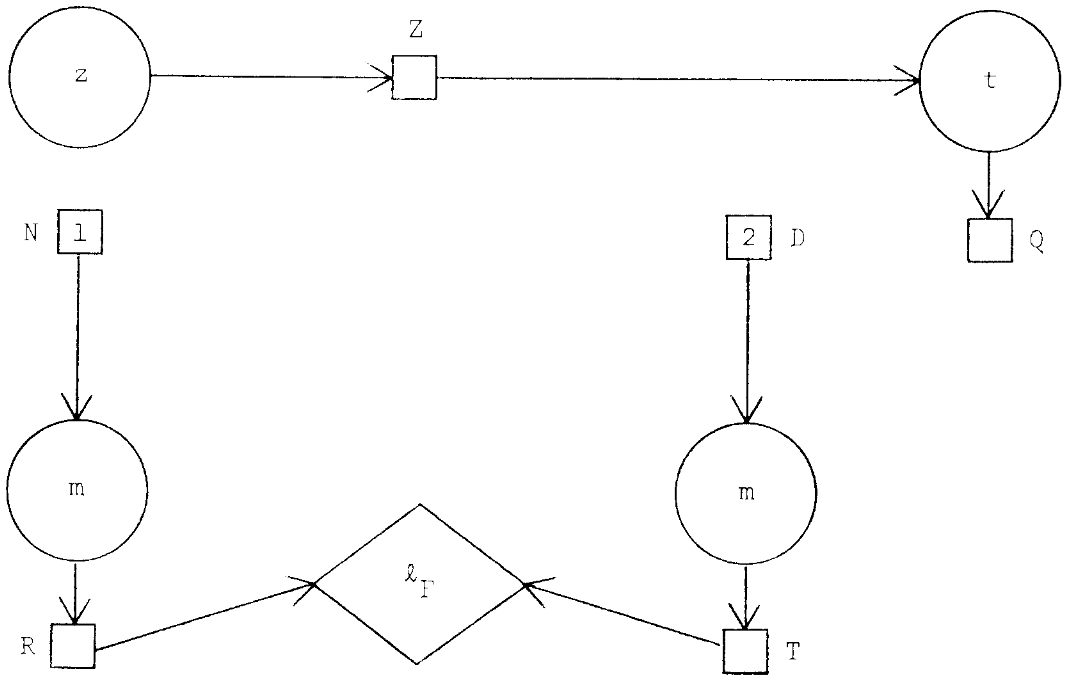


Figure 5.1ℓ Dadep graph of $\bar{w}\underline{w}\bar{c}\bar{a}\bar{b}\underline{a}\underline{b}\bar{d}d_F\underline{c}\bar{f}\underline{f}$

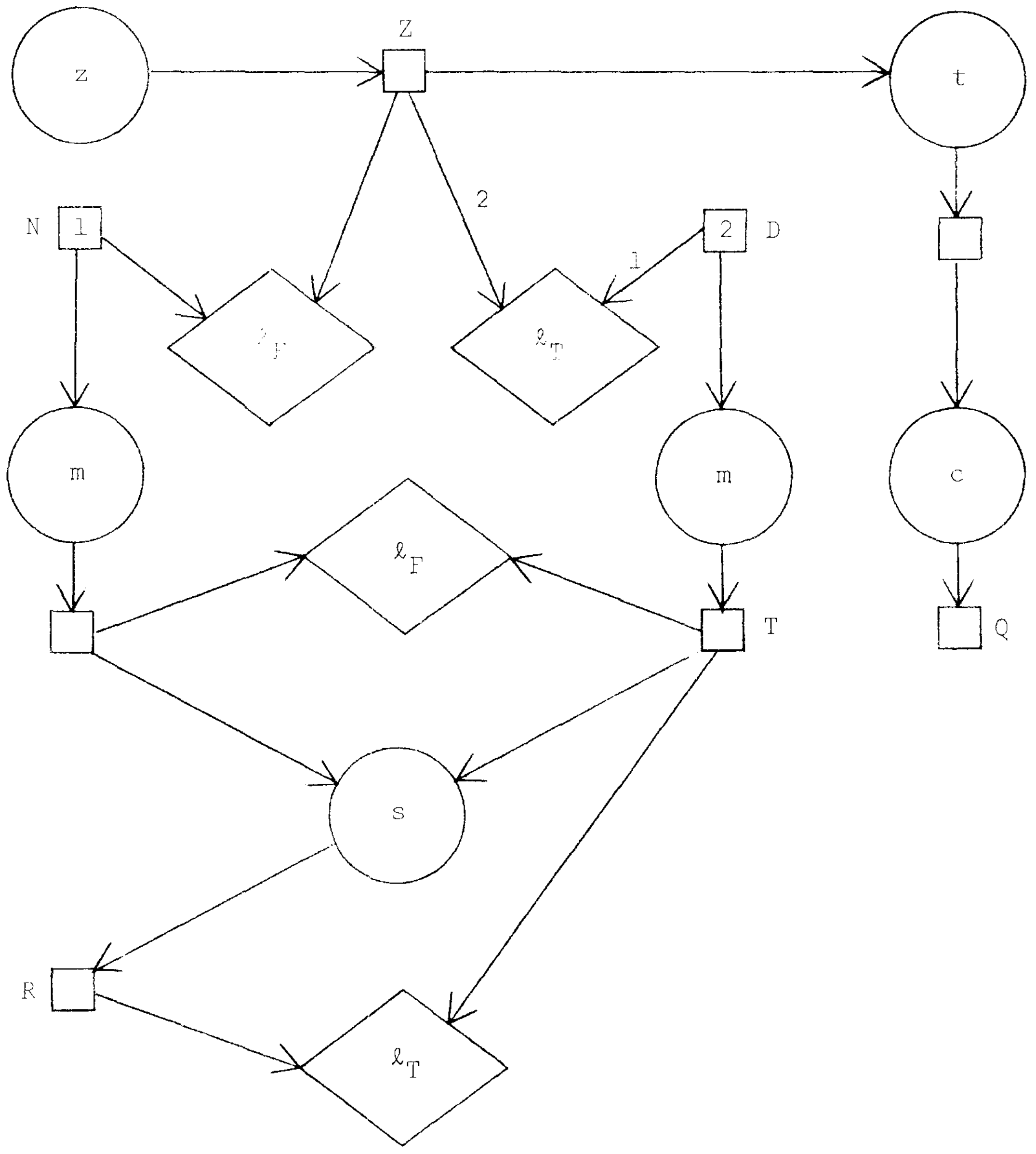


Figure 5.2 Dadep graph of x

6.1.1 Dadep Graphs - Definition of Depth

In the sequel, we will find it convenient to use certain structural properties of dadep graphs without having to consider the particular sequence and data flow graph which determines them. One such feature is the *depth* of values and actions in a dadep graph. The depth will simply be the length (in terms of number of arcs) of the longest directed path leading to the value or action.

Being a little more precise, we define the depth of all initial values to be 0. We also define the depth of all 0-ary operations in a dadep graph to be 0. The depth of any value other than an initial value is defined to be 1 plus the depth of the operation of which it is the output value. (Such an operation exists by hypothesis 3 of dadep graphs.) The depth of an operation which is not 0-ary is defined to be 1 plus the maximum depth of its input values. The acyclic nature of dadep graphs (hypothesis 2) guarantees the validity of this definition. Figure 6.1 shows a dadep graph with the depth of each node indicated.

Note that if there is a value [action] at depth k>1, there must be a value [operation] at depth k-2 on a path to it. Thus, there can be no "gaps" in the depths of values [actions] greater than 2.

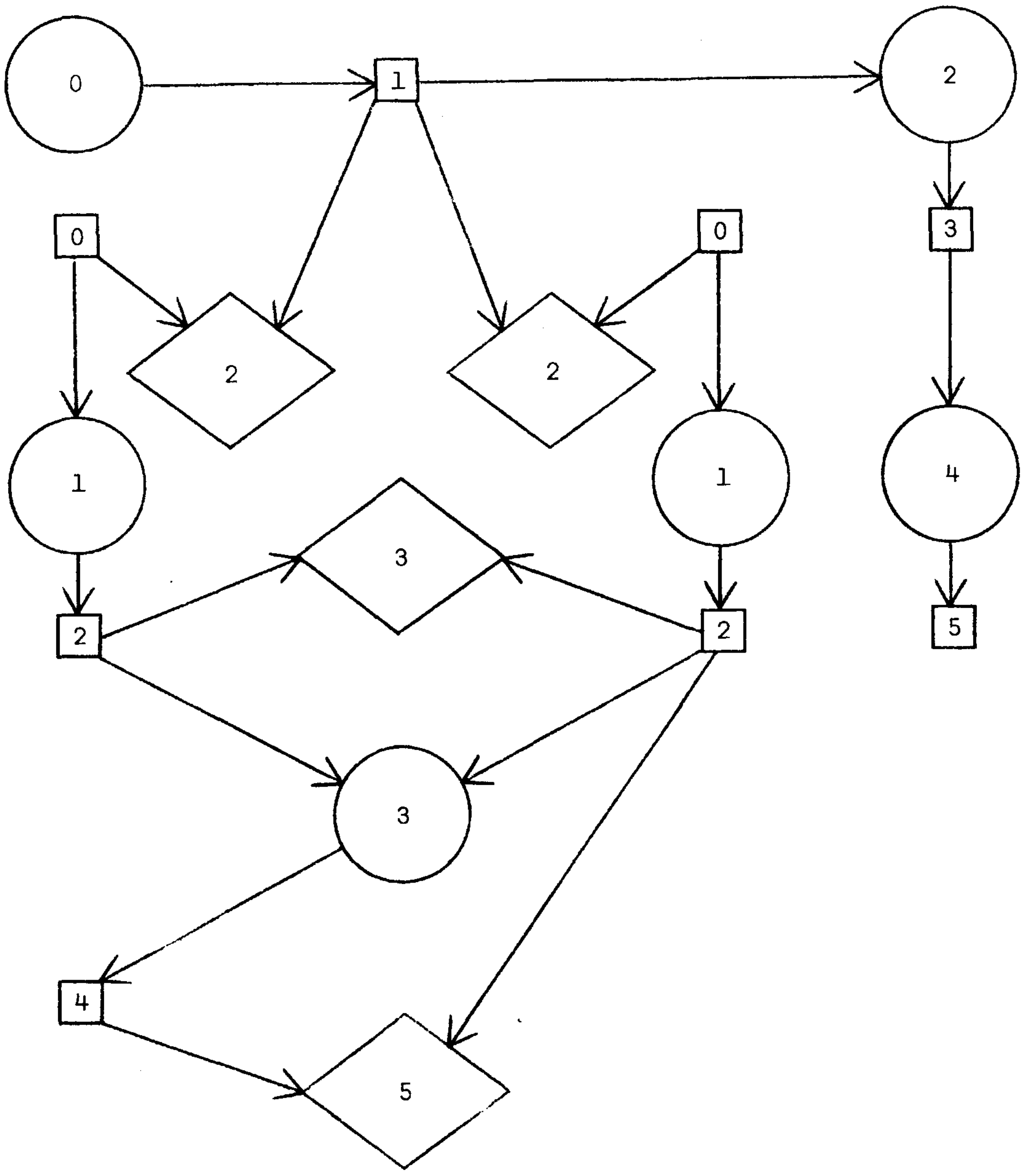


Figure 6.1 Depths of the nodes in the dadep graph of Figure 5.2

6.2.1 Interpreting Dadep Graphs

Any interpretation of a schema will supply all detail we need to associate specific domain elements with the values in a dadep graph. The rôle of decisions in the generation of values is indirect, and its explanation presupposes a thorough understanding of the nature of values and operations. We will therefore ignore, for now, the component of interpretations which specifies predicates, and focus on the domain, functions and initial elements.

These three components define an assignment of domain elements to values in the obvious way. Initial elements are paired with initial values and the functions determine all other assignments.

More formally, given a dadep graph D and an associated interpretation I, define a mapping ξ^I: VALUES $\rightarrow$ DOMAIN by induction on depth as follows.

Basis 1): If the depth of value v is 0, then v is an initial value, say the i^{th}. Define $\xi^I(v)$ to be the initial element of I corresponding to the i^{th} input location.

Basis 2): If the depth of v is 1, then v must be the output value of an operation with a 0-ary function name, say f. Define $\xi^I(v)$ to be the element $f^I()$ determined by I.

Induction step): Suppose ξ^I is defined at all values whose depth is k (>1) or less. If there is no value at depth k+1 or k+2, then ξ^I is defined for all values. (Recall that no gap greater than 2 can exist between depths of values.) Otherwise, consider a typical operation o with m-ary function name f, of which one of these values is an output value. The depth of o is either k or k+1 so the depths of o's input values, v_1, v_2, $\cdots$, v_m, are k or less. Hence, ξ^I is defined at each v_i and we define $\xi^I(v)$ to be $f^I(\xi^I(v_1),\cdots,\xi^I(v_m))$.

(The next induction step is at level k+2.) □

In Figure 6.2 we show how such an assignment can be made by combining the dadep graph of Figure 5.2 with the interpretation of Example 4.1. The element associated with each value is shown inside the box representing the value.

Such an interpreted graph shows a "history" of the elements generated if the given sequence can be observed under the given interpretation. (This "can be observed" issue is where the predicates will come in.) The labeling with memory locations then identifies the final contents of the schema output locations. Thus, such an interpreted dadep graph constitutes a completely satisfactory description of the input/output behavior of schemata for what we will define as *I-computations*.

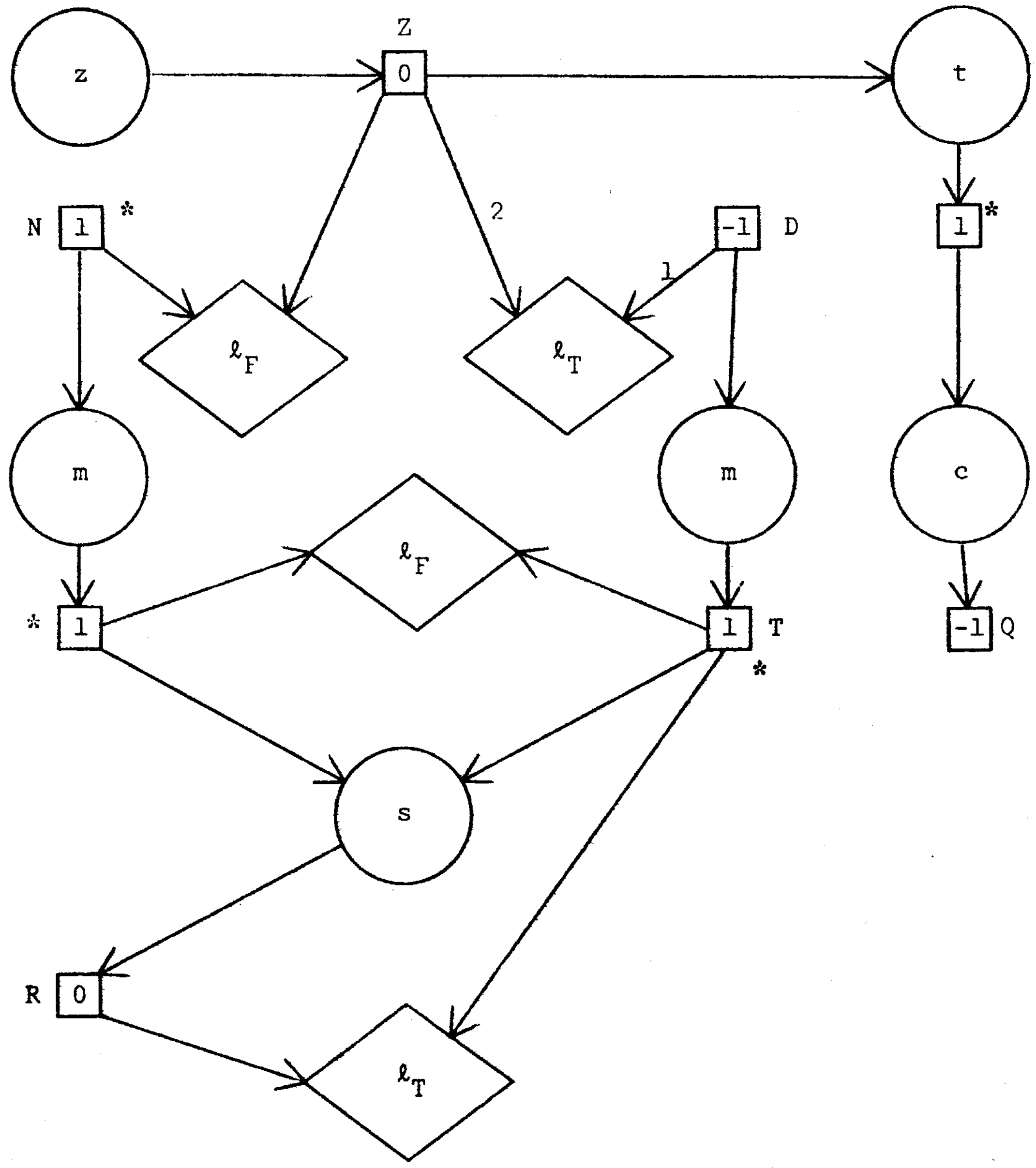


Figure 6.2 Assignment of elements to the values of the dadep graph of Figure 5.2 using the interpretation of Example 4.1

6.3.1 Free Interpretations - Introduction

The starred values in Figure 6.2 all were assigned the same domain element. We would like to know if this was purely coincidental, or if it will be true for all interpretations. This leads us to introduce *free* or *Herbrand* or *one-one interpretations*, and the notion of *similarity* in dadep graphs.

A brief inspection of the interpreted dadep graph of Figure 6.2 reveals that if a different set of initial elements had been chosen, say 5 and -3, then the starred values would have been distinct. It would be nice if we could find an interpretation which assigns the same domain element to distinct values only if <u>all</u> interpretations assign the same element to them. Such interpretations in fact exist, and are variously called *free, Herbrand,* or *one-one interpretations*. These interpretations are a formalization of the "symbolic representation" we referred to in the definition of events.

6.3.2 Free Interpretations - Formal Definition

To define a *free interpretation* of a schema, let DOMAIN be the set of strings over the "alphabet" composed of FUNCTION NAMES, the integers from 1 through the number of schema inputs, and, for clarity, commas and parentheses. The initial element corresponding to the first schema input location is '1', that corresponding to the second schema input is '2', and so forth.

If f is an m-ary function name, then f^I applied to domain elements (which are strings) s_1, s_2, $\cdots$, s_m is defined to be the string formed by concatenating function name f, a left parenthesis, s_1, a comma, s_2, a comma, and so on through s_m which is followed by a right parenthesis. For example, f^I['1','g(2)','h()'] = 'f(1,g(2),h())'.

Any set of total predicates over the given DOMAIN can be used to complete the interpretation. Thus, a data flow graph really has a family of free interpretations, differing only on the predicates assigned to the predicate names.

6.3.3 Free Interpretations - Universality

Suppose G_1 and G_2 are data flow graphs with the same number of schema inputs. Let D_1 and D_2 be dadep graphs of well-defined sequences for the respective data flow graphs. Then values v_1 of D_1 and v_2 of D_2 are assigned the same element by every interpretation if and only if they are assigned the same element of a free interpretation.

The 'only if' half of the above result is trivial. Therefore, we need only demonstrate that if v_1 and v_2 are assigned the same element by a free interpretation, then every interpretation assigns them the same element. We assume the contrary and derive a contradiction.

Suppose that it is possible for values v_1 and v_2 to be assigned different elements by some interpretation I even though they are assigned the same element by a free interpretation. We can assume without loss of generality that v_1 is a value of least depth for which such a condition can arise.

If the depth of v_1 is 0, it must be an initial value, say the i^{th}. Since v_2 is assigned the same element 'i' by a free interpretation, it must be the i^{th} initial value in D_2. But then the same initial element of every interpretation is assigned to both v_1 and v_2. So v_1 must be at depth 1 or greater.

If v_1 is at depth 1, then it is the output value of an operation with some 0-ary function name f. The element of a free interpretation assigned to v_1 is therefore 'f()'. If v_2 is assigned this same element by a free interpretation, it too must be the output value of an operation with 0-ary function name f. Since $f^I()$ would then be assigned to both v_1 and v_2 for every interpretation I, it follows that v_1 must be at depth 2 or greater.

It must therefore be that v_1 is the output value of an operation o_1 with m-ary function name g, and $m \geq 1$. Therefore v_1 and v_2 are assigned element '$g(s_1, s_2, \cdots, s_m)$' by a free interpretation, where s_i is the element assigned by the free interpretation to the i^{th} input value, v_{1i}, of o_1. Thus v_2 must also be the output value of an operation o_2 with function name g and input values v_{21} through v_{2m} which are assigned elements s_1 through s_m of the free interpretation. But each v_{1j} has path depth at least 2 less than v_1. By hypothesis, each v_{1j}, being assigned the same element s_j of a free interpretation as is assigned to v_{2j}, must be assigned the same element as v_{2j} for every interpretation. But then v_1 and v_2 could be assigned differing elements only if g^I gave different results for the same m-tuple of elements. This violates the functionality of g^I.

We conclude that an interpretation which assigns different elements to v_1 and v_2 cannot exist. □

6.4.1 Similarity - Formal Definition

Suppose we have two data flow graphs, G_1 and G_2, and dadep graphs D_1 and D_2 of corresponding well-defined sequences over the respective data flow graphs. We define *similarity*, a relation on actions and on values, by the following induction.

Values v_1 in D_1 and v_2 in D_2 are *similar* if each is the i^{th} initial value, or if they are output values of *similar* operations.

Actions a_1 in D_1 and a_2 in D_2 are *similar* if they are labeled with the same function name or predicate name, and, if the name is m-ary with $m \geq 1$, then for $1 \leq i \leq m$, the i^{th} input value of a_1 is *similar* to the i^{th} input value of a_2.

Again, the acyclic nature of dadep graphs guarantees the validity of this inductive definition. That is, working back along directed arcs always leads to an initial value or a 0-ary operation. Values or actions which are not similar are said to be *dissimilar*.

6.4.2 Similarity - Relation to Free Interpretations

We now show that values are similar if and only if they are assigned the same element by free interpretations.

If values v_1 and v_2 are similar and the lesser depth is 0, then one of the values, say v_1, is the i^{th} initial value. By hypothesis 3 of dadep graphs and the definition of similarity, v_2 must also be the i^{th} initial value of its dadep graph. Both values are thus assigned 'i' by a free interpretation.

Suppose similar values are assigned the same element by a free interpretation if the lesser depth of the values is k or less. Suppose

v_1 and v_2 are similar values such that the depth of v_1 in its dadep graph is k+1 or k+2. Since v_1 and v_2 are similar and neither is an initial value, v_1 and v_2 are output values of similar operations o_1 and o_2. Since o_1 and o_2 are similar, they bear the same m-ary function name f. If m is 0, v_1 and v_2 are both assigned 'f()' by a free interpretation. If m is positive, then, by definition, the i^{th} input value of o_1 is similar to the i^{th} input value of o_2 for $1 \le i \le m$. Since the depth of each input value of o_1 is k or less, it follows by our induction hypothesis that the i^{th} input value of o_1 is assigned the same element, s_i, of a free interpretation as is the i^{th} input value of o_2. But then v_1 and v_2 are both assigned '$f(s_1,s_2,\cdots,s_m)$' by any free interpretation.

The proof that values assigned the same element by a free interpretation are similar can be carried out by induction on lesser depth in a manner directly analogous to the proof above. We therefore omit the details of the proof. □

Combining the last two proofs allows us to draw the following important conclusion.

Theorem 6.1): Values v_1 and v_2 in dadep graphs D_1 and D_2 are assigned the same element by all interpretations if and only if the values are similar.

6.5.1 Dadep Graphs - The Rôle of Decisions

Having investigated in some detail the nature of values in dadep graphs and the relation they bear to interpretations, we are now in a position to study decisions.

As we saw, decisions are irrelevant with respect to the assignment of elements to values by any interpretation. However, the decisions in a dadep graph indicate whether the graph (and underlying sequence) are in some sense legitimate.

For example, if ℓ were interpreted as "larger than or equal to" instead of as "less than" in Figure 5.2, the predicates determined by the interpretation would have specified the opposite outcomes from those in the dadep graph. Thus, the dadep graph is not consistent with such an interpretation.

We will formalize this notion of *consistency* in several steps. We will first define what it means for a single decision to be consistent with an interpretation. This easily extends to the definition of dadep graphs being consistent with an interpretation. It will then be possible to define what we mean by sequences, even infinite ones, being consistent with an interpretation. We can then circumvent interpretations and define consistency between decisions, dadep graphs and sequences. Throughout, consistency has the flavor of "agreeing on the outcome of identical decisions".

6.5.2 Consistency - Formal Definition

Suppose we have a dadep graph D and a decision d in D. Suppose d is labeled with predicate name p and one of the outcome labels, T or F. Then decision d is said to be *inconsistent* with interpretation I if p^I on the elements assigned to the input values of d has the outcome opposite that with which d is labeled. In any other case, including the case where d has no outcome label, d is said to be *consistent* with I.

A dadep graph D is said to be *consistent* with an interpretation I if each decision in D is consistent with I. D is *inconsistent* with I if any decision in D is inconsistent with I.

We have defined dadep graphs only for finite, well-defined sequences over a given data flow graph. Although it would be possible to extend the definition to infinite well-defined sequences, we are not as interested in the details of such infinite sequences as we are in the fact that they never terminate. To handle (potentially infinite) sequences, then, we make the following definition.

A well-defined sequence is *consistent* with an interpretation I if the dadep graph of every prefix of the sequence is consistent with I. Otherwise, the sequence is said to be *inconsistent* with I.

A control sequence consistent with an interpretation I can be thought of as a computation which might be observed for the schema under interpretation I. In this spirit, we call them *I-computations*.

If d_1 is a decision in dadep graph D_1 and d_2 a decision in D_2, then we define d_1 to be *consistent* with d_2 if there is some interpretation with which both d_1 and d_2 are consistent. Otherwise we say d_1 is *inconsistent* with d_2. From our knowledge of similarity and interpretations, we know that d_1 and d_2 are inconsistent if and only if they are similar and are labeled with opposite outcomes.

A dadep graph is said to be *(self-)consistent* if every decision in the graph is consistent with every other decision in the graph. It is not difficult to see that a dadep graph is consistent if and only if there is an interpretation with which it is consistent.

Dadep graph D_1 is defined to be *consistent* with dadep graph D_2 if each is self-consistent and each decision in D_1 is consistent with every decision in D_2. (It follows automatically that each decision in D_2 is then consistent with every decision in D_1.) Because we required that each dadep graph be self-consistent, it follows that D_1 is consistent with D_2 (having the same number of initial values) if and only if there is some interpretation with which both are consistent. If D_1 and D_2 are not consistent we say they are *inconsistent*.

A control sequence is *(self-)consistent* if the dadep graph of every prefix of the sequence is self-consistent. Control sequence x_1 of schema S_1 is *consistent* with control sequence x_2 of S_2 (having the same number of schema inputs) if both are self-consistent and the dadep graph of every prefix of x_1 is consistent with the dadep graph of each prefix of x_2. Therefore, two control sequences are consistent if and only if there is some interpretation with which both are consistent.

6.5.3 Consistency - Some Perspective

In sequential schemata such as Paterson's flow chart schemata (which we henceforth simply call Paterson schemata to avoid confusion with data flow graphs), an interpretation completely determines what will occur. Only at decider outcomes is there any "choice" in a sequential schema, and the outcome of all decisions is fixed by an interpretation.

When we move to parallel schemata, the choice of interpretation no longer determines which event must occur at each step in a computation. For example, in the Petri net model of Figure 3.4, the first event to occur might be $\bar{w}$, $\bar{a}$, or $\bar{b}$, no matter what interpretation is involved. (Which of these events actually occurs first might, for some particular implementation, depend upon such things as the availability of a suitable processor or the details of a scheduling algorithm. These issues are not dealt with by our model: We are only concerned that the control set might allow any of several events to occur.) Although an interpretation specifies the outcomes of all decisions, for parallel schemata this no longer uniquely determines a control sequence. The choice of an interpretation I only restricts control sequences to that subset consistent with I. Within this set of I-computations, any sequence might be observed for the schema under interpretation I.

As mentioned, the self-consistent sequences are precisely those which are consistent with some interpretation. This subset of CONTROL is sufficiently important that we give it a name, EXECUTION SEQUENCES. A control sequence which is not an execution sequence is, in one sense, uninteresting since it cannot be observed under any interpretation. It may not be possible to uniformly eliminate inconsistent sequences, however.

6.6.1 Equivalence of Sequences - Formal Definition

We can now define a meaningful comparison between control sequences of the same or differing schemata. Our definition of equivalent sequences will amount to producing the same, if any, outputs for all interpretations. We find it useful not to insist that equivalent sequences be consistent.

Formally, suppose we have control sequence x_1 of schema S_1 having m_1 schema inputs and n_1 schema outputs, $\{\ell_{11},\ell_{12},\cdots,\ell_{1n_1}\}$. Suppose also that x_2 is a control sequence of schema S_2 having m_2 inputs and n_2 outputs $\{\ell_{21},\ell_{22},\cdots,\ell_{2n_2}\}$. Then x_1 and x_2 are *(output) equivalent* if

1) $m_1=m_2$ and $n_1=n_2$, and

2a) both x_1 and x_2 are infinite, or

2b) both x_1 and x_2 are finite, and, for all interpretations I, and for all $1\le i\le n_1$, the element assigned by I to the value with memory label ℓ_{1i} in the dadep graph of x_1 is the same as the element assigned to the value with label ℓ_{2i} in the dadep graph of x_2.

An immediate corollary of Theorem 6.1 is that clause 2b) can be replaced with

2b') both x_1 and x_2 are finite, and, for all $1\le i\le n_1$, the value with label ℓ_{1i} in the dadep graph of x_1 is similar to the value with label ℓ_{2i} in the dadep graph of x_2.

We will generally use this second form which makes no explicit mention of interpretations. We write $x_1 \equiv x_2$ to denote that x_1 is equivalent to x_2.

6.7.1 Determinacy and Equivalence of Schemata - Formal Definition

A schema S is *determinate* if, for all interpretations I, all I-computations are equivalent sequences. Alternatively, schema S is determinate if, for all control sequences x_1 and x_2 such that x_1 is consistent with x_2, x_1 and x_2 are equivalent sequences. That the two definitions are equivalent follows from the fact that x_1 and x_2 are consistent if and only if there is an interpretation I such that both are I-computations.

This form of determinacy is sometimes called *output determinacy* or *output functionality*, since it deals only with the final contents of output locations. Since it is the only form of determinacy we will treat, we will use the simpler term.

Two schemata S_1 and S_2 are *equivalent* if, for all interpretations I, if x is an I-computation of one schema, then there exists an equivalent I-computation y of the other schema. Again, the definition can be restated so that interpretations do not appear. In particular, S_1 and S_2 are equivalent if, for each execution sequence x of one schema, there is a consistent, equivalent execution sequence y of the other.

Note that as stated, schemata need not be determinate for equivalence to apply. It is an easy exercise to show that two schemata are equivalent only if both are determinate or both are non-determinate.

For determinate schemata, other definitions of equivalence are also seen in the literature. We can define two determinate schemata to be *weakly equivalent* if, for all interpretations I, if both schemata have finite I-computations, then these sequences are equivalent. There is no natural analogue to weak equivalence among non-determinate schemata.

6.8.1 Schemata - Some Perspective

Schemata as we have defined them have been the result of taking the common notion of a computation and splitting it into three parts. One part, the data flow graph, specified structural relationships between memory locations and actors, and indicated where inputs and outputs were expected. Another part, the interpretation, supplied the detailed information about the potential contents of memory locations and the effect of actors on these contents. The third part, the control set, specified allowable sequences of actor initiations and terminations.

Dadep graphs were defined as a representation of a sequence and a data flow graph. We showed the "universality" of free interpretations, and the relationship of similarity in dadep graphs to equality in free interpretations. This enabled us to show that the structure of dadep graphs also includes "all we need to know" about interpretations. That is, the definitions of schemata determinacy and equivalence obtained by quantifying over all interpretations can also be stated in terms of dadep graph structure without explicit mention of interpretations.

Thus dadep graphs constitute a powerful investigative tool combining all three of the parts mentioned above. They will find considerable use in the remainder of this dissertation.

7.1.1 Schematology - Some Pitfalls

It has long been known that the equivalence problem for Turing machines is not decidable[10,14]. Since one can encode a Turing machine in most interpreted models for computation[10], the equivalence problem for these models is also undecidable. An early hope for schematology was that by demanding equivalence for <u>all</u> interpretations, these encoding tricks could be circumvented and the schemata equivalence problem found solvable.

Unfortunately, the equivalence problem for Paterson schemata has been shown to be undecidable[9,11]. Since the schemata we have defined can mimic Paterson schemata, it follows that the general equivalence problem for our schemata must also be unsolvable.

As a further consequence, we can show that the general determinacy problem is also undecidable. To see this, suppose we have two m-input n-output Paterson schemata S_1 and S_2 whose equivalence we would like to test. We can assume that $ACTORS_1 \cap ACTORS_2 = \phi$, since actors can be relabeled without altering function names or predicate names. We can then form a composite schema by merging the schema inputs and schema outputs of the data flow graphs as shown in Figure 7.1, and letting the new control set be $CONTROL_1 \cup CONTROL_2$.

It is a property of Paterson schemata that each interpretation I defines exactly one I-computation. Thus, our composite schema would have exactly two I-computations, one from S_1 and one from S_2. The composite schema will be determinate, then, if and only if S_1 and S_2 are equivalent. Since the latter is undecidable, so is the former.

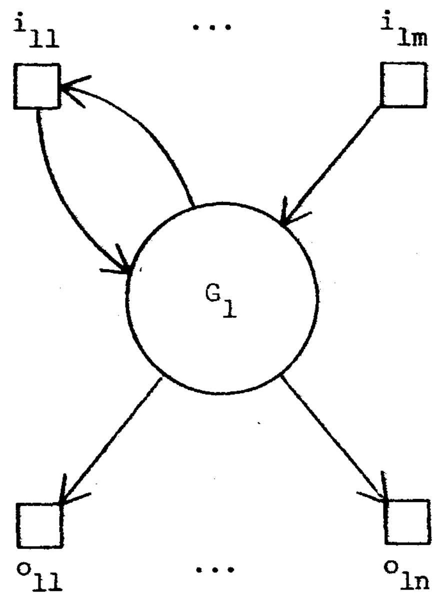


Figure 7.1a Data flow graph of S_1

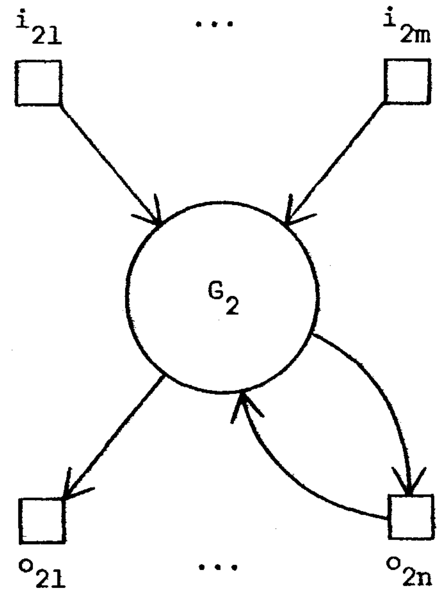


Figure 7.1b Data flow graph of S_2

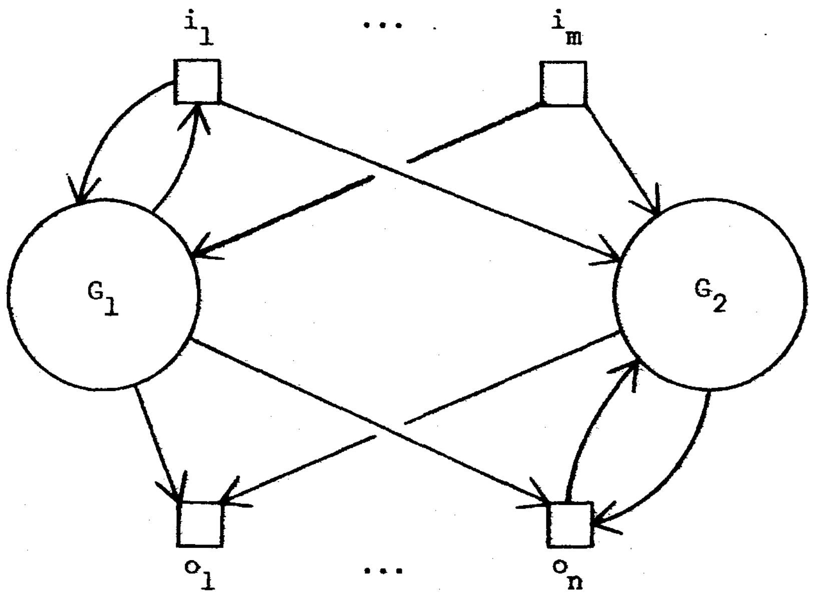


Figure 7.1c Data flow graph of composite schema

7.2.1 Properties of CONTROL - Introduction

Knowing that the general equivalence and determinacy problems are undecidable, we will concentrate our efforts on identifying classes of schemata where the problems are more tractable. We define these classes by putting "syntactic" restrictions on control sets. So far, the only restrictions on control sets of schemata are that the sequences be well-defined and that finite control sequences leave results in the schema output locations. As we shall see, this allows control sets which run counter to our intuition about "proper" control. After developing some familiarity with control sets, we will begin restricting them to obtain more reasonable classes.

If σ is an event and x is a prefix of a control sequence, then we say that σ is *enabled* after prefix x if $x\sigma$ is also a prefix of a control sequence. One must be careful not to read more into this definition than is really there. Without further details about the mechanism which determines the control set, all we can conclude is that after x has occurred, there is nothing to <u>prevent</u> σ from happening next. Of course, $x\sigma$ may not be consistent with a given interpretation, or, for that matter, with any interpretation. In parallel schemata, we generally expect that many events may be enabled after a given prefix.

If a control mechanism allows two events σ_1 and σ_2 to occur "simultaneously" after prefix x, this will be manifested in the existence of prefixes $x\sigma_1\sigma_2$ and $x\sigma_2\sigma_1$. The converse is not strictly true. Figure 7.2 shows two Petri net controls each of which allows sequences $\{\sigma_1\sigma_2,\sigma_2\sigma_1\}$. Only in the first can the events be truly simultaneous. To us, however, the distinction is not an important one.

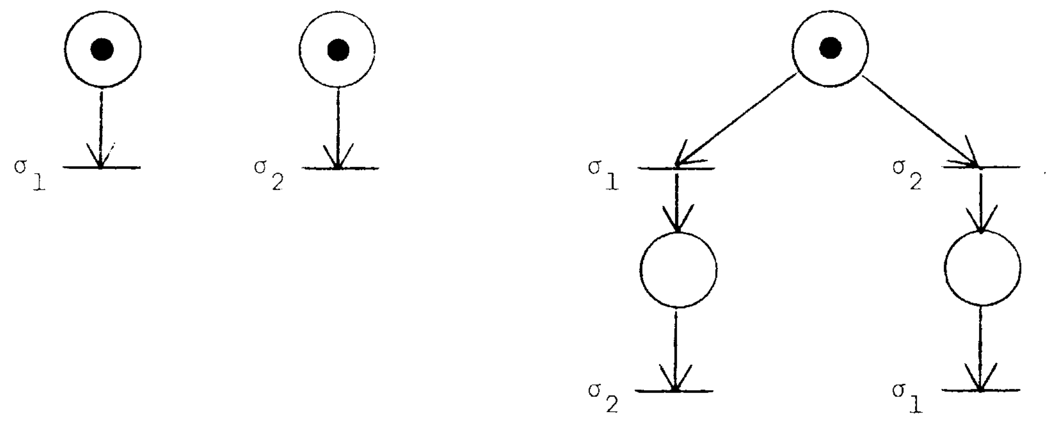


Figure 7.2 Two Petri nets with CONTROL = $\{\sigma_1\sigma_2, \sigma_2\sigma_1\}$

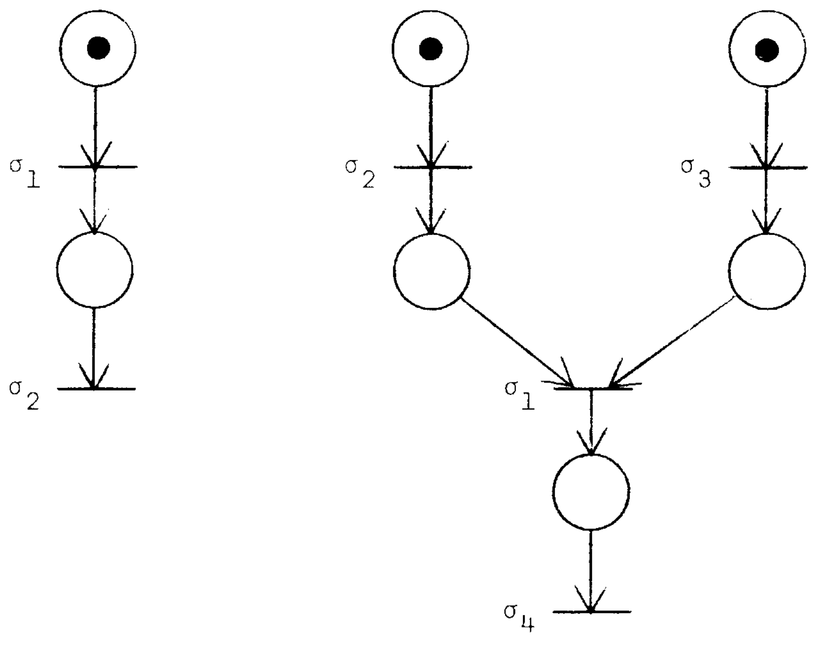


Figure 7.3 A Petri net control specification which is not commutative

7.3.1 CONTROL - The Prefix Property

One property we might expect of a reasonable control mechanism is that it be able to detect when a computation had completed. Since we are dealing with speed-independent systems, we must rely on the control mechanism to announce when things have come to a halt and results are ready in the schema outputs.

The *prefix property* guarantees that when a computation terminates, no further activity is possible. Stated formally, a control set has the *prefix property* if no control sequence is the proper prefix of any other control sequence. In other words, after a finite control sequence, nothing is enabled.

When dealing with a particular control mechanism, the control sequences are usually so defined as to have the prefix property automatically. This bears out our comment about the set-theoretic approach appearing to formalize the trivial. However, as the single state control mechanism of Figure 3.8 shows, what is obvious in some mechanisms may be absent in others.

7.4.1 CONTROL - Persistence

In the proof of the undecidability of determinacy, the control set of the composite schema had some peculiar properties. The composite control set was formed by taking the union of two control sets of sequential schemata whose alphabets of events were presumed disjoint. Initially, then, there would be two events enabled, corresponding to the first event from each of the component schemata. As soon as an event occurs, however, one of the component schemata becomes "irrelevant" since it has no control sequences starting with that event. The event originally enabled in this component schema ceases to be enabled. Note that this happens, not because of an inconsistency with an interpretation, but because "something clicks" in the control set and precludes any further activity.

If the reader feels that something is amiss here, it is probably because most of the familiar models for parallel computation have a property we call *persistence* which is lacking in the composite schema. In a persistent schema, once an event is enabled, it remains enabled until it occurs, or, in the case of a decider termination, the termination with opposite outcome occurs.

Formally, a schema is *persistent* if, given any prefixes $x\sigma_1$ and $x\sigma_2$ of control sequences such that $\sigma_1 \neq \sigma_2$ and σ_1 and σ_2 are not opposite terminations of the same decider, then $x\sigma_1\sigma_2$ is also the prefix of a control sequence. (By symmetry, $x\sigma_2\sigma_1$ would also be a prefix.)

Sequential schemata are trivially persistent since the only distinct prefixes of the form $x\sigma_1$ and $x\sigma_2$ are those for which σ_1 and σ_2 <u>are</u> opposite terminations of a decider. Fork/join formalisms are also persistent since the loci of control are independent except at join nodes. Petri nets are persistent if multiple arcs out of state nodes lead only to alternative termination events of a decider. (This is a sufficient condition, but not a necessary one, as demonstrated in Figure 7.2.) Dennis-Fosseen schemata are always persistent since firing any enabled element cannot disable any other element. Finite state control mechanisms need not be persistent, but the property can easily be verified by inspecting the state graph.

7.5.1 CONTROL - Commutativity

Persistence implied that the occurrence of one event could not disable another event. Thus, if $x\sigma_1$ and $x\sigma_2$ are consistent, distinct prefixes, $x\sigma_1\sigma_2$ and, reversing roles, $x\sigma_2\sigma_1$, are also prefixes. *Commutativity* will imply that if two events can occur in either order, as with σ_1 and σ_2 after prefix x above, then the actual order of occurrence is unimportant to the control mechanism. This "unimportance" can be stated in syntactic terms by requiring that anything which can happen after $x\sigma_1\sigma_2$ can also happen after $x\sigma_2\sigma_1$, and vice versa.

Formally, a schema is *commutative* if, for all prefixes $x\sigma_1\sigma_2$ and $x\sigma_2\sigma_1$ of control sequences, $x\sigma_1\sigma_2y$ is a control sequence if and only if $x\sigma_2\sigma_1y$ is a control sequence.

Sequential schemata are trivially commutative. Petri net and fork/join formalisms are not necessarily commutative. We show a Petri net counterexample in Figure 7.3. A fork/join counterexample would look much the same. Both $\sigma_3\sigma_1\sigma_2$ and $\sigma_3\sigma_2\sigma_1$ are prefixes, but $\sigma_3\sigma_2\sigma_1\sigma_4$ is a prefix whereas $\sigma_3\sigma_1\sigma_2\sigma_4$ is not. The problem here is that σ_1 in the two prefixes corresponds to different events in the Petri net. If events cannot be "enabled in parallel with themselves" as above, then fork/join controls and Petri nets, with branching out of state nodes restricted as in the last section, will be commutative. Dennis-Fosseen schemata are always commutative. If a finite state machine is in reduced form, commutativity can easily be verified by checking that $\sigma_1\sigma_2$ and $\sigma_2\sigma_1$ lead to the same state from any given state in the state diagram.

Commutativity is common because is requires more "states" to recall the order of events than to ignore it. Thus, commutative control mechanisms tend to be simpler than non-commutative ones. Furthermore, since sequences of the form $x\sigma_1\sigma_2$ and $x\sigma_2\sigma_1$ generally reflect the situation that σ_1 and σ_2 can occur simultaneously, there are practical (and philosophical) difficulties in actually determining the order of occurrence.

7.6.1 CONTROL - Conflict

Conflict, or the absence thereof, is a joint property of control sets and data flow graph topology. A *conflict* is said to exist if there are prefixes $x\underline{o}$ and $x\sigma$ of control sequences where either

1) σ is the initiation of some actor a such that the output location of operator o is an input location of actor a, or
2) σ is the termination of some operator $r \neq o$ such that r and o have the same output location.

In either case, we say that $x\underline{o}$ and $x\sigma$ are *in conflict*. A schema is said to be *conflict-free* if its control set has no conflicts.

A conflict exists, then, if two events can be enabled after the same prefix and both are about to write into the same location or one is about to write into a location the other is about to read. If the reader senses something dangerous here, his fears are not groundless as will become clear in the following sections.

7.7.1 Determinacy - An Overview

A schema which is persistent, commutative and conflict-free and has the prefix property may nevertheless exhibit properties which are quite unconventional. For example, it may be that many initiated actors are left unterminated when computations complete, or there may exist interpretations with which <u>no</u> control sequence is consistent. In view of such unorthodox possibilites, it might be surprising that such schemata are always determinate! Those familiar with the work of Karp and Miller will note the similarity of the proof of this fact to an analogous result for their schemata[7].

7.8.1 Determinacy - Some Preliminary Notions

The rules for constructing dadep graphs are such that distinct sequences need not generate dadep graphs which are different from one another. To make this statement precise, we must define what we mean by two sequences generating the "same" dadep graph.

Suppose x and y are finite, well-defined sequences over a data flow graph G. Let D_x and D_y be the dadep graphs of x and y, respectively. Then x and y are *dadep indistinguishable* if the following four conditions hold.

1) For each event σ in the alphabet of events of G, the number of occurrences of σ in x equals the number of occurrences of σ in y.

2) If there is a j^{th} occurrence of uninterpreted actor a in x, then the action in D_x corresponding to the j^{th} occurrence of a is similar to the action in D_y corresponding to the j^{th} occurrence of a, and, if a

is a decider, both decisions have the same outcome label, if any.

3) If there is a jth occurrence of identity operator o in x, then the value in D_x corresponding to the jth occurrence of o is similar to the value in D_y corresponding to the jth occurrence of o.

4) If ℓ is a memory location label on a value in D_x, then ℓ is a label on a similar value in D_y.

Condition 1 simply means that the dadep indistinguishable sequences are made up of the same events, possibly occurring in a different order. Conditions 2, 3, and 4 together imply that the differences in order do not cause any significant differences in the values, actions, or memory labeling in the dadep graphs.

For the data flow graph of Figure 2.4, each pair of sequences in the following set are dadep indistinguishable.

$\{ \bar{w}\underline{w}\bar{a}\underline{a}\bar{b}\underline{b} , \bar{w}\bar{a}\underline{w}\underline{a}\bar{b}\underline{b} , \bar{w}\bar{a}\underline{a}\underline{w}\bar{b}\underline{b} , \bar{w}\bar{a}\underline{a}\bar{b}\underline{w}\underline{b} , \bar{w}\bar{a}\underline{a}\bar{b}\underline{b}\underline{w} , \bar{w}\underline{w}\bar{a}\bar{b}\underline{a}\underline{b} , \bar{w}\bar{a}\underline{w}\bar{b}\underline{a}\underline{b} \}$

A little combinatorial mathematics reveals that N operators in parallel can occur in $(2N)! / (2^N)$ distinct sequences. Thus, the three operators, w, a, and b, determine 90 such dadep indistinguishable sequences, a considerable gain in flexibility over the single sequence of sequential schemata.

If two execution sequences are dadep indistinguishable, condition 4 immediately guarantees that the sequences are equivalent, and condition 2 ensures that they are consistent.

Equally as important, if xz and yz are finite control sequences and x is dadep indistinguishable from y, then xz is dadep indistinguishable from yz. This follows by induction from the fact that $x\sigma$ and $y\sigma$ are dadep indistinguishable if x and y are. This, in turn, can easily be verified by considering the five cases involved in the construction of dadep graphs. We will carry out the proof here, but will avoid such detailed expositions in the future.

Lemma 7.1: If x and y are dadep indistinguishable sequences and $x\sigma$ is well-defined, then $x\sigma$ and $y\sigma$ are dadep indistinguishable.

Proof: Condition 1 of dadep indistinguishability for $x\sigma$ and $y\sigma$ follows immediately from Condition 1 applied to x and y. If σ is the termination of some decider d, let j be the number of occurrences of terminations of d in $x\sigma$. Otherwise, let j be the number of occurrences of σ in $x\sigma$. Let D_x and D_y be the dadep graphs of x and y, respectively. We consider the following cases.

Case 1) σ is the initiation of an actor c other than an identity operator. By condition 4, the values in D_x which bear the names of the input locations to c are similar to the values bearing these labels in D_y. Condition 2 holds for the new action, and conditions 3 and 4 are unaffected.

Case 2) σ is the initiation of an identity operator c. By condition 4, the values in D_x and D_y to which this j^{th} occurrence of c corresponds are similar. Thus condition 3 holds, and conditions 2 and 4 are unaffected.

Case 3) σ is the termination of an operator c which is not an identity operator. By condition 2, the actions in D_x and D_y corresponding to the j^{th} occurrence of c are similar. Hence, the values added are similar, so the values labeled with the name of the output location of c in the dadep graphs of $x\sigma$ and $y\sigma$ are similar. This preserves condition 4, and conditions 2 and 3 are unaffected.

Case 4) σ is the termination of a decider c. By condition 2, the decisions in D_x and D_y corresponding to the j^{th} occurrence of c are similar, and here are given the same outcome label. This preserves condition 2 and conditions 3 and 4 are unaffected.

Case 5) σ is the termination of identity operator c. By condition 3, the values in D_x and D_y corresponding to the j^{th} occurrence of c are similar. Thus, the values in the dadep graphs of $x\sigma$ and $y\sigma$ bearing the name of the output location of c are similar, preserving condition 4. The other conditions are unaffected. □

Putting this lemma together with the observation which preceded it, we obtain the following lemma.

Lemma 7.2: If xz and yz are finite execution sequences and x and y are dadep indistinguishable, then xz and yz are consistent and equivalent.

7.9.1 Persistence, Commutativity and Conflict - Some Lemmas

Persistence, commutativity and conflict all deal with well-defined sequences of the form $x\sigma_1$, $x\sigma_2$, $x\sigma_1\sigma_2$, and $x\sigma_2\sigma_1$. We therefore derive some useful lemmas about such sequences.

The first lemma shows that if $\sigma_1 \neq \sigma_2$ and σ_1 and σ_2 are not the opposite terminations of some decider, then if $x\sigma_1$ and $x\sigma_2$ are well-defined sequences which are not in conflict, $x\sigma_1\sigma_2$ and $x\sigma_2\sigma_1$ are dadep indistinguishable. The proof, similar to that of Lemma 7.1, involves checking a number of cases against the definition of dadep graphs. We leave the details to interested readers and only outline the proof.

Lemma 7.3: Suppose $\sigma_1 \neq \sigma_2$ and σ_1 and σ_2 are not opposite terminations of the same decider. If $x\sigma_1$ and $x\sigma_2$ are well-defined sequences which are not in conflict, then $x\sigma_1\sigma_2$ and $x\sigma_2\sigma_1$ are dadep indistinguishable.

Proof: By exhausting all possible combinations of cases.

Case 1) $\sigma_1=\bar{a}$, $\sigma_2=\bar{b}$. Initiation events do not alter the labeling of values with memory locations. The actions are therefore attached to the same values independent of order. If either a or b or both are identity operators, no action is added, but the value corresponding to its occurrence is the one bearing the name of its input location. As mentioned, this is not altered by another initiation.

Case 2) $\sigma_1=\bar{a}$, $\sigma_2=\underline{b}$, b an operator. Since $x\underline{b}$ is well-defined, $\underline{b}$ must terminate some actor initiated in x. Notably, the initiation of a can have no effect on the operation to which this occurrence of $\underline{b}$ corresponds. After $\underline{b}$ has been added to the dadep graph, the memory labeling is changed. However, the change only involves the name of the output location of b. If this is an input location of a, $x\bar{a}$ and $x\underline{b}$ are in conflict. Otherwise, the change does not affect the action added by the occurrence of $\bar{a}$.

Case 3) $\sigma_1=\bar{a}$, $\sigma_2=d_T$ or $\sigma_2=d_F$. Since decider outcomes only add decision labels, there can be no interference between σ_1 and σ_2.

Case 4) $\sigma_1=\underline{a}$, $\sigma_2=\underline{b}$, a and b operators. If the output location of a is the same as the output location of b, $x\underline{a}$ and $x\underline{b}$ are in conflict. Otherwise, the order is immaterial.

Case 5) $\sigma_1=\underline{a}$, $\sigma_2=d_T$ or $\sigma_2=d_F$. As in case 3, there can be no problems.

Case 6) $\sigma_1=d_T$ or $\sigma_1=d_F$, $\sigma_2=e_T$ or $\sigma_2=e_F$. If d≠e, the decisions referred to must be distinct, and no problems arise. If d=e, then, since $\sigma_1\neq\sigma_2$, they must have opposite outcomes. But we also hypothesized that σ_1 and σ_2 were not opposite terminations of the same decider. Thus it cannot be that d=e. □

The next lemma shows that persistence has some global implications.

Lemma 7.4: In a persistent schema with the prefix property, if $x\sigma$ is the prefix of a control sequence and xy is a finite control sequence, then σ occurs in y, or, if σ is a decider termination, the opposite termination occurs in y.

Proof: Since xy is a finite control sequence, y is finite and we can write it as a sequence of events $y=a_1a_2\cdots a_n$. If a_1 is σ or the decider termination of which σ is the opposite termination, we are done. Otherwise, xa_1 and $x\sigma$ are prefixes which, by persistence, imply that $xa_1\sigma$ is also a prefix of a control sequence. Repeating the argument for

σ and a_2, σ and a_3, and so forth, we must either find an a_i equal to σ or the opposite termination thereof, or else $xa_1a_2\cdots a_n\sigma$ is the prefix of some control sequence. But this would violate the prefix property, for xy was a finite control sequence. Thus, such an a_i must be found. $\square$

Lemma 7.5: In a persistent schema with the prefix property, if $x\sigma$ is the prefix of a control sequence and is consistent with finite execution sequence xy, then σ occurs in y.

Proof: If σ is not a decider termination, this follows immediately from Lemma 7.4. So suppose σ is the j^{th} termination of some decider. By Lemma 7.4, either σ or the opposite termination must occur in y. If the opposite termination occurred first, then $x\sigma$ and xy would be inconsistent about the outcome of the j^{th} occurrence of the decider. Therefore, σ must be the next termination of the decider to occur. $\square$

The next lemma shows that in a persistent, commutative schema, an event which did not occur when it was first enabled can "slide back".

Lemma 7.6: In a persistent, commutative schema, if $x\sigma$ is the prefix of a control sequence and $xy\sigma z$ is a control sequence where neither σ nor, in the case that σ is a decider termination event, the opposite termination event, occurs in y, then $x\sigma yz$ is also a control sequence.

Proof: Suppose $y=a_1a_2\cdots a_n$, where each a_i is an event. By hypothesis, no a_i is σ or the opposite termination event if σ is a decider termination. By the same argument used in Lemma 7.4, $xa_1\sigma$, $xa_1a_2\sigma$, and so on through $xa_1a_2\cdots a_{n-1}\sigma$ are all prefixes of control sequences. Persistence then implies that $xa_1a_2\cdots a_{n-1}\sigma\ a_n$ is also the prefix of a control sequence. Since the schema is commutative,

$xa_1a_2\cdots a_{n-1}\sigma a_n z$ is also a control sequence. Repeating the argument n-1 times, we can conclude that $x\sigma yz$ is also a control sequence. □

An easy corollary follows if the schema is also conflict-free.

Lemma 7.7: In a persistent, commutative, and conflict-free schema, if $x\sigma$ is the prefix of a control sequence and $xy\sigma z$ is a control sequence where neither σ nor, in the case that σ is a decider termination event, the opposite termination event, occurs in y, then $x\sigma yz$ is a control sequence which is dadep indistinguishable from $xy\sigma z$.

Proof: From Lemma 7.3 we know that $xa_1\cdots a_{i-1}a_i\sigma a_{i+1}\cdots a_n z$ and $xa_1\cdots a_{i-1}\sigma a_i a_{i+1}\cdots a_n z$ are dadep indistinguishable for $1 \le i \le n$. One need only show that dadep indistinguishability is a transitive relation for the result to follow. By inspecting the definition of dadep indistinguishability, it is clear that it is an equivalence relation, hence transitive. □

7.10.1 Determinacy - Some Sufficient Conditions

We are now in a position to put some of the preceding lemmas together to prove the following determinacy result.

Theorem 7.8: A persistent, commutative schema with the prefix property is determinate if it is conflict-free.

Proof: Suppose x and y are I-computations. If both are infinite, they are equivalent and we are done. So suppose that x is finite and no longer than y. Suppose $x=a_1a_2\cdots a_n$ and $y=b_1b_2\cdots b_n\cdots$. We will find a sequence of I-computations, x_0, x_1, $\cdots$, x_n, such that x_i is dadep indistinguishable from x and the prefix of length i of x_i is $b_1b_2\cdots b_i$.

Letting $x_0=x$, we have the basis for our inductive argument. Suppose we have I-computation x_{i-1} satisfying the above conditions. We know that $x_{i-1}=b_1b_2\cdots b_{i-1}c_ic_{i+1}\cdots c_n$. Since $b_1b_2\cdots b_{i-1}b_i$ is the prefix of an execution sequence, y, which is consistent with x, and hence with x_{i-1}, Lemma 7.5 ensures that b_i occurs among the c_j's. Suppose c_k is the first such occurrence. By Lemma 7.7, $b_1b_2\cdots b_{i-1}b_ic_ic_{i+1}\cdots c_{k-1}c_{k+1}\cdots c_n$ is a control sequence which is dadep indistinguishable from x_{i-1}. By the transitivity of dadep indistinguishability, this sequence, which we shall call x_i, is also dadep indistinguishable from x.

Now consider $x_n=b_1b_2\cdots b_n$. If y is longer than x, $b_1b_2\cdots b_nb_{n+1}$ is in violation of the prefix property. By hypothesis, x was no longer than y. Hence, $x_n=y$. Since dadep indistinguishability implies equivalence, $x\equiv y$, which is precisely what we needed to show determinacy. $\square$

7.11.1 Determinacy - Retrospect

If the reader followed the last proof carefully, he will note that the result is even stronger than we stated. Persistence, commutativity, the prefix property, and conflict-freeness not only guarantee that I-computations are equivalent in our sense; they imply that all finite I-computations are dadep indistinguishable, mere "permutations" of one another.

Persistence, commutativity, conflict-freeness and the prefix property are sufficiently strong that even if our model is extended to allow subroutine-like use of schemata by other schemata, determinacy is preserved. We formalize how schemata can be so extended and outline the proof of determinacy in Appendix I.

8.1.1 Conflict and Non-determinacy - Introduction

We have seen that for persistent, commutative schemata with the prefix property, conflict-freeness is sufficient for determinacy. We will now turn to a class of schemata for which conflict-freeness is necessary for determinacy. We do not investigate such schemata simply "because they are there". Non-determinacy is generally something we wish to avoid, not to guarantee. By studying how a local problem such as a conflict can develop into a global problem of non-determinacy, however, we improve our understanding of how parallel systems behave. This understanding may help to circumvent non-determinacy in other classes of parallel systems.

8.2.1 Paths in Dadep Graphs - Similarity

Paths in a dadep graph show how actions depend upon the initial values and output values of operations. To gain some additional familiarity with the properties of paths in dadep graphs, we establish a simple result about paths and similarity.

Recall that arcs from values to actions are labeled with numbers. If an action is m-ary for some positive m, there are arcs into the action with labels 1 through m. Arcs from operations to values are unlabeled since there is precisely one such arc, if any.

A path π in a dadep graph is a sequence of arcs $\alpha_1, \alpha_2, \cdots, \alpha_n$ such that for $1<i\leq n$, the node from which arc α_i emanates is the node to which arc α_{i-1} leads. Let us agree that the i^{th} node on path π is the node from which arc α_i emanates, and the $n+1^{st}$ (and final) node on path π is the node upon which arc α_n terminates. We say that π is a path from the first node of π to the last node of π.

We say that path $\pi_1=\alpha_1, \alpha_2, \cdots, \alpha_m$ in dadep graph D_1 is *similar to* path $\pi_2=\beta_1, \beta_2, \cdots, \beta_n$ in dadep graph D_2 if all of the following conditions hold:

1) $m=n$. That is, the paths have the same length.

2) For $1\leq i\leq m$, either α_i and β_i are both unlabeled, or both have the same label. It follows that two similar paths both must begin on value nodes, or both must begin on action nodes.

3) For $1\leq i\leq m+1$, the i^{th} node on path π_1 is similar to the i^{th} node on path π_2.

We can relate similar paths to similar nodes by the following straightforward but useful lemma.

Lemma 8.1: Let $\pi_1=\alpha_1,\alpha_2,\cdots,\alpha_m$ be a path to node n_1 in dadep graph D_1. Node n_2 in dadep graph D_2 is similar to n_1 if and only if there is a path $\pi_2=\beta_1,\beta_2,\cdots,\beta_m$ to n_2 in D_2 which is similar to π_1.

Proof: Since the final nodes on similar paths are similar, one half of the lemma is trivial.

Suppose n_1 and n_2 are similar. If n_1 is a value node, α_m must be the unlabeled arc from the operation o_1 of which n_1 is the output value. Since n_2 is similar to n_1, it must be the output value of an operation o_2 similar to o_1. Let β_m be the unlabeled arc from o_2 to n_2.

Suppose n_1 is an action node. Then α_m must be the arc labeled i from v_1, the i^{th} input value of n_1. Since n_2 is similar to n_1, v_2, the i^{th} input value of n_2, must be similar to v_1. Let β_m be the arc labeled i from v_2 to n_2.

Working our way back in this manner, it is obvious that we can construct $\pi_2=\beta_1,\beta_2,\cdots, \beta_m$ similar to π_1. □

Two corollaries follow easily from Lemma 8.1.

Lemma 8.2: If n_1 and n_2 are similar nodes in dadep graphs D_1 and D_2, and if m_1 is a node on a path to n_1, then there is a node m_2 similar to m_1 on a path to n_2.

Lemma 8.3: If D_1 and D_2 are dadep graphs and n_1 is a node in D_1 which is not similar to any node in D_2, then no node on a path from n_1 is similar to any node in D_2.

8.3.1 Schemata - Some Additional Properties

Since the first determinacy result did not depend upon properties other than those mentioned, we made no attempt to further limit control sets. We now discuss some properties which lead to behavior more in line with the systems we mentioned in the introduction.

In a speed-independent system, one can make no assumptions about the length of time between the reading of an m-tuple of values and the completion of the processing thereon. If a control mechanism cannot block termination events, they must be anticipated at any time after the corresponding initiation event has occurred.

To make this precise, we say that a schema has the *immediate property* if, whenever x is the prefix of a control sequence and either o is an operator such that $x\underline{o}$ is well-sequenced or d is a decider such that xd_T and xd_F are well-sequenced, then $x\underline{o}$ or xd_T and xd_F are prefixes of control sequences.

Sequential schemata, as we have noted, have exactly one I-computation for each interpretation I. We generally expect parallel schemata to have more than one I-computation, but we would hardly expect them to have none at all. However, there are data flow graphs for which $\{dd_T\}$ satisfies all the conditions necessary to qualify as a control set. It is even persistent, commutative and conflict-free, and it has the prefix property. Any interpretation I which specified a false outcome for d's predicate would have no I-computations.

Even the fact that a control set contains an I-computation for each interpretation I is not sufficient to guarantee reasonable behavior.

For example, $\{\bar{d}\bar{a}\underline{a}d_T, \bar{d}\bar{b}\underline{b}d_F\}$ could be such a control set for some data flow graphs. If we view the sequence of events as occurring through time, however, we see that once $\bar{d}\bar{a}$ has occurred, somehow d cannot terminate with outcome false. This runs counter to our intuition about how control mechanisms and decisions interact.

To preclude some of this pathological behavior, we define a property called *completeness*. A schema is *complete* if, for every prefix x of a control sequence and for every interpretation I with which x is consistent there is an I-computation xy. In a complete schema, it is impossible to "run into a dead-end" under any interpretation.

It might seem that a schema with the immediate property would always be complete, for whenever xd_T is a prefix, so is xd_F. Unfortunately, this is not enough to guarantee completeness. For example, the control set $\bar{d}(d_T\bar{a}\underline{a}\bar{d})^*d_F$ has the immediate property, but, having no infinite sequences, there is no I-computation for the interpretation in which d's predicate is always true.

8.4.1 Productivity - Overview

We now turn to a very important notion, that of *productivity*. Productivity formalizes the concept of an event in a computation "accomplishing something useful." Initiating an actor in the course of a computation can be justified in a number of ways (as one can empirically verify by asking a programmer, "Why is that instruction in your program?") At the most trivial level, an actor is initiated to carry out a particular transformation or test. At a more satisfactory level, one can explain how the output of an operation will be used by subsequent actors, or what will happen if a decision comes out true. Best of all, the effect of an action on the input/output behavior of the schema could be pointed out. From a modular point of view, only the last justification is truly relevant.

We will define, in several cases, what we mean by the occurrence of an actor being *productive* in a sequence. The cases are rather involved, but the fundamental ideas are straightforward. For notational convenience, we define a *schema output value* to be any value in the dadep graph of a finite control sequence labeled with the name of some schema output location. In other words, a schema output value corresponds to the final contents of a schema output location.

An occurrence of an operator which is not an identity operator will be productive in a finite control sequence if the value it produces is a schema output value, or influences a schema output value, or influences a decision in the sequence. A decision will be productive if there are two non-equivalent control sequences between which the decision outcome arbitrates. An occurrence of an identity operator in a finite sequence is productive if the location to which the value is moved is a schema output location which is subsequently unchanged, or is the input location of a subsequent productive actor initiation. After formalizing these ideas, we will philosophize some about their meaning.

8.4.2 Productivity - Formal Definition

Suppose $w=x\bar{c}y\underline{c}z$ is a finite control sequence where the indicated occurrence of uninterpreted operator c is the j th such in w. Let D be the dadep graph of w, and let v be the value which is added to D when c terminates for the j th time. We say that the j th occurrence of c in w is *productive* if v is a schema output value of D, or lies on a path to a schema output value of D, or lies on a path to a decision in D.

Suppose $w_1=x\bar{d}z_1$ and $w_2=x\bar{d}z_2$ are control sequences where the indicated occurrences of decider d are the j th such in w_1 and w_2. We say that the j th occurrence of d is *productive* for w_1 and w_2 if w_1 and w_2 are not equivalent, there exist j th terminations of d in w_1 and w_2 having opposite outcomes, and all other decisions in w_1 and w_2 are consistent.

Finally, suppose $w=x\bar{c}y\underline{c}z$ where the indicated occurrence of identity operator c with output location ℓ is the j th such in w. Let D be the dadep graph of w. We say that the j th occurrence of c in w is *productive* if any or all of the following hold:

1) ℓ is a schema output location and no operator terminating in z has output location ℓ. (That is, c moves a value to a schema output location where it remains.)

2) z is of the form $z_1\bar{b}z_2\underline{b}z_3$ where the indicated occurrence of operator b is productive in w, ℓ is an input location of b, and no operator

terminating in z_1 has output location ℓ. (That is, the value moved to location ℓ is subsequently used from there by a productive operator.)

3) z is of the form $z_1 \bar{d} z_2$ where d is a decider which has ℓ as an input, and no operator terminating in z_1 has output location ℓ. (That is, the value is moved to a location from which it is used by a decision.)

8.4.3 Productivity - What Does it Mean?

The definitions of productivity just given probably look mysterious at best, but once understood, they are quite natural. For example, our definition of decider productivity follows from the common-sense principle: Don't ask questions if you don't care about the answers. This translates into our requirement that under some circumstances, the answer to just 1 question determines which of two non-equivalent computations take place. An example might help to clarify this point. Consider the following program and tabular representation of the output.

```
input ( X )
if p ( X )
then if q ( X ) then X ← f ( X ) else X ← g ( X )
else if q ( X ) then X ← f ( X ) else X ← g ( X )
output ( X )
```

p (X)	q (X)	OUTPUT
T	T	f(X)
T	F	g(X)
F	T	f(X)
F	F	g(X)

From the second and third lines of the table summarizing the behavior of the program, it can be seen that if p(X) is true, the output may be g(X), and if it is false, the output may be f(X). However, we do not consider the occurrence of p in the sequences

$\bar{p}p_T\bar{q}q_F\bar{f}\underline{f}$ and $\bar{p}p_F\bar{q}q_T\bar{g}\underline{g}$ productive because the sequences are not consistent about the decision made by q. There are no sequences for which p is productive which is probably in line with the reader's intuition about the usefulness of the test p(X) in the program. Our definition would find productive occurrences of q, however, which is again what one would expect.

The other forms of productivity are also grounded in common sense principles. They can be paraphrased, "Don't generate values you aren't going to use!" and "Don't move a value somewhere unless you need it there!"

8.5.1 Productivity of Schemata - Formal Definition

Knowing what it means for an occurrence of an actor to be productive in sequences, it is easy to extend the notion of productivity to schemata. In fact, there are several ways in which we could extend the definition, and we shall mention these briefly after our definition of *weakly productive schemata*.

A schema is said to be *weakly operator productive* if, for every prefix $x\bar{o}$ of a control sequence, there is a finite control sequence $x\bar{o}y$ in which the indicated occurrence of operator o is productive.

A schema is said to be *weakly decider productive* if, for every prefix $x\bar{d}$ of a control sequence, there are control sequences $x\bar{d}y_1$ $x\bar{d}y_2$ for which the indicated occurrence of decider d is productive.

A schema is said to be *weakly productive* if it is both weakly operator productive and weakly decider productive.

8.5.2 Productivity of Schemata - Discussion

In plain English, a schema is weakly productive if it does not initiate actions which cannot be useful. We have called this weak productivity because it is possible to impose much stronger conditions.

For example, we could define a schema to be *strongly operator productive* if, for every prefix $x\bar{o}$ of a control sequence, the occurrence of o is productive in <u>every</u> finite control sequence $x\bar{o}y$.

Analogously, we could define a schema to be *strongly decider productive* if, for every prefix $x\bar{d}$ of a control sequence, the occurrence of d is productive for <u>all</u> control sequences $x\bar{d}y$ and $x\bar{d}z$ in which the occurrences have opposite outcome. Or, we could require that for each control sequence $x\bar{d}y$, there exists a control sequence $x\bar{d}z$ in which the occurrence is productive. Or, we could require that there exist a control sequence $x\bar{d}y$ such that the occurrence of d is productive for all control sequences $x\bar{d}z$ in which the termination has opposite outcome.

Putting these variations on operator productivity and decider productivity together, we could obtain many classes of productive schemata. We will concentrate on weakly productive schemata because they are the largest of the classes and seem the most natural for encouraging parallelism: One can initiate actors if there is a possibility they will be useful.

Before leaving this discussion of productivity, we would like to make one observation. The Karp-Miller formalism for parallel schemata, because of the stronger form of equivalence and the way all actors alter memory, are inherently "very productive". We feel that this quality contributes substantially to the cleanness of their mathematical results. We pay for greater generality with more complicated proofs.

8.6.1 Schemata - Repetitions

The properties we have previously defined were ones that we found natural or desirable for a schema to possess. The next two properties are introduced less because they are natural than for the reason that they make the study of schemata more tractable. For example, the absence of these properties is intimately involved in Paterson's proof of the undecidability of the equivalence problem. The properties involve the notion of doing the same thing more than once in the course of a computation.

A sequence x is said to be *free* if no dadep graph of a prefix of x contains distinct, similar decisions. A sequence x is said to be *liberal* if no dadep graph of a prefix of x contains distinct, similar operations. A sequence is said to be *repetition-free* if it is both free and liberal.

A schema is said to be *free* [*liberal, repetition-free*] if all of its control sequences are free [liberal, repetition-free.] One nice feature of free schemata is that all control sequences are execution sequences. This is easily seen if one recalls that a control sequence which does not contain similar decisions with opposite outcomes is an execution sequence. Since free sequences do not contain similar decisions at all, they cannot have any with opposite outcomes. When dealing with free schemata, then, we will use the terms control sequence and execution sequence interchangeably.

8.7.1 Weakly Productive Schemata - A Preliminary Result

As an example of the intuitive appeal of weakly productive schemata, we prove a straightforward theorem. The proof will introduce a useful proof technique for productive schemata.

Suppose we define a *reduced schema* to be a schema for which every actor in the data flow graph is used in at least one control sequence. We will show that two free, determinate, reduced schemata cannot be equivalent if they employ substantially different functions or predicates.

Theorem 8.4: Let S_1 and S_2 be free, weakly productive, reduced, determinate schemata. S_1 is equivalent to S_2 only if PREDICATE NAMES$_1$ = PREDICATE NAMES$_2$ and, except possibly for the identity function name, FUNCTION NAMES$_1$ = FUNCTION NAMES$_2$.

Proof: Suppose uninterpreted operator o of S_1 has function name f and f $\notin$ FUNCTION NAMES$_2$. S_1 is reduced, so there is a control sequence in which o occurs, say $x\bar{o}y$. Since S_1 is weakly productive, there is a finite control sequence $x\bar{o}z$ in which the occurrence of o is productive. Because S_1 is free, $x\bar{o}z$ is an execution sequence. Suppose the operation o_1 which corresponds to the indicated occurrence of o lies on a path to a schema output value in the dadep graph of $x\bar{o}z$. Since o_1 has function name f, no operation in a dadep graph of a control sequence from S_2 could be similar to o_1. By Lemma 8.3, it follows that no dadep graph of a control sequence of S_2 could contain a schema output value similar to that to which o_1 leads. Hence, no sequence from S_2 could be equivalent to $x\bar{o}z$.

Since o_1 cannot lie on a path to a schema output value if the schemata are to be equivalent, it must be that o_1 lies on a path to a decision d_1 in the dadep graph of $x\bar{o}z$. We can write $x\bar{o}z$ as $x\bar{o}z_1\bar{d}z_2$, where the action corresponding to the indicated occurrence of d is d_1. By the weak productivity and freeness of S_1, there exist execution sequences $x\bar{o}z_1\bar{d}v_1$ and $x\bar{o}z_1\bar{d}v_2$ for which the occurrence of d is productive. By definition, the only inconsistent decisions in these sequences are the indicated occurrences of d. Therefore, there is an interpretation I_T in which the indicated occurrence of d is true and which is consistent with all other decisions in <u>both</u> sequences. Let I_F be identical to I_T except at the outcome of d. One of the sequences is an I_T-computation, and the other is an I_F-computation.

If S_2 has no I_T-computations, then S_1 and S_2 are obviously not equivalent schemata. So suppose w is an I_T-computation of S_2. We claim that w is also an I_F computation, for, if not, it must contain a decision similar to d_1 having true outcome. But d_1 lies on a path from o_1 and no dadep graph of a sequence from S_2 can contain an operation similar to o_1. By Lemma 8.3, neither can it contain a decision similar to d_1.

Thus, w is both an I_T-computation and an I_F-computation. Since $x\bar{o}z_1\bar{d}v_1$ and $x\bar{o}z_1\bar{d}v_2$ were not equivalent, w is equivalent to at most one of them. It is therefore consistent with but not equivalent to the other, so S_1 and S_2 cannot be determinate and equivalent.

Exactly the same argument applies if there is a predicate name in PREDICATE NAMES_1 - PREDICATE NAMES_2. □

8.8.1 Productivity - Undecidability

The reader may have foreseen that productivity is "too good to be decidable." In fact, the proof that productivity is not generally decidable follows easily from the unsolvability of the equivalence problem.

Suppose we take two m-input n-output Paterson schemata whose equivalence we would like to determine. As in the proof of the undecidability of the determinacy question, we can assume ACTORS_1 and ACTORS_2 are disjoint. Since the schemata have the same number of input locations and output locations, we can form, as before, a composite data flow graph by identifying the input locations of the schemata and the output locations of the schemata.

Let p be a unary predicate name which is not in $\text{PREDICATE NAMES}_1 \cup \text{PREDICATE NAMES}_2$, and let f be a 0-ary function name which is not in $\text{FUNCTION NAMES}_1 \cup \text{FUNCTION NAMES}_2$. Add to the composite data flow graph a memory location which is the output location of a new operator o with function name f, and the input location of a new decider d with predicate name p. Form the composite CONTROL by prefacing each sequence in CONTROL_1 with $\bar{o}\underline{o}\bar{d}d_T$ and each sequence in CONTROL_2 with $\bar{o}\underline{o}\bar{d}d_F$. We claim d is productive if and only if S_1 and S_2 are not equivalent. This is true because d is productive if and only if there exist sequences $\bar{o}\underline{o}\bar{d}d_T z_1$ and $\bar{o}\underline{o}\bar{d}d_F z_2$ which are not equivalent but are consistent except for the outcome of d. This obviously means that z_1 and z_2 are consistent, non-equivalent sequences, that is, S_1 and S_2 are not equivalent.

8.9.1 Determinacy and Conflict - Some Additional Results

For the remainder of this chapter, we will investigate a class of schemata for which conflict-freeness is necessary for determinacy. We are not suggesting that the properties which cause this problem with determinacy are desirable. On the contrary, we are much more concerned with ensuring determinacy than with precluding it. By understanding how conflict leads to non-determinacy, however, we are in a better position to avoid non-determinacy in schemata.

We begin by establishing a number of lemmas concerning repetition-free schemata in which there are no identity operators. Let us fix some such schema, S. For notational convenience, we will use DADEP(x) to denote the dadep graph which is determined by the data flow graph of S and a well-defined sequence x. If $x\sigma_1 y\sigma_2$ is a sequence which is well-defined for S, then we say that event σ_1 *influences* event σ_2 if there is a path in DADEP($x\sigma_1 y\sigma_2$) from the action corresponding to the indicated occurrence of σ_1 to the action corresponding to the indicated occurrence of σ_2.

The lemmas which follow share a great deal of notation. We shall introduce the common notations here to avoid the necessity of repeating the definitions with each lemma.

Let $x\bar{a}$ and $x\underline{o}$ be prefixes of control sequences of S such that $x\bar{a}$ and $x\underline{o}$ are in conflict. Let $w_1 = x\bar{a}\underline{o}y\bar{b}$ and $w_2 = x\underline{o}\bar{a}y\bar{b}$ be prefixes of control sequences of S. Let a_1 be the action in DADEP(w_1) corresponding to the indicated occurrence of a, and let a_2 be the action in DADEP(w_2) which corresponds to the indicated occurrence of a in w_2. Similarly, let b_1 and b_2 be the actions in DADEP(w_1) and DADEP(w_2), respectively, which correspond to the indicated occurrences of b.

We shall prove the following lemmas.

Lemma 8.5: a_1 is not similar to a_2.

Lemma 8.6: $\bar{b}$ is influenced by $\bar{a}$ in w_1 iff $\bar{b}$ is influenced by $\bar{a}$ in w_2.

Lemma 8.7: b_1 is similar to b_2 iff $\bar{a}$ does not influence $\bar{b}$ in w_1.

Lemma 8.8: If S is commutative, there is no prefix $x\underline{o}\bar{a}z$ of a control sequence of S such that DADEP($x\underline{o}\bar{a}z$) contains an action similar to a_1, and no prefix $x\bar{a}\underline{o}z$ of a control sequence such that DADEP($x\bar{a}\underline{o}z$) contains an action similar to a_2.

Lemma 8.9: If $x\bar{a}\underline{o}z$ and $x\underline{o}\bar{a}z$ are control sequences of S, they are consistent.

Proof of Lemma 8.5 (a_1 is not similar to a_2.):

Since $x\bar{a}$ and $x\underline{o}$ are in conflict, m, the output location of operator o, is an input location, say the k^{th}, of actor a. Because $x\bar{a}$ is well-defined, m must be the label on some value v in DADEP(x). Since $x\underline{o}$ is well-defined, there must be an operation o' in DADEP(x) to which this termination of o corresponds. In DADEP($x\underline{o}$), a value v' is added as the output value of o', and label m is removed from v to become the label on v'. If v and v' were similar, v would have to be the output value of an operation similar to o'. But this operation and o' would be distinct, similar operations, contrary to the hypothesis that S is repetition-free. Therefore, the k^{th} input value of a_1 in DADEP($x\bar{a}$) is not similar to the k^{th} input value of a_2 in DADEP($x\underline{o}\bar{a}$). It follows that a_1 and a_2 cannot be similar. $\square$

Proof of Lemma 8.6 ($\bar{b}$ is influenced by $\bar{a}$ in w_1 iff $\bar{b}$ is influenced by $\bar{a}$ in w_2.):

Note that except for the order of occurrence of $\bar{a}$ and $\underline{o}$, w_1 and w_2 are identical. If the indicated occurrence of a in w_1 is not terminated, then neither is the occurrence of a in w_2, and $\bar{b}$ is uninfluenced in both sequences. If $y=c_1c_2\cdots c_n$ and c_i is the termination of a in w_1, then c_i is also the termination of a in w_2. It is easy to see that the events influenced by c_i in w_1 are the same as those influenced by c_i in w_2, from which the lemma follows. □

Proof of Lemma 8.7 (b_1 is similar to b_2 iff $\bar{a}$ does not influence $\bar{b}$ in w_1.):

We split the proof into halves. In both cases, we assume the contrary and derive a contradiction.

Suppose b_1 is similar to b_2 and there is a path from a_1 to b_1. We can assume that $y=c_1c_2\cdots c_n$ is the shortest sequence after which such similar, influenced initiations occur. The nodes on the path in DADEP(w_1) from a_1 to b_1 all correspond to events occurring in y. Let c_j be the last operation termination to which a node on the path corresponds. (There must be at least one operator termination in y since a must terminate if there is a path from a_1.) It follows that b_1 takes as an input value the output value of the operation which c_j terminates. It is also true that b_2 takes as an input value the output value of the operation which c_j terminates in w_2. Since b_1 and b_2 are similar, these operations must be similar. By Lemma 8.5, we know that a_1 and a_2 are not similar, so c_j cannot be the termination of the occurrences of a in w_1 and w_2.

It follows that c_j must be the termination of an operation initiated in y, say by event c_i. Both c_j and c_i are influenced by $\bar{a}$ in w_1, and we have concluded that the operation corresponding to c_i in DADEP(w_1) is similar to the operation corresponding to c_i in DADEP(w_2). This contradicts our hypothesis that y was the shortest sequence after which such initiations could occur. We can conclude that if b_1 and b_2 are similar, then $\bar{b}$ is not influenced by $\bar{a}$ in w_1.

To obtain the other half of the lemma, assume that b_1 and b_2 are not similar, that there is no path from a_1 to b_1 in DADEP(w_1), and that $y=c_1c_2\cdots c_n$ is the shortest sequence after which these conditions can arise. If b were a 0-ary operator, b_1 and b_2 would be similar, so we can assume that b has at least one input location. The function names or predicate names on actions b_1 and b_2 are, of course, the same, so it must be that the i^{th} input value of b_1 is dissimilar to the i^{th} input value of b_2 for some i. Let m be the i^{th} input location of b. If no operator terminating in w_1 has output location m, then m must be a schema input location, say the j^{th}, and the i^{th} input value of b_1 is the j^{th} initial value. Sequence w_2 contains exactly the same termination events as does w_1, so the i^{th} input value of b_2 would also be the j^{th} initial value, contrary to the assumption that the i^{th} input values of b_1 and b_2 are dissimilar. So some operator terminating in w_1 has output location m. Suppose the k^{th} occurrence of operator c in w_1 is the last operator with output location m to terminate in w_1. It follows that the output value of the operation in DADEP(w_1) corresponding to the k^{th} occurrence of c is the i^{th} input value of b_1. It also follows that the output value of the operation in DADEP(w_2) corresponding to the kth occurrence of c is the ith input value of b_2. Since the ith input values are dissimilar, so must be the operations. We now show that no matter where the kth initiations of c take place in w_1 and w_2, some hypothesis is contradicted.

Case 1): The kth initiation of c in w_1 occurs in prefix x. The kth initiation of c in w_2 must also occur in x, and the corresponding operations would be similar, contrary to hypothesis.

Case 2): The kth initiation of c in w_1 is the indicated occurrence of $\bar{a}$. This contradicts the assumption that there is no path from a_1 to b_1.

Case 3): The kth initiation of c in w_1 (and w_2) is event c_h in y. It follows that y is not the shortest sequence after which the hypothesized conditions can arise.

We conclude that if b_1 and b_2 are dissimilar, then $\bar{b}$ is influenced by $\bar{a}$ in w_1. $\square$

Proof of Lemma 8.8 (If S is commutative, there is no prefix $x\underline{o}\bar{a}z$ of a control sequence of S such that DADEP($x\underline{o}\bar{a}z$) contains an action similar to a_1, and no prefix $x\bar{a}\underline{o}z$ of a control sequence such that DADEP($x\bar{a}\underline{o}z$) contains an action similar to a_2.):

We prove only the first half of the lemma. The second half follows by similar arguments. We assume that there is a prefix $x\underline{o}\bar{a}z$ of a control sequence of S such that DADEP($x\underline{o}\bar{a}z$) contains an action a_3 similar to a_1. We show that no matter where the initiation of the action occurs, we can derive a contradiction.

Case 1): The initiation event to which a_3 corresponds occurs in prefix x. Then $x\bar{a}$ is the prefix of a control sequence containing a repetition, contrary to hypothesis.

Case 2): The initiation event to which a_3 corresponds is the indicated occurrence of a in $x\underline{o}\bar{a}z$. By Lemma 8.5, a_1 is not similar to a_3.

Case 3): The initiation event to which a_3 corresponds takes place after prefix $x\underline{o}\bar{a}$. We can assume that a_3 corresponds to the initiation of b in $x\underline{o}\bar{a}y\bar{b}$. Let us hereafter refer to action a_3 as action b_2 so we can make use of the notation we have used in the previous lemmas. By assumption, b_2 is similar to a_1. Since S is commutative, $w_1=x\bar{a}\underline{o}y\bar{b}$ is also the prefix of a control sequence.

Case 3a): $\bar{b}$ is not influenced by $\bar{a}$ in w_1. Lemma 8.7 implies that b_1 is similar to b_2. Since b_2 is similar to a_1, b_1 must be similar to a_1, contrary to the hypothesis tbat S is repetition-free.

Case 3b): $\bar{b}$ is influenced by $\bar{a}$ in w_1. There is therefore a path from a_1 to b_1 in DADEP(w_1). By Lemma 8.6, there must also be a path from a_2 to b_2 in DADEP(w_2). Since b_2 is similar to a_1, it follows from Lemma 8.2 that there is an action similar to a_2 on a path to a_1 in DADEP(w_1). Actions on a path to a_1, however, would have to correspond to events occurring in prefix x. Prefix $x\underline{o}\bar{a}$ would therefore contain a repetition of action a_2, contrary to hypothesis.

It follows that no action in DADEP($x\underline{o}\bar{a}z$) can be similar to a_1. □

Proof of Lemma 8.9 (If $x\bar{a}\underline{o}z$ and $x\underline{o}\bar{a}z$ are control sequences of S, they are consistent.):

We assume the contrary and derive a contradiction. Suppose the j^{th} occurrence of decider b in $x\bar{a}\underline{o}z$ is inconsistent with some decision in $x\underline{o}\bar{a}z$.

Case 1): The j^{th} initiation of b in $x\bar{a}\underline{o}z$ takes place in prefix x. The decision corresponding to the j^{th} initiation of b in $x\underline{o}\bar{a}z$ would then be similar and would have the same outcome. No other decision in $x\underline{o}\bar{a}z$ could be similar unless there were a repetition, so no inconsistency is possible.

Case 2): The j^{th} initiation of b in $x\bar{a}\underline{o}z$ is the indicated occurrence of a. By Lemma 8.8, there is no decision in $x\underline{o}\bar{a}z$ which is similar. Therefore, there is no inconsistent decision in $x\underline{o}\bar{a}z$.

Case 3): The j^{th} initiation of b in $x\bar{a}\underline{o}z$ takes place after prefix $x\bar{a}\underline{o}$. Let us write $x\bar{a}\underline{o}z$ as $x\bar{a}\underline{o}y\bar{b}y'$ to indicate the j^{th} occurrence of b and to match our previous notations.

Case 3a): $\bar{b}$ is not influenced by $\bar{a}$ in $x\bar{a}\underline{o}y\bar{b}y'$. By Lemma 8.7, the decisions corresponding to the occurrences of b in $x\bar{a}\underline{o}y\bar{b}y'$ and $x\underline{o}\bar{a}y\bar{b}y'$ are similar and they have the same outcome. As in the first case, no inconsistency can arise unless there is a repetition.

Case 3b): $\bar{b}$ is influenced by $\bar{a}$ in $x\bar{a}\underline{o}y\bar{b}y'$. This means there is a path from a_1 to b_1 in DADEP($x\bar{a}\underline{o}y\bar{b}$). By Lemma 8.8, no prefix of $x\underline{o}\bar{a}z$ has a dadep graph containing an action similar to a_1. By Lemma 8.3, there can be no decision in $x\underline{o}\bar{a}z$ similar to b_1, hence no decision inconsistent with it.

We conclude that $x\bar{a}\underline{o}z$ and $x\underline{o}\bar{a}z$ are consistent. □

With these lemmas to build upon, we can now show how one kind of conflict leads to non-determinacy.

Theorem 8.10: Let $x\bar{a}$ and $x\underline{o}$ be prefixes of a complete, persistent, commutative, repetition-free, weakly productive schema with no identity operators. If $x\bar{a}$ and $x\underline{o}$ are in conflict, then the schema is not determinate.

Proof: Suppose $x\bar{a}$ and $x\underline{o}$ are in conflict. We will show that consistent, non-equivalent execution sequences exist, demonstrating that the schema is not determinate. By the persistence of the schema, $x\underline{o}\bar{a}$ is a prefix of a control sequence.

Case 1): Actor a is an operator. Since the schema is weakly productive, there is a control sequence $x\underline{o}\bar{a}z$ in which the occurrence of a is productive.

Case 1a): $x\underline{o}\bar{a}z$ is finite and there is a path in DADEP($x\underline{o}\bar{a}z$) from the action a_2 corresponding to the indicated occurrence of a to some schema output value, say that value labeled by schema output location m. Since the schema is commutative, $x\bar{a}\underline{o}z$ is a control sequence and, by Lemma 8.9, it is consistent with $x\underline{o}\bar{a}z$. By Lemma 8.8, there is no operation in DADEP($x\bar{a}\underline{o}z$) similar to a_2. By Lemma 8.3, no value in DADEP($x\bar{a}\underline{o}z$) is similar to the value in DADEP($x\underline{o}\bar{a}z$) with label m. Thus, $x\bar{a}\underline{o}z$ and $x\underline{o}\bar{a}z$ cannot be equivalent. Since the schema is repetition-free, these are consistent, non-equivalent execution sequences.

Case 1b): $x\underline{o}\bar{a}z$ can be written $x\underline{o}\bar{a}y\bar{b}y'$ where the indicated initiation of decider b is influenced by $\bar{a}$. Appealing the the weak productivity of the schema, there are control sequences $v_1 = x\underline{o}\bar{a}y\bar{b}z_1$ and $v_2 = x\underline{o}\bar{a}y\bar{b}z_2$ for which the occurrence of b is productive. Let I_T be any interpretation in which the decision corresponding to the indicated occurrence of b is true, and which is consistent with all other decisions in v_1 and v_2. Let I_F be identical to I_T except at the outcome of the decision corresponding to b which I_F specifies as false. Since the schema is complete and $x\bar{a}\underline{o}$ is consistent with I_T, there is an I_T-computation $x\bar{a}\underline{o}u$. We claim $x\bar{a}\underline{o}u$ is also an I_F-computation, for, by Lemma 8.8, the dadep graph of any prefix of $x\bar{a}\underline{o}u$ can contain no operations similar to a_2. Since a_2 is on a path to the only decision about whose outcome I_T and I_F differ, it follows from Lemma 8.3 that there is no similar decision in $x\bar{a}\underline{o}u$. Since v_1 and v_2 are not equivalent, $x\bar{a}\underline{o}u$ is equivalent to at most one of them, but is consistent with both. Therefore there is a non-determinacy.

Case 2): Actor a is a decider. This is virtually identical to case 1b, so we omit the proof. □

Prefixes $x\underline{a}$ and $x\underline{b}$ of control sequences were also said to be in conflict if a and b were distinct operators with the same output location. We will show that this kind of conflict can also cause non-determinate behavior. Many of the arguments we use are virtually identical to those in the case of conflict of the form $x\bar{a}$ and $x\underline{o}$. We will therefore go into less detail in the following proofs.

Lemma 8.11: Suppose $x\underline{a}\underline{b}$ and $x\underline{b}\underline{a}$ are prefixes of control sequences in a commutative, repetition-free, weakly productive, complete schema with no identities. Suppose b is productive in control sequence $x\underline{a}\underline{b}z$. If $x\underline{a}$ and $x\underline{b}$ are in conflict, then the schema is not determinate.

Proof: Let ℓ be the memory location which is the output location of both a and b. If value v_1 in DADEP($x\underline{a}\underline{b}$) with label ℓ is similar to the value v_2 in DADEP($x\underline{b}\underline{a}$) with label ℓ, there is clearly a repetition in $x\underline{a}\underline{b}$. For the same reason, there can be no input value of an action in the dadep graph of a sequence with prefix $x\underline{b}\underline{a}$ similar to v_1 since a's termination removes the label from a value similar to v_1. Analogously, no action after prefix $x\underline{a}\underline{b}$ can have an input value similar to v_2.

As in Lemma 8.7, it can be seen that the actions corresponding to the indicated occurrences of c in $x\underline{a}\underline{b}y\bar{c}$ and $x\underline{b}\underline{a}y\bar{c}$ are similar if and only if there is no path from v_1 to the action in DADEP($x\underline{a}\underline{b}y\bar{c}$). From this, as in Lemma 8.9, it follows that $x\underline{a}\underline{b}z$ and $x\underline{b}\underline{a}z$ are consistent. Since b was, by hypothesis, productive in control sequence $x\underline{a}\underline{b}z$, the same arguments used in Theorem 8.10 apply and a non-determinacy can be shown to exist. $\square$

The last lemma shows that conflicts of the type $x\underline{a}$ and $x\underline{b}$ cause trouble when b is productive. We now show that conflicts of this type imply the existence of conflicts involving productive operations.

Lemma 8.12: Let $x\underline{a}$ and $x\underline{b}$ be conflicting prefixes of a persistent, commutative, weakly productive schema with the prefix property and the immediate property. Either there exists a conflict of the form $v\underline{o}$, $v\bar{c}$, or there exist sequences $v\underline{c}\underline{o}z$ and $v\underline{o}\underline{c}z$ in which c is productive and $v\underline{c}$ and $v\underline{o}$ are in conflict.

Proof: We can assume, without loss of generality, that a initiates before b in prefix x, say $x= x_1\bar{a}x_2\bar{b}x_3$. Since the schema is weakly productive, there is a finite sequence $x_1\bar{a}x_2\bar{b}z$ in which the occurrence of b is productive. If a does not terminate in z, then, by the immediate property, $x_1\bar{a}x_2\bar{b}z\underline{a}$ is the prefix of a control sequence, in violation of the prefix property. So we can assume both b and a terminate in z.

Suppose a terminates before b so $z=z_1\underline{a}z_2\underline{b}z_3$. Let $z_2=c_1c_2\cdots c_n$. Using the immediate property, persistence and commutativity, we can "slide $\underline{b}$ to the left" as in Lemma 7.6. That is, $x_1\bar{a}x_2\bar{b}z_1\underline{a}c_1\cdots\underline{b}c_i\cdots c_nz_3$ is also a control sequence for $1\le i\le n$. From Lemma 7.7 , we know that the sequences formed by moving $\underline{b}$ are dadep indistinguishable unless $\underline{b}$ reaches a c_i such that $x_1\bar{a}x_2\bar{b}z_1\underline{a}c_1c_2\cdots c_{i-1}c_i$ and $x_1\bar{a}x_2\bar{b}z_1\underline{a}c_1c_2\cdots c_{i-1}\underline{b}$ are in conflict. This will certainly happen when $\underline{b}$ reaches $\underline{a}$. But when it first happens, we have a conflict of the type hypothesized in the lemma.

So suppose b terminates before a in z so that $z=z_1\underline{b}z_2\underline{a}z_3$. Since a overwrites the location ℓ written by $\underline{b}$, b could not be productive in the sequence unless some actor initiating in z_2 has ℓ as an input location.

Let us write z_2 as $z_{21}\bar{c}z_{22}$ where actor c has input location ℓ. As above, $\underline{a}$ can migrate left so that $x_1\bar{a}x_2\bar{b}z_1\underline{b}z_{21}\bar{c}$ and $x_1\bar{a}x_2\bar{b}z_1\underline{b}z_{21}\underline{a}$ are prefixes in conflict as hypothesized. $\square$

Putting the last two lemmas together with Theorem 7.8 and Theorem 8.10, we obtain the following major result.

Theorem 8.13: A persistent, commutative, complete, repetition-free, weakly productive schema without identity operators which has the prefix property and the immediate property is determinate if and only if it is conflict-free.

8.10.1 Theorem 8.13 - A Post-Mortem

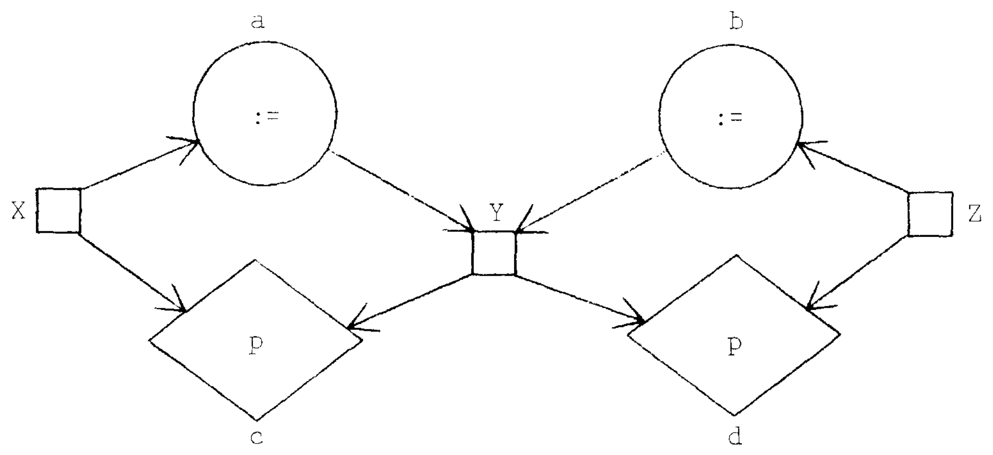
Theorem 8.13 contains a not inconsiderable number of conditions, and the reader may question whether any schemata of interest can satisfy all of them. To allay such doubts, we will review the nature of these conditions.

Completeness, as we mentioned, is such a natural property that one would probably not bother to point out that a schema model possesses it. Practically any control mechanism which anticipates both outcomes of decisions will automatically ensure completeness. Similarly, the prefix property is generally satisfied by definition: finite control sequences are defined to be those for which no further activity is possible. Thus, completeness and the prefix property do not seriously restrict the class of systems we can model.

Persistence and commutativity are natural properties for systems with multiple, independent loci of control. These properties reflect practical limitations on the ability of such systems to co-ordinate activity. Commutativity might be viewed as accepting that if two events can occur arbitrarily close together in time, it makes little sense to behave differently on the basis of order of occurrence. Similarly, persistence can be considered to model the inability to cancel activity on arbitrarily short notice. Of course, non-commutative and non-persistent control mechanisms can be defined, and they may exhibit certain advantages. We simply suspect that practical systems will tend to be persistent and commutative.

Weak productivity, even though it is not generally a decidable property, appears to be a desirable one in practical terms. The increase of concurrency in computation should not arise solely from the introduction of unproductive activity. We should point out that parallelism, persistence, and productivity are occasionally at odds with one another. A simple example of this is the case where we wish to perform some action if either or both of two decisions are true. Parallelism encourages us to allow the decisions to proceed concurrently. If one decision terminates with true outcome before the other has even initiated, however, productivity would incline us to cancel the other decision, whereas persistence forbids cancellation. Clearly, some trade-offs between parallelism and productivity must be made.

Whether Theorem 8.13 remains valid if identity operators are allowed is not known. Using an extended notion of repetition-freeness which prohibits use of an identity operator to restore the contents of a memory location to a value similar to one it held earlier in the computation (X:=X being a simple violation), we were unable to disprove the theorem. However, certain of the lemmas we used to prove the theorem certainly cease to apply. For example, sequences $x=\bar{a}\bar{b}\underline{a}\underline{b}\bar{c}c_T\bar{d}d_F$ and $y=\bar{a}\bar{b}\underline{b}\underline{a}\bar{c}c_T\bar{d}d_F$ for the data flow graph of Figure 8.1 are not consistent since the decision corresponding to c_T in x is similar to the decision corresponding to d_F in y. Thus, the analogue to Lemma 8.9 fails. Although we suspect the theorem remains true nevertheless, a different approach will be needed to prove it.



SCHEMA INPUTS = {X,Z}

Figure 8.1 Data flow graph with identities for which conflict can cause inconsistency

Repetition-freeness and the immediate property are less pervasive than some of the other properties mentioned, but they are by no means unlikely to be observed. The immediate property, like persistence and commutativity, will tend to be present when there is considerable independence among loci of control. Sequential schemata, fork/join formalisms, and Dennis-Fosseen schemata always have the immediate property. Petri net and finite-state controls can easily be checked for the immediate property.

Repetition-freeness in sequential schemata is decidable[11], and it appears that *t-counter transition systems* as described in an appendix of Slutz's thesis[16] can be used to determine repetition-freeness in fork/join controls, Dennis-Fosseen schemata, Petri net models in which the number of tokens in the net is bounded, and finite state models in which the number of "unterminated initiations" of actors is bounded. We know of no decision procedure for repetition-freeness of unbounded Petri net or state-machine controlled schemata. As shown by Paterson[11], the freeness of an illiberal schema is not decidable.

Conflicts in sequential schemata and Dennis-Fosseen schemata cannot arise. Conflict-freeness for fork/join, Petri net, and finite state controlled schemata is decidable.

Thus, the conditions in Theorem 8.13 are not so unusual or restrictive as one might first suspect. The reader can verify that the various implementations of our sample program in Chapter 3 are all persistent, commutative, complete, repetition-free, weakly productive, and have the prefix property and the immediate property. Fortunately, they are also conflict-free, hence determinate.

9.1.1 Summary

Our state of knowledge about parallel processing is, at the moment, rather primitive. In a way, this is not surprising. Individuals seem to be incapable of consciously working on more than one thing at a time, so we have less "introspective intuition" about parallel computation than we do for sequential processing. Only recently have computer hardware and software made parallel processing possible, and parallelism at the instruction level is still not practical.

This dissertation has attempted to do some groundwork in parallel processing. Recognizing the lack of practical experience with parallel control mechanisms, we have defined a very flexible model for parallel computation. Many proposed control mechanisms can be cast in terms of our model, allowing meaningful comparison of apparently diverse schemes.

Our definition of equivalence reflected a modular, input/output orientation. A schema is determinate if all the computations which might be observed for a given interpretation are equivalent. Although we noted that determinacy is not generally decidable, we showed that persistent, commutative schemata with the prefix property are determinate if they are conflict-free. Many control mechanisms which have been or might be proposed are persistent, commutative, and have the prefix property.

Since not all actors alter schema output locations, we introduced the notion of productivity to formalize the way in which the occurrence of an actor can ultimately affect input/output behavior. It was seen that the rôles of operators and deciders are quite different, supporting our decision to treat them as distinct actors as opposed to the hybrid operations of Karp and Miller[7].

We extended the concept of productivity to schemata, requiring, in the weakest form, that no actor initiate unless it might be useful. Such weakly productive schemata satisfied some intuitive properties such as precluding equivalence of schemata which use different functions or predicates.

Although weak productivity is not decidable in general, it can usually be observed in "real programs". Programmers may not have our formal notion of productivity in mind when they create programs, but in the process of program development, instructions are added "for a reason". Generally, at least for good programmers, this satisfies our requirements for productivity.

In the last chapter, we showed how conflicts can lead to non-determinacy. Although conditions such as repetition-freeness are less natural than others like completeness, the conditions for Theorem 8.13 are not unlikely to be met. This argues for models such as Dennis-Fosseen schemata in which conflicts simply cannot occur. At the very least, it seems we should favor models where detection of conflicts is uncomplicated.

9.2.1 Future Directions

We would like to see the issue of identity operators in Theorem 8.13 put to rest. We suspect that "non-repetitious" identity operators will not alter the validity of the theorem. It may also be possible to relax repetition-freeness which seems to be an unnecessarily restrictive condition.

We share with other schematologists a desire to see a better model for dealing with structured data such as arrays and lists. Considering the complications introduced by the identity operator alone, we expect that allowing "just enough interpretation" to handle structured data without getting into problems of undecidability may be difficult indeed.

It would be worthwhile to look into treating mathematical properties such as associativity and commutativity of actors. It seems that commutative arguments can be modeled in a data flow graph by using the same index on arcs from the inputs which commute. (Of course, *similarity* would have to be redefined, but the changes appear straightforward.) Associativity and distributivity might also be handled, but appear to lead us toward the treacherous territory of rewrite rules.

Identifying classes of schemata for which equivalence is decidable is an important goal. We would hope that such classes could be characterized "syntactically" to make the results relatively independent of control mechanism. Schemata which are strongly productive in the sense of Section 8.5.2 seem to be a promising start for such a class.

Those who have read this far in this dissertation will probably see other directions for extending our work. We hope that the model and results presented here may be of some use in continuing the study of parallel systems.

BIBLIOGRAPHY

1. Conway, M. E., "A Multiprocessor System Design", *Proceedings of the FJCC*, 1963, pp. 139-146.

2. Dennis, J. B., *Computation Structures*. Notes for subject 6.232, Department of Electrical Engineering, M.I.T., Cambridge, Massachusetts, 1970.

3. Dennis, J. B., J. B. Fosseen and J. P. Linderman, "Data Flow Schemas", paper prepared for Symposium on Theoretical Programming, Novosibirsk, USSR, 1972.

4. Dennis, J. B. and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations", *Communications of the ACM, Vol. 9, No. 3*, March, 1966, pp. 143-155.

5. Fosseen, J. B., *Representation of Algorithms by Maximally Parallel Schemata*, S.M. Thesis, M.I.T., Cambridge, Massachusetts, June 1972.

6. Holt, A. W. and F. Commoner, *Events and Conditions*, (in three parts), Applied Data Research, New York, 1970.

7. Karp, R. M. and R. E. Miller, "Parallel Program Schemata", *Journal of Computer and System Sciences, Vol. 3, No. 2*, May, 1969, pp. 147-195.

8. Keller, R. M., "On Maximally Parallel Schemata", *IEEE Conference Record. Eleventh Annual Symposium on Switching and Automata Theory*, 1970, pp. 32-50.

9. Luckham, D. C., D. M. R. Park and M. S. Paterson, "On Formalized Computer Programs", *Journal of Computer and System Sciences, Vol. 4, No. 3*, June, 1970, pp. 220-249.

10. Minsky, M. L., *Computation: Finite and Infinite Machines*, Prentice-Hall, Englewood Cliffs, New Jersey, 1967.

11. Paterson, M. S., *Equivalence Problems in a Model of Computation*, Ph.D. Thesis, University of Cambridge, August, 1967.

12. Paterson, M. S. and C. E. Hewitt, "Comparitive Schematology", *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*, ACM, New York, 1970, pp. 119-127

13. Rodriguez, J. E., *A Graph Model for Parallel Computation*, Report MAC-TR-64, Project MAC, M.I.T., Cambridge, Massachusetts, 1968.

14. Rogers, H., *Theory of Recursive Functions and Effective Computability*, McGraw-Hill, 1967.

15. Rutledge, J. D., "On Ianov's Program Schemata", *Journal of the ACM, Vol. 11, No. 1*, January, 1964, pp. 1-9.

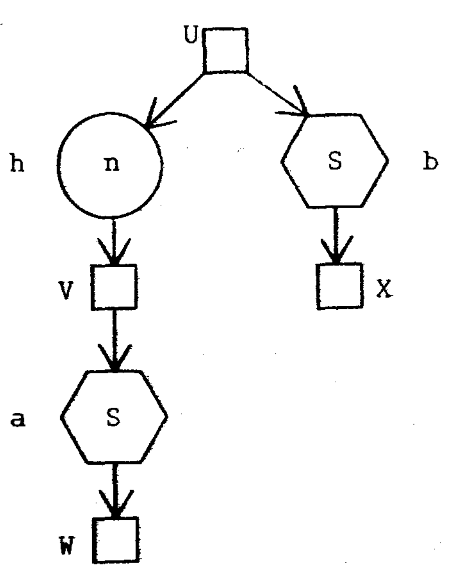
16. Slutz, D. R., *The Flow Graph Schemata Model of Parallel Computation*, Report MAC-TR-53, Project MAC, M.I.T., Cambridge, Massachusetts, 1968.

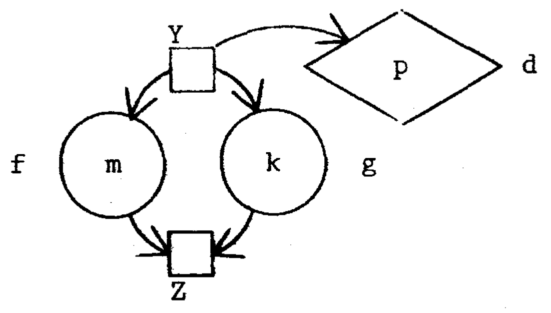
Appendix I Schema Systems

An m-input, n-output schema looks, from our modular viewpoint, very much like an operator. Operators always have one output, whereas n can be greater than one, but this is not an important distinction. A more significant difference is that operators have well defined outputs for all interpretations while schemata may have infinite computations for some interpretations and therefore may not produce outputs.

There is a natural way of allowing a schema to use other schemata as "macro-operators". Data flow graphs can be extended to allow m-input n-output actors with associated schema names. We call this new form of actor an *application*. When an application initiates in a control sequence, we can consider that a new *activation* of the named schema is created. The schema input locations for the activation are initialized from the input locations of the application, and the activation can then proceed concurrently with the sequence which spawned it. Of course, the corresponding application termination event in the original sequence cannot occur until the activation has completed. If the activation does terminate and the application termination event is enabled, it can occur, copying the schema output values of the activation into the output locations of the application. In a practical system, the storage used by the activation could then be deallocated.

In Figure I.1, we show a simple example of the use of a schema as a macro-operator. Applications appear as hexagons to distinguish them from the other actors.





$\text{CONTROL}_S=\{\bar{d}d_T\bar{f}\underline{f},\bar{d}d_F\bar{g}\underline{g}\}$

$\text{SCHEMA INPUTS}_S=\{Y\}$

$\text{SCHEMA OUTPUTS}_S=\{Z\}$

$\text{CONTROL}=\{\bar{b}\bar{h}\underline{h}\bar{a}\underline{b}\underline{a},\cdots\}$

SCHEMA INPUTS={U}

SCHEMA OUTPUTS={W,X}

Main sequence	$\bar{b}$	$\bar{h}$	$\underline{h}$				$\bar{a}$				$\underline{b}$			$\underline{a}$
First activation of S				$\bar{d}$	d_T	$\bar{f}$			$\underline{f}$					
Second activation of S								$\bar{d}$		d_F		$\bar{g}$	$\underline{g}$	

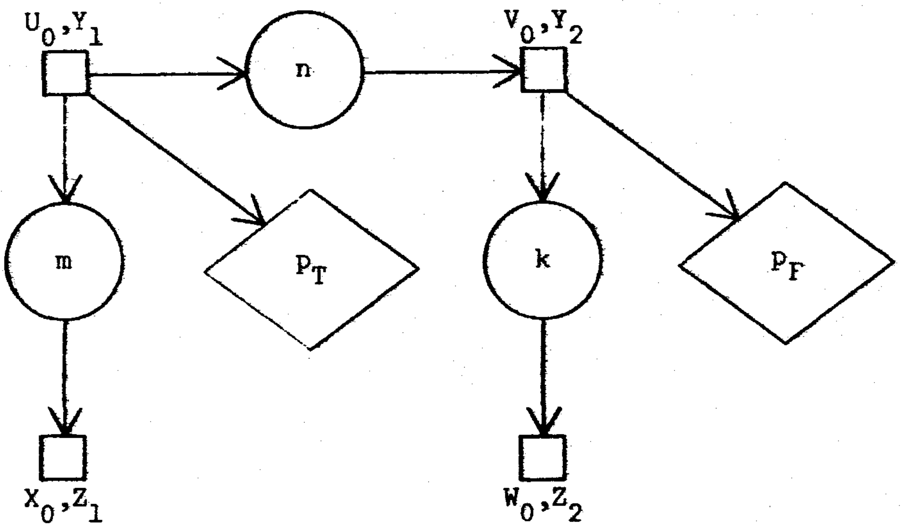


Figure I.1 Use of a schema as a macro-operator

The example also shows possible sequences of events from the main schema and the two activations of schema S. Note that $\underline{b}$, the termination of application b, may have been *enabled* in the sequence from the main schema immediately after b initiated, but could not *occur* until the first activation of S completed. An extended dadep graph shows the effect of the sequences. Labels from the activations are subscripted to avoid confusion. The reader may be able to reconstruct the dadep graph step by step from the sequences shown and the description of the way application initiations and terminations behave.

Once one understands the use of schema applications, it is easy to see that there is, in principle, no difficulty in allowing schemata to apply themselves, or to allow recursive applications of arbitrary complexity. In this appendix, we will formalize the notions introduced here and give a semi-formal proof that commutativity, persistence and the prefix property are sufficient to guarantee determinacy for such *schema systems*.

I.2.1 Definition of Terms

An *extended schema* S is a schema in which ACTORS may include *applications* with associated names from a set of SCHEMA NAMES. If S_i is an m_i-input, n_i-output schema name and a is an application with schema name S_i, then the data flow graph of S contains arcs labeled 1 through m_i from (not necessarily distinct) locations in MEMORY to a, and arcs labeled 1 through n_i from a to distinct locations in MEMORY. The alphabet of events, Σ, of S contains an initiation event $\bar{a}$ and a termination event $\underline{a}$ for each application a.

Terms such as *well-sequenced, well-defined, persistent, commutative, control sequence,* and *conflict-free* are analogous to the definitions for ordinary schemata when applications are treated like operators. In particular, $x\underline{a}$ and $x\bar{b}$ are *in conflict* if <u>any</u> output location of a is also an input location of b, and similarly for $x\underline{a}$ and $x\underline{b}$.

A *schema system* S is a finite collection $S_1, S_2, \cdots, S_k$ of extended schemata such that $\{S_1, S_2, \cdots, S_k\} \supseteq$ SCHEMA NAMES_i for $1 \le i \le k$. S_1 is the distinguished *main schema*. We can assume without loss of generality that the alphabets of events of the extended schemata are pairwise disjoint. Let $\Sigma = \Sigma_1 \cup \Sigma_2 \cup \cdots \cup \Sigma_k$ be the *alphabet of events* of S.

The control sets of the extended schemata in a schema system determine what we will be defining as the *system sequences* of the schema system. As in the example, system sequences will comprise events from the individual schemata. Since there may be several concurrent activations of individual schemata, there must be additional structure imposed on the sequence of events to indicate which events correspond to which activations. In the example, we used spacing to show this. We add this structure formally by defining a mapping $\psi : N \rightarrow N \cup \{0\}$ which specifies, for each event in the system sequence, the index of an activation to which it belongs. That is, $\psi(i)=j$ means the i^{th} event in the system sequence is from the j^{th} activation. We reserve index 0 to identify the initial activation of the main schema. The other indexes are important only insofar as they uniquely identify activations.

If x is a sequence of events and ψ is a mapping as described above, we will write $E_{\psi}(x,i)$ to denote the sequence formed by eliminating from x all those events which are <u>not</u> mapped into i by ψ. When ψ is clear from context, we may also write this as $E(x,i)$. Since ψ is defined to identify activations of schemata, we expect $E(x,i)$ to be the control sequence of some extended schema, or at least a prefix, possibly empty, of such a control sequence.

We need still another formalism to recapture the information present in our example. Although ψ determines the activation in which an event occurs, we need to relate application initiations and terminations to activations. We formalize this information by another partial mapping $\zeta : N \rightarrow N$ where, if the i^{th} event in sequence x is an application initiation or an application termination, $\zeta(i)$ is the index of the activation it initiates or terminates.

For our sample sequence, we can tabularly present some of these formalisms as follows. It should be evident that x, ψ, and ζ supply all the information we need to reconstruct our example.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
x	$\bar{b}$	$\bar{h}$	$\underline{h}$	$\bar{d}$	d_T	$\bar{f}$	$\bar{a}$	$\bar{d}$	$\underline{f}$	d_F	$\underline{b}$	$\bar{g}$	$\underline{g}$	$\underline{a}$
ψ	0	0	0	1	1	1	0	2	1	2	0	2	2	0
ζ	1	-	-	-	-	-	2	-	-	-	1	-	-	2
$E(x,0)$	$\bar{b}$	$\bar{h}$	$\underline{h}$				$\bar{a}$				$\underline{b}$			$\underline{a}$
$E(x,1)$				$\bar{d}$	d_T	$\bar{f}$			$\underline{f}$					
$E(x,2)$								$\bar{d}$		d_F		$\bar{g}$	$\underline{g}$	

Figure I.2 Formalisms for describing a system sequence

We formalize the notion of a system sequence as follows. (Throughout this appendix, we will make no distinction between a function and the restriction of that function to a subset of its domain. No serious confusion should arise.)

Suppose $x=a_1a_2a_3\cdots$ is a sequence in $\hat{\Sigma}$, and ψ and ζ are mappings as introduced above. We say that $x;\psi;\zeta$ is a *system sequence* for S if and only if the following conditions hold:

1) $\psi(1)=0$. $E(x,0)$ is a control sequence of the main schema, S_1, or is the prefix of a control sequence of S_1. (Thus, the first thing to happen must come from the main schema.)

2) Suppose $\psi(j)\neq 0$. Then there is in x an initiation a_i of an application a with schema name S_k, say the m^{th} initiation of a in $E(x,\psi(i))$, such that $i<j$, $\zeta(i)=\psi(j)$, and $E(x,\psi(j))$ is a control sequence of schema S_k or a prefix thereof. Furthermore, if there exists an $n\neq i$ such that $\zeta(n)=\psi(j)$, then a_n must be the m^{th} termination of a in $E(x,\psi(i))$, $E(x,\psi(j))$ must be a finite control sequence of S_k, and, if t is the largest integer for which $\psi(t)=\psi(j)$, then $t<n$. (As complicated as this may appear, it simply states that activations are started by application initiations, behave like the schema applied, and must themselves terminate before the application termination can occur.)

We can extend the familiar notion of prefixes to schema systems as follows. Let $x;\psi;\zeta$ and $z;\psi';\zeta'$ be system sequences of S. Then $x;\psi;\zeta$ is said to be a *prefix* of $z;\psi';\zeta'$ if

1) x is a prefix of z. Say $x=a_1a_2\cdots a_n$.
2) For all $i,j\leq n$, $\psi(i)=\psi(j)$ if and only if $\psi'(i)=\psi'(j)$.
3) For all $i,j\leq n$, if a_i is an application initiaition or an application termination, then $\zeta(i)=\psi(j)$ if and only if $\zeta'(i)=\psi'(j)$.

If conditions 1 through 3 are met and $x\neq z$, then we say that $x;\psi;\zeta$ is a *proper prefix* of $z;\psi';\zeta'$.

A system sequence $x;\psi;\zeta$ is a *control sequence* of S if x is infinite, or if x is finite and there is no system sequence $z;\psi';\zeta'$ of which $x;\psi;\zeta$ is a proper prefix. In other words, a system sequence is a control sequence if it runs forever or if it comes to a halt and no event can occur in any activation. This last restriction resembles the prefix property extended to schema systems.

If $x;\psi;\zeta$ is a system sequence of S and if x is finite, we can outline the definition of the *dadep graph* of $x;\psi;\zeta$ by the following induction.

The dadep graph of $\lambda;\psi;\zeta$ consists of m_1 initial values where m_1 is the number of schema input locations of the main schema, S_1. If X is the i^{th} schema input location of S_1, the i^{th} initial value has label X_0.

In general, all memory location labels in our extended dadep graphs will be subscripted with the index of the activation which uses them. This will ensure that activations operate independently of one another. Only application initiations and terminations will be able to influence labels from another activation and, by definition, there is no activity within the influenced activation when the application events occur.

Suppose $x=a_1a_2\cdots a_{i-1}a_i\cdots$ and we have already defined the dadep graph of $a_1a_2\cdots a_{i-1};\psi;\zeta$.

If a_i is the initiation of an application a with schema name S_k, the dadep of $a_1\cdots a_i;\psi;\zeta$ is produced from the dadep graph of $a_1\cdots a_{i-1};\psi;\zeta$ by adding labels to values as follows. Suppose X is the j^{th} input location of application a, and Y is the j^{th} schema input location of schema S_k. Then label $Y_{\zeta(i)}$ is added to the value bearing label $X_{\psi(i)}$. (In a more formal definition, we would use an induction hypothesis to guarantee that $X_{\psi(i)}$ exists as a label on precisely one value in the dadep graph of $a_1a_2\cdots a_{i-1};\psi;\zeta$. We leave it to the reader to fill in such missing details using the definition of ordinary dadep graphs as a model.)

If a_i is the termination of an application a with schema name S_k, then we know that $E(a_1a_2\cdots a_{i-1},\zeta(i))$ is a finite control sequence of S_k. By the definition of control sequences of individual schemata, all output locations of schema S_k are assigned values by such an activation. We form the dadep graph of $a_1\cdots a_i;\psi;\zeta$ from the dadep graph of $a_1\cdots a_{i-1};\psi;\zeta$ by moving and/or adding labels as follows. Suppose X is the j^{th} output location of application a, and Y is the j^{th} schema output location of schema S_k. Then $X_{\psi(i)}$ is removed from any value on which it may have been a label, and $X_{\psi(i)}$ is added as a label to the value with label $Y_{\zeta(i)}$.

The cases where a_i is an operator or decider initiation or termination are directly analogous to the cases for ordinary dadep graphs. The only difference is that they deal with labels subscripted with $\psi(i)$. We therefore omit the details for these cases.

An *interpretation* for a schema system $S_1,S_2,\cdots,S_k$ simply defines a single DOMAIN, functions for each function name in any schema S_i, predicates for each predicate name in any schema S_i, and a set of initial elements for the main schema S_1.

Assignment of elements to values, consistency, similarity and other terms carry over to extended dadep graphs in a straightforward way. In particular, two control sequences $x;\psi;\zeta$ and $z;\psi';\zeta'$ are *equivalent* if both x and z are infinite, or if both are finite, and, for each output location X of the main schema, the value labeled X_0 in the dadep graph of $x;\psi;\zeta$ is similar to the value labeled X_0 in the dadep graph of $z;\psi';\zeta'$.

We say that a schema system $S = S_1,S_2,\cdots,S_k$ is *persistent, commutative, conflict-free* and has the *prefix property* if each extended schemata S_i is persistent, commutative, conflict-free and has the prefix property. We can show that such a system is determinate by the same techniques we used in the earlier proof for unextended schemata. Rather than repeat almost verbatim the lemmas proved earlier, we will only outline parts of the proof. It will hopefully be possible for interested readers to use the formalisms we have introduced to fill in the details with as much rigor as they wish.

Suppose $x;\psi;\zeta$ and $z;\psi';\zeta'$ are I-computations of schema system S. If both x and z are infinite, the computations are equivalent, and we are done. We can therefore suppose that $x=a_1a_2\cdots a_n$ and is no longer than $z=b_1b_2\cdots b_n\cdots$. As before, we will find I-computations $x_0;\psi_0;\zeta_0 \cdots x_n;\psi_n;\zeta_n$ equivalent to $x;\psi;\zeta$ such that $x_i=b_1b_2\cdots b_ic_{i+1}\cdots c_n$, and $b_1b_2\cdots b_i;\psi_i;\zeta_i$ is a prefix of $z;\psi';\zeta'$.

Letting $x_0;\psi_0;\zeta_0$ be $x;\psi;\zeta$, the basis for our inductive argument is established. Suppose $x_{i-1};\psi_{i-1};\zeta_{i-1}$ has been found. This means that $z = b_1b_2\cdots b_{i-1}b_i\cdots$ and $x_{i-1} = b_1b_2\cdots b_{i-1}c_ic_{i+1}\cdots c_n$, with $\psi_{i-1}(1) = 0 = \psi'(1)$. Additionally, for $j,k<i$, $\psi_{i-1}(j) = \psi_{i-1}(k)$ iff $\psi'(j) = \psi'(k)$ and $\zeta_{i-1}(j) = \psi_{i-1}(k)$ iff $\zeta'(j) = \psi'(k)$.

We consider the activation with index $\psi'(i)$ in $z;\psi';\zeta'$ in which event b_i occurs. Let $v = E_{\psi'}(\ b_1b_2\cdots b_{i-1}\ ,\ \psi'(i)\ )$. We know that v is the prefix of some control sequence and that b_i can occur next in this activation. We will proceed as follows. We will show that there is an activation with index j_0 in $b_1b_2\cdots b_{i-1};\psi_{i-1};\zeta_{i-1}$ in which prefix v has occurred and for which b_i can occur next. We will show that if b_i does not occur later in this activation, $x_{i-1};\psi_{i-1};\zeta_{i-1}$ is the proper prefix of a system sequence in which b_i <u>does</u> occur in the activation. Since this would contradict our hypothesis that $x_{i-1};\psi_{i-1};\zeta_{i-1}$ is a control sequence, we will conclude that b_i eventually occurs in the activation with index j_0. We will then argue that b_i's occurrence can take place immediately after prefix $b_1b_2\cdots b_{i-1};\psi_{i-1};\zeta_{i-1}$ without changing the dadep graph of $x_{i-1};\psi_{i-1};\zeta_{i-1}$. This will provide us with I-computation $x_i;\psi_i;\zeta_i$, completing the induction.

We begin by finding the activation of $x_{i-1};\psi_{i-1};\zeta_{i-1}$ with index j_0 which is "analogous" to the activation of $z;\psi';\zeta'$ with index $\psi'(i)$. If $\psi'(i) = 0$, let $j_0 = 0$. $E_{\psi_{i-1}}(b_1b_2\cdots b_{i-1} , 0) = v$ because $\psi'(1) = 0$ and $\psi_{i-1}(1) = 0$ and $\psi'(j) = \psi'(1)$ iff $\psi_{i-1}(j) = \psi_{i-1}(1)$. If $\psi'(i) \neq 0$, then there is some $k < i$ such that $\zeta'(k) = \psi'(i)$. Again, by our inductive hypotheses, $\psi'(j) = \zeta'(k)$ iff $\psi_{i-1}(j) = \zeta_{i-1}(k)$. In this case, let j_0 be $\zeta_{i-1}(k)$. In either case, we see that $E_{\psi_{i-1}}(b_1b_2\cdots b_{i-1} , j_0) = v$. Let $vw = E_{\psi_{i-1}}(x_{i-1} , j_0)$. That is, w is the rest of the sequence from the activation of which v was the prefix.

We now argue that event b_i must occur in w, for suppose not. Since $vb_i = E_{\psi'}(b_1b_2\cdots b_i , \psi'(i))$, b_i is enabled after prefix v. By the persistence of the individual schemata, vwb_i would then be the prefix of a control sequence. Suppose b_i is the termination of an application. (This is the most complicated case. Other types of events can be treated in a similar manner.) Let k_0 be the index in z of the corresponding application initiation event. That is, k_0 is the unique index for which $\zeta'(k_0) = \psi'(i)$. Since $k_0 < i$, we know by our induction hypotheses that for all $j < i$, $\zeta'(k_0) = \psi'(j)$ iff $\zeta_{i-1}(k_0) = \psi_{i-1}(j)$. Since b_i was able to occur when it did in $z;\psi';\zeta'$, $E_{\psi'}(b_1b_2\cdots b_{i-1} , \zeta'(k_0))$ must have been a finite control sequence. But $E_{\psi_{i-1}}(b_1b_2\cdots b_{i-1} , \zeta_{i-1}(k_0))$

is the same finite control sequence, so b_i is free to occur in the activation with index j_0 after prefix $b_1b_2\cdots b_{i-1};\psi_{i-1};\zeta_{i-1}$. Extend ψ_{i-1} and ζ_{i-1} so that $\psi_{i-1}(n+1) = j_0$ and $\zeta_{i-1}(n+1) = \zeta_{i-1}(k_0)$, and consider $x_{i-1}b_i;\psi_{i-1};\zeta_{i-1}$. If this is a system sequence, $x_{i-1};\psi_{i-1};\zeta_{i-1}$ is clearly a proper prefix of it, violating the hypothesis that $x_{i-1};\psi_{i-1};\zeta_{i-1}$ is a control sequence. We have shown that vwb_i is $E_{\psi_{i-1}}(\ x_{i-1}b_i\ ,\ j_0\ )$, the prefix of a control sequence. We know that the activation indexed by $\zeta_{i-1}(n+1)$ has already terminated, so this cannot prevent b_i from occurring. The only thing which could prevent $x_{i-1}b_i;\psi_{i-1};\zeta_{i-1}$ from satisfying the definition of a system sequence is that some application termination event in x_{i-1} terminate the activation indexed by j_0 before b_i can occur. But this would imply that some proper prefix of vwb_i is a control sequence, contradicting the hypothesized prefix property of the individual schemata. Thus, we can conclude that event b_i occurs in w.

We have succeeded in showing that $w = w_1c_kw_2$ where $c_k = b_i$. The persistence and commutativity of the individual schemata allow us to conclude that $vc_kw_1w_2$ is a control sequence or prefix thereof.

Let $x_i = b_1 b_2 \cdots b_{i-1} c_k c_i c_{i+1} \cdots c_{k-1} c_{k+1} \cdots c_n$

$$\begin{aligned} \text{Let } \psi_i(j) &= \psi_{i-1}(j) && \text{for } j < i \text{ and } j > k \\ &= \psi_{i-1}(k) && \text{for } j = i \\ &= \psi_{i-1}(j-1) && \text{for } i < j \le k \end{aligned}$$

$$\begin{aligned} \text{Let } \zeta_i(j) &= \zeta_{i-1}(j) && \text{for } j < i \text{ and } j > k \\ &= \zeta_{i-1}(k) && \text{for } j = i \\ &= \zeta_{i-1}(j-1) && \text{for } i < j \le k \end{aligned}$$

In short, $x_i;\psi_i;\zeta_i$ is the result of promoting the occurrence of event c_k in activation j_0 of $x_{i-1};\psi_{i-1};\zeta_{i-1}$. By arguments similar to those used in showing that b_i occurred in w, $x_i;\psi_i;\zeta_i$ is a control sequence. The sequences observed in all activations except that activation indexed by j_0 remain the same as in $x_{i-1};\psi_{i-1};\zeta_{i-1}$. $E_{\psi_{i-1}}(\ x_{i-1}\ ,\ j_0\ ) = vw_1 c_k w_2$ while $E_{\psi_i}(\ x_i\ ,\ j_0\ ) = vc_k w_1 w_2$. Since the individual schemata are conflict-free, it is not too difficult to see that $vw_1c_kw_2$ and $vc_kw_1w_2$ do exactly the same thing to the initial values of activation j_0. (A formal proof would follow the lines of the dadep indistinguishability results for unextended schemata.) Therefore, the dadep graph of $x_i;\psi_i;\zeta_i$ is virtually identical to the dadep graph of $x_{i-1};\psi_{i-1};\zeta_{i-1}$. This means $x_i;\psi_i;\zeta_i$ is an I-computation equivalent to $x_{i-1};\psi_{i-1};\zeta_{i-1}$, completing the induction.

We can now conclude the proof as in the case of ordinary schemata. That is, $x_n = b_1 b_2 \cdots b_n$. If $x_n;\psi_n;\zeta_n$ is a proper prefix of $z;\psi';\zeta'$, we contradict the assumption that $x_n;\psi_n;\zeta_n$ is a control sequence. If $x_n = z$, and $x_n;\psi_n;\zeta_n$ is a prefix of $z;\psi';\zeta'$, it follows that the sequences are equivalent. (They need not be identical. That is, $\psi_n(i)$ need not equal $\psi'(i)$. However, the restrictions that $\psi_n(i) = \psi_n(j)$ iff $\psi'(i) = \psi'(j)$ and $\zeta_n(i) = \psi_n(j)$ iff $\zeta'(i) = \psi'(j)$ make any differences unimportant. That is, the sequences differ only on the "names" of the activations.)

Since $x;\psi;\zeta$ is equivalent to $x_n;\psi_n;\zeta_n$ which is, in turn, equivalent to $z;\psi';\zeta'$, the original I-computations are equivalent, and the schema is determinate. This concludes the outline of the proof.

GLOSSARY AND INDEX

BIOGRAPHICAL NOTE

John Parent Linderman was born in Eau Claire, Wisconsin on September 18, 1946. He graduated from Regis High School, Eau Claire, Wisconsin, in 1964. Mr. Linderman received an S.B. in mathematics at MIT in 1968. He entered the computer science area of the Electrical Engineering Department at MIT where he received an S.M. in 1970 and a Ph.D. in 1973.

As an undergraduate, Mr. Linderman was a Sloan National Scholar. He held an NSF Fellowship during the first three years of graduate study, and was a research assistant at Project MAC for the last two years.

Mr. Linderman was employed as a programmer-analyst by the MIT Office of Administrative Information Systems during the summers of 1965 through 1971, and on a part-time basis during the school years. He became a full-time research assistant at Project MAC in the fall of 1971 and remained there until June, 1973.

Mr. Linderman expects to join the staff at Bell Labs Raritan River Center in New Jersey.

BIBLIOGRAPHIC DATA SHEET	1. Report No. NSF-OCA-GJ34671-TR-111	2.	3. Recipient's Accession No.

4. Title and Subtitle Productivity in Parallel Computation Schemata	5. Report Date : Issued December 1973 6.
7. Author(s) John P. Linderman	8. Performing Organization Rept. No. MAC-TR-111
9. Performing Organization Name and Address PROJECT MAC; MASSACHUSETTS INSTITUTE OF TECHNOLOGY: 545 Technology Square, Cambridge, Massachusetts 02139	10. Project/Task/Work Unit No. 11. Contract/Grant No. GJ-432 GJ-34671
12. Sponsoring Organization Name and Address Associate Program Director Office of Computing Activities National Science Foundation Washington, D. C. 20550	13. Type of Report & Period Covered: Interim Scientific Report 14.

15. Supplementary Notes

Ph.D. Thesis, M.I.T., Department of Electrical Engineering

16. Abstracts

A general model for parallel computation is developed in three parts. One part, the data flow graph, describes how actors which transform and test values are connected to the locations in a finite memory. Another part, an interpretation, supplies information about the contents of memory and the detailed nature of the transformations and tests.

The third part specifies how initiations and terminations of the actors are allowed to occur. We define this in a general way, using a set of sequences of initiation and termination events to model control. This allows us to prove results which apply to a broad class of control mechanisms.

Our major results are analogous to a theorem of Karp and Miller. Their theorem defines a class of schemata for which conflict-freeness is necessary and sufficient for determinacy. We use a weaker notion of determinacy which depends only upon the final contents of a subset of the memory locations. To establish necessity, we introduce the property of productivity which expresses whether individual transformations and tests contribute to the final results of a computation.

17. Key Words and Document Analysis. 17a. Descriptors

Program Schemata

Parallel Computation

Theory of Programs

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement Unlimited Distribution Write Project MAC Publications	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 153
	20. Security Class (This Page UNCLASSIFIED	22. Price