COMPUTER DESIGN FOR ASYNCHRONOUSLY REPRODUCIBLE MULTIPROCESSING

by

EARL CORNELIUS VAN HORN, JR.

S.B., Massachusetts Institute of Technology
1961

S.M., Massachusetts Institute of Technology
1963

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
September, 1966

Signature of Author Signature redacted
Department of Electrical Engineering, August 22, 1966

Certified by Signature redacted
Thesis Supervisor

Accepted by Signature redacted
Chairman, Departmental Committee on Graduate Students

COMPUTER DESIGN FOR ASYNCHRONOUSLY REPRODUCIBLE MULTIPROCESSING

by

EARL CORNELIUS VAN HORN, JR.

Submitted to the Department of Electrical Engineering on August 22, 1966, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ABSTRACT

A concept is presented for designing either a computing system, or a programming language system, so that the following problem is avoided: during a multiprocess computation in which several processes communicate, and in which the relative timing of the performance of the processes is arbitrary, the output produced by the computation might not be a function of only the initial computation state, i.e., of only the inputs and initial program of the computation. The design concept for avoiding this problem is explained by defining an apparently new class of abstract machines called machines for coordinated multiprocessing, or MCM's. Processes are coordinated in an MCM by means of a count matrix, which may be modified by actions of processes, and which determines the processes enabled to proceed at any instant. Remarks are made to suggest that a computing facility which behaves like an MCM can be both constructed and programmed at reasonable cost. It is proved that every MCM has the properties of output functionality and output assuredness. Output functionality means that each symbol in every output stream is a function only of the initial computation state. Output assuredness means that for each output stream the maximum number of symbols produced in the stream, or the fact that the number of such symbols has no upper bound, is a function only of the initial computation state.

Thesis Supervisor: Jack B. Dennis

Title: Associate Professor of Electrical Engineering

Preface

The Thesis discusses three broad topics: (1) a problem encountered in some contemporary computing systems, (2) a concept for designing a computing system in order to solve this problem, and (3) a proof that the concept presented does indeed solve the problem.

The problem concerns multiprocessing. Multiprocessing occurs whenever a computing system is programmed by a single user in a manner that allows more than one sequence of actions to be performed for him simultaneously. Suppose information is transmitted among such sequences of actions, and suppose the relative timing of the performance of these sequences is not under user control. The problem to which the Thesis is addressed is that in these circumstances the output produced for a user by the performance of such sequences might not be a function of only the inputs and initial program that are specified by the user. A more precise statement of the problem and a discussion of the problem's origin and significance are the subjects of Chapter I.

A concept for designing a computing system in order to avoid the problem just mentioned has been discovered. An attempt has been made to capture the essence of this design concept by describing, in Chapter II, a class of abstract machines called machines for coordinated multiprocessing, or MCM's for short. A preview of the structure of an MCM is given near the beginning of Chapter II.

Chapter III discusses the feasibility of constructing and programming a computing system whose internal structure can be placed into correspondence with the structure of an appropriately chosen MCM.

In Chapters IV and V it is proved that the problem mentioned above does not arise in an MCM. Conclusions and suggestions for future research are presented in Chapter VI.

The Thesis might be of interest to three groups: (1) those concerned with designing the hardware and supervisory software of computing systems, (2) those concerned with designing programming language systems, i.e., programming languages and compilers for these languages, and (3) those concerned with the theory of automata. The primary orientation of the Thesis is toward the design of the hardware and supervisory software of a computing system. The Thesis is also relevant, however, to the design of programming language systems, in two respects. First, the programming of a computing system constructed along the lines to be discussed, although shown in the Thesis to be feasible, presents an interesting challenge both to programmers, and to the designers of programming language systems. Second, by employing the design concept to be discussed, it is possible to develop for any computer a programming language system having the property that if a user interacts with the computer only by means of the language system, the user will never encounter the problem mentioned above. Finally, the Thesis is relevant to automata theory, because it appears that the class of machines called MCM's has not been studied before.

Although the Thesis as a whole is oriented toward the area of computer engineering practice, the topics and methods of Chapters II, IV, and V are characteristic of the area of automata theory. The techniques of automata theory have been employed in the Thesis for three reasons. First, the essence of the design concept that has been discovered is captured more effectively in a description of an abstract machine than in a description of a hypothetical computing system; in a description of the latter kind the essence of the design concept would tend to be obscured by irrelevant detail. Second, by expressing the design concept in terms of the structure of an abstract machine, the nature of an apparently new phenomenon is made readily available for study and comparison with other concepts in the theory of automata. Third, the mathematical language and techniques used in automata theory allow the presentation of highly rigorous proofs of facts that, although intuitively satisfying, have turned out to be slippery to verify by means of logic.

For the reasons just mentioned, the Thesis may be said to stand both in the area of computer engineering practice, and in the area of automata theory. As with any interdisciplinary work, the technical jargon of one area might not be readily understood by those working in the other area. An effort has been made, therefore, to explain technical terms not common to both fields, and to avoid locutions likely to be misleading.

The doctoral research program which led to the present Thesis began with an inquiry into the possibility of analyzing quantitatively the problems of storage allocation in computing systems. This inquiry led to a search for ways of characterizing the structural aspects of computing systems, particularly multiprogrammed computing systems. The present research topic, in turn, resulted from an effort to describe the properties of those events which constitute the execution of a single program within a multiprogrammed system.

I should like to acknowledge gratefully the support of M.I.T.'s Project MAC[*], the activities of which stimulated my search toward the present research topic. The presentation of ideas in the Thesis has been substantially improved as a result of comments and suggestions by Professors R. McNaughton, R. Y. Kain, and F. J. Corbató. I am grateful to my advisor, Professor Jack B. Dennis, not only for his assistance on technical matters, but also for his unswerving confidence in my abilities. Finally, to my wife, Sandra, I extend special thanks for the typing of the manuscript, and for her encouragement and devotion.

Earl Van Horn

[*]Work reported herein was supported in part by Project MAC, an M.I.T. research program sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Number Nonr-4102(01).

"We cut up and organize the spread and flow of events as we do, largely because, through our mother tongue, we are parties to an agreement to do so, not because nature itself is segmented in exactly that way for all to see."

Benjamin Lee Whorf

Contents

List of Figures

Chapter I

Contemporary Multiprocessing

Introduction

This Chapter presents an explanation of selected problems and issues associated with contemporary multiprocessing. The explanation is organized to achieve two specific goals: (1) the establishment of a certain outlook, or viewpoint, toward the phenomenon of multiprocessing, in order to provide a base for the discussions in subsequent Chapters, and (2) the statement of the particular difficulties whose solutions are sought in subsequent Chapters. Thus Chapter I serves as an introduction for the Thesis as a whole.

The Chapter begins by introducing, through the use of examples, the notion of a single-process program. Next the notion of a multiprocess program is defined as a generalization of the notion of a single-process program, and multiprocessing is explained to be a phenomenon that involves the execution of a multiprocess program. Next some additional concepts are defined, and finally several problems and issues associated with contemporary multiprocessing are discussed in depth.

Single-Process Programs

A program is a recipe for transforming an initial set of values into a final set of values; for example, the following program tells how to find the largest value present in the list of numbers, $x_1, x_2, \ldots, x_n$.

```
1. Make $i$ be 1.
2. Make $j$ be 1.
3. If $i = n$ then the answer is $x_j$.
4. Subtract $x_i$ from $x_{i+1}$ and remember the result.
5. If the result is positive then add 1 to $j$.
6. Add 1 to $i$.
7. Go to step 3.
```

This program is composed of seven procedure steps, and the set of values to which the program refers is the set of quantities named i, j, n, and $x_1, x_2, \ldots, x_n$.

One can imagine execution of the above program by a clerk, which passes from one procedure step to the next, obeying the directions encountered at each step. Between procedure steps a clerk might remember information within itself; for example, between steps (4) and (5) of the above program the clerk retains the result of the subtraction performed at step (4). In general, a procedure step might direct a clerk to modify either the clerk's own information, or the value set, or both.

The sequence of actions that a clerk performs in executing a program is called a process [11, 26].* The above program is called a single-process program, because it directs the activity of exactly one clerk.

*Numbers in square brackets refer to items in the list of references, which follows the Appendices.

The simplest kind of digital computer, having one arithmetic unit and one memory unit, executes single-process programs. The arithmetic unit of such a computer can act as a clerk; then the state word[*] of the arithmetic unit is clerk information, and each word in the memory unit is value set information.

Multiprocess Programs

A multiprocess program is a set of procedure steps that directs the activities of two or more clerks. Since each processing unit, such as an arithmetic unit or i/o channel, of a computing system can act as a clerk, then any program permitting the simultaneous operation of processing units is an example of a multiprocess program.

One reason for specifying a computational activity in the form of a multiprocess program is that such a program can indicate an absence of sequencing constraints among portions of an activity by explicitly permitting several clerks to act simultaneously. Such permissions for simultaneity are desirable because they allow a system to perform a computational activity more rapidly than if the activity had to be performed sequentially.

Two examples of systems for multiprocessing, i.e., systems that can execute multiprocess programs, are the IBM 7090 [17], and the DEC PDP-1 [23]. In the 7090, the data channels and the arithmetic unit

[*]A typical arithmetic unit state word includes program counter, instruction register, and accumulator information. Additional remarks on state words are given by Conway [3].

can execute separate processes simultaneously: the procedure steps executed in the data channel processes are drawn from a set of "channel commands". In the PDP-1, the single-channel sequence break hardware switches the arithmetic unit between a main process and an interrupt process.

The term "multiprocessing" is often used to describe systems having more than one processing unit. Nevertheless, when "multiprocessing" is used in this contemporary sense, it is the author's suspicion that many of the problems actually being discussed might be more effectively studied as problems of multiprocess programming rather than as problems of the simultaneous use of physically distinct processing units. For this reason, the term "multiprocessing" is used throughout the Thesis to refer to a programmer's specification of potential multiple activity, and not to refer to a particular method for carrying out such a specification. For example, suppose a multiprocess program is written to direct the activities of two clerks. If two processing units are available, they might execute the program together, each unit playing the role of one of the program's clerks. On the other hand, if only one processing unit is available, it might play the role of now one clerk and then the other, alternating back and forth between the two clerks according to some arbitrary scheduling strategy. Multiprocessing occurs in both of these situations, because in both cases a multiprocess program is being executed.

The Computation State

An execution of either a single-process program or a multiprocess program is called a computation. At an instant during a computation, the computation state is denoted by the information contained in the clerks and value set of the program being executed. For example, during the execution of the single-process program mentioned at the beginning of this Chapter, the computation state at some instant is denoted both by the information contained within the executing clerk, and also by the information held in the value set, i.e., held in the quantities i, j, n, and $x_1, x_2, \ldots, x_n$. A system performing a computation exhibits a succession of computation states: each transition from one computation state to the next is caused by one or more processing units, each playing the role of one of the program's clerks.

In practice, four kinds of information are encoded into a computation's initial state: (1) the procedure steps of a program, (2) the initial values of the program's internal quantities, (3) the information initially held in the program's clerks, and (4) the input symbols to be read during the program's execution. In other words, the initial computation state holds, of the information controlling a computation's performance, just that portion which a user can specify either explicitly, or implicitly through the use of programs and data prepared by others.

Computing Facilities

Many discussions to be presented in the Thesis concern the manner in which a computing system might execute an individual program. A fact that might complicate these discussions is the fact that some computing systems can be simultaneously in the midst of executing two or more programs; such systems are said to be multiprogrammed [10]. The notion of a computing facility is introduced so that the manner in which an individual program is executed can be easily discussed without regard to whether the executing system is multiprogrammed.

A computing facility is a perhaps time-varying collection of hardware that executes at most one single-process or multiprocess program at a time. Every computing system provides at least one facility, but a computing system that is multiprogrammed can provide several facilities simultaneously. For example, a typical configuration of the CTSS system [14] provides 24 facilities. A user gains access to one of these facilities when he identifies himself to the system by "logging in".

In providing a facility, a computing system provides (1) input units by means of which a user can specify an initial computation state, (2) processing and storage units that can perform a computation beginning from such an initial computation state, and (3) output units that can produce for a user during such a computation one or more output streams, i.e., sequences of output symbols. An example of the production of three output streams is the writing of symbols on a typewriter, a printer, and a card punch unit.

An improved understanding of the way in which a computing system provides a facility can be achieved by studying some of the details of the way in which a computing system executes a program. A program refers to, or directs the activities of, a set of objects: each object to which a program refers is either a clerk, a value set quantity, an input device, or an output device. During the execution of a program, it is appropriate for the role of each object referred to by the program to be played by a specific and distinct physical unit. Nevertheless, there need not exist such a fixed correspondence between program objects and physical units. In particular, the role of a program object might be played by a succession of physical units, and during occasional intervals the role of an object might not be played by any unit.

To provide a physical unit as part of a facility is to provide a physical unit to play the role of one of the objects referred to by the program the facility is executing. As was mentioned, each physical unit of a facility might be provided either continuously or intermittently. Thus, when a computing system provides one or more facilities, the system allocates its available physical units among the objects that are referred to by the programs the facilities are executing. In general, this allocation is time-varying, i.e., dynamic; dynamic allocation results in several distinct correspondences between physical units and program objects as time progresses.

An example of dynamic allocation is found in the previously mentioned PDP-1 system. When the single facility provided by the PDP-1 executes a program that uses the single-channel sequence break feature, the PDP-1's memory unit is permanently allocated to hold the program's value set, but the PDP-1's arithmetic unit is dynamically allocated by the sequence break hardware to carry out alternately the activities of the program's main clerk and interrupt clerk. Other examples of dynamic allocation are found in the CTSS system. Here the central processing unit and the main memory unit are allocated dynamically among the clerks and the value set quantities, respectively, that are referred to in the programs of several facilities.

Arbitrarily-Timed Cooperative Multiprocessing

Clerks executing a multiprocess program are said to cooperate [11] whenever information is transmitted among them; such transmissions take place either through the internal memories of the clerks, or through the program's value set. Cooperative multiprocessing is assumed throughout the present study: a facility might allow inter-clerk cooperation, and a multiprocess program might direct such cooperation.

A multiprocess program is executed in an arbitrarily-timed manner just when there is at least one computation state for which the set of the program's clerks accomplishing the transition to the next computation state is at least partially determined by influences other than the current computation state. If a program's execution is arbitrarily-timed, then knowledge of the current computation state generally does not imply knowledge of the next computation state, because the next computation state can depend on the set of the program's clerks that happen to accomplish the next computation state transition.

An arbitrarily-timed facility is one that might execute a multiprocess program is an arbitrarily-timed manner. A facility might be arbitrarily-timed because of slight variations in the speeds of autonomous processing units, because of replacement of one system component by another of different speed, because of variations in the duration of i/o activity, or, perhaps most significantly, because of the scheduling strategy of a multiprogrammed system. Arbitrarily-timed facilities are assumed throughout the present study.

Output Functionality

A facility is output-functional just when each symbol produced in every output stream is a function only of the initial computation state.

An alternate description of an output-functional facility gives two intermediate definitions. A facility is (S, i, j)-output-definite just when S is an initial computation state and each S-initiated computation producing a j-th symbol in output stream i produces the same j-th symbol in output stream i. A facility if S-output-definite just when it is (S, i, j)-output-definite for each output stream, i, and each positive integer, j. Finally, a facility is output functional just when it is S-output-definite for each initial computation state, S.

Many contemporary facilities are S-output-definite only if S belongs to some proper subset of the initial computation states; such facilities require a user desiring output functionality to begin each of his computations from one of the S-output-definite initial computation states.

The present study describes methods for making the S-output-definite subset of a facility's initial computation states equal to the set of all the facility's initial computation states, so that computations will produce deterministic outputs regardless of programming mistakes or improper input data. In Chapter II an abstract machine for coordinated multiprocessing is described, and a proof is given in Chapter IV that a facility which behaves like such a machine is an output-functional facility.

The following two examples show that, in the absence of inter-clerk coordination constraints such as will be proposed, cooperative multiprocessing performed by an arbitrarily-timed facility is nondeterministic in a way that can permit the symbol at some position in an output stream to depend on factors other than the initial computation state. In the first example, two clerks are ready to write different symbols into the same output stream. If the clerks' order of execution is arbitrary, then either symbol can appear as the next symbol in the output stream.

The second example is depicted in Figure 1.1. Suppose a clerk computes each of a stream of output symbols using only a result that a second clerk computes and stores as a shared quantity. If the first clerk stops for an arbitrary time while the second clerk continues execution, then an arbitrary number of symbols might be omitted from the output stream. On the other hand, if the second stops and the first continues, then some particular symbol might appear in the output stream an arbitrary number of times.

No one ever questions the output functionality of facilities that execute only single-process programs, because single-process computing is inherently deterministic. Unfortunately, the term "computer" has been associated for so many years with single-process computing that users sometimes expect any apparatus bearing the stamp "computer" to

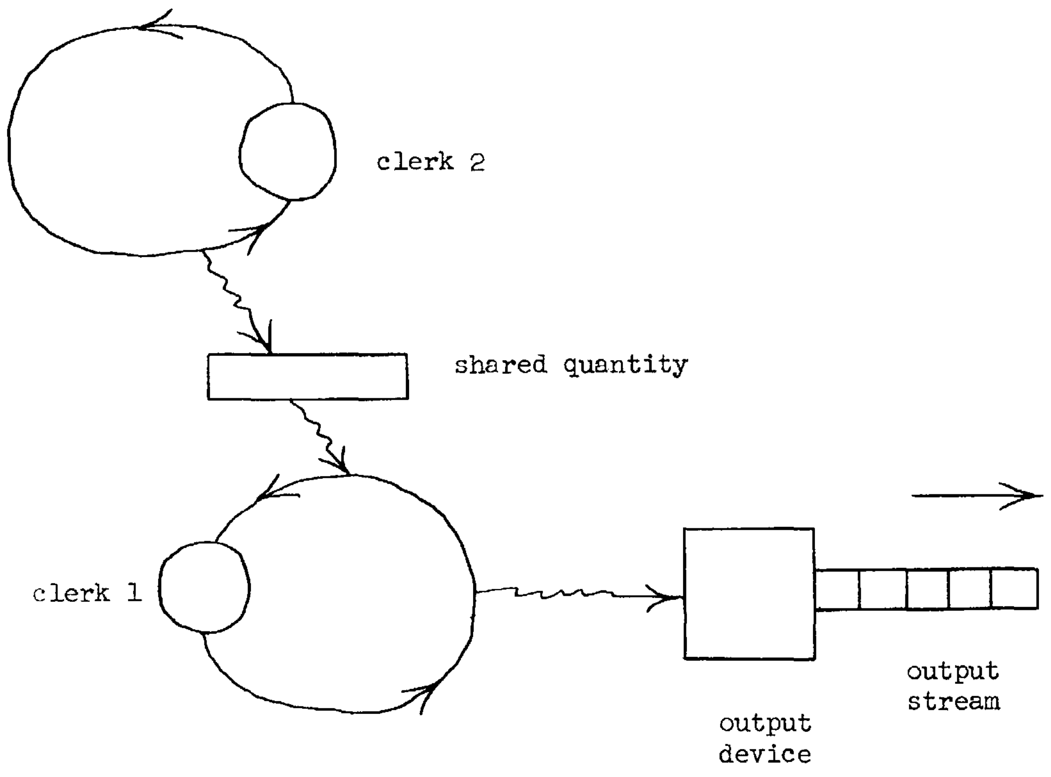


Figure 1.1. Example of output-nonfunctional multiprocessing.

exhibit a functional relationship between initial state and output streams. Thus the term "multiprocess computer" has a deterministic connotation, which is often misleading.

Lurking Bugs

As mentioned previously, many contemporary facilities for multiprocessing appear output-nonfunctional to their users. This lack of output functionality has not substantially hindered the development of computing applications, because programmers have been able to recognize the effects of arbitrary execution timing, and have been able to isolate these effects from the output streams of computations. For instance, a programmer might have introduced into the example of Figure 1.1 an interlock quantity to be tested by each clerk before the clerk proceeded with the reading or writing of the shared data quantity.

The seeking of deterministic outputs through the skillful programming of an output-nonfunctional facility is undesirable because the detection and the diagnosis of programming mistakes, i.e., bugs, are much more difficult in an output-nonfunctional facility than in an output-functional facility. The issues are best understood by considering again the example of Figure 1.1 in which two clerks cooperate using shared data and interlock quantities. As shown in Figure 1.2, suppose a bug in the procedure steps being executed by a third clerk causes the third clerk to change the value of the data quantity shared by clerks one and two. During some computations from a given initial state, this change does not affect the output stream, because at the instant of the change, clerk one has already read the shared data quantity's current

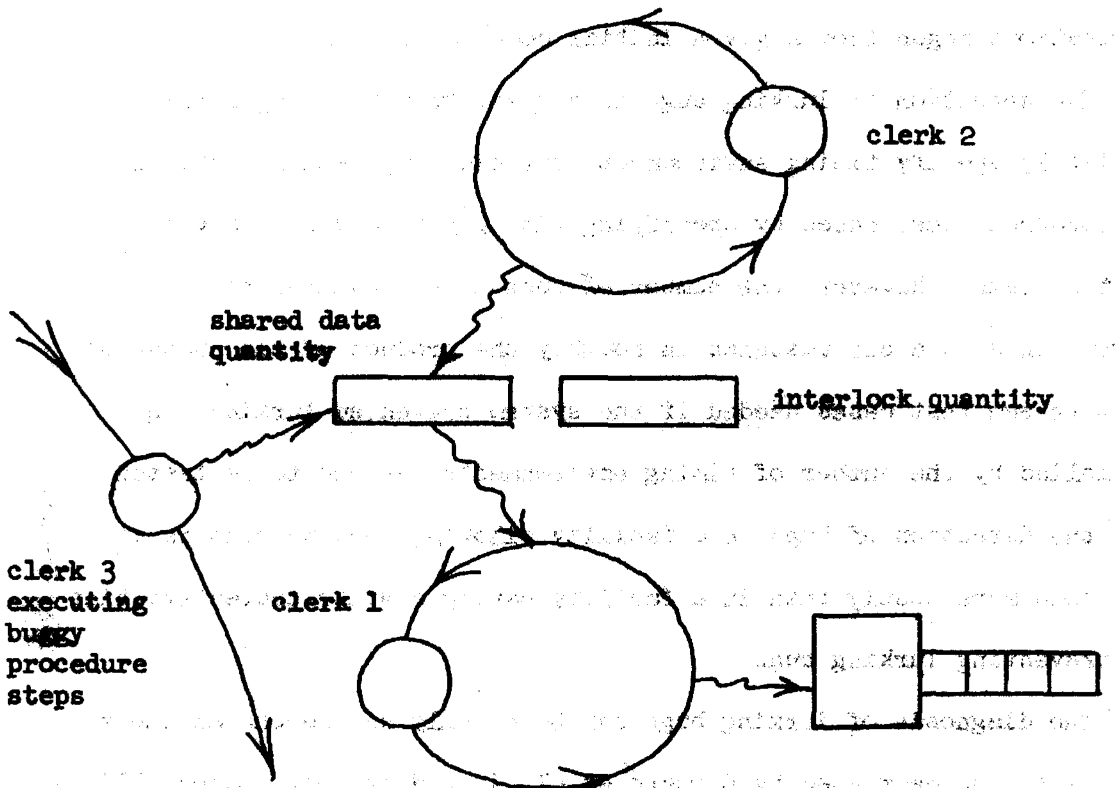


Figure 1.2. Lurking bug destroying output functionality.

value and clerk two has not yet set the next value. During other computations from the same initial state, the change does affect the output stream, because at the instant of the change, the current value of the shared data quantity has not yet been read by clerk one. The bug in the procedure steps being executed by clerk three is a lurking bug -- a bug whose effects are noticed during only some of the computations begun from a given initial computation state.

The detection of lurking bugs is easiest when the programmer can completely specify timing environments for his computations; then he can construct test cases by specifying timing environments as well as input streams. However, the number of test cases required to debug a program in such a circumstance is roughly the product of the number of input stream test cases needed if the system prevented lurking bugs multiplied by the number of timing environments desired to be tested. Thus the detection of bugs in a facility allowing lurking bugs is at best much more costly than in a facility having similar characteristics but preventing lurking bugs.

The diagnosis of lurking bugs can be equally as onerous as their detection. As mentioned by Corbató et al. in a different context [5], a lurking bug appears to the programmer to be indistinguishable from transient hardware failure. Breakpoints, dynamic dumps, traces, and other diagnostic tools all require the re-running of a computation at least once and possibly several times to isolate a bug into progressively smaller sections of a program. Unless the programmer can reproduce the timing environment of the computation in which a bug was detected, the programmer must choose between two alternatives: he can insert his diagnostic tools into the program and continue to run computations until

Providence shows the bug to him again, or he can try to diagnose using the program listing, the computation's output, and a dump of the final computation state. Both of these alternatives are costly as well as discouraging.

Lurking Bug Effects

The example of Figure 1.2 shows how a lurking bug's effect can be the destruction of output functionality. Such a lurking bug effect, i.e., the production of arbitrary output from identically-initiated computations, is called a non-functionality effect.

Another lurking bug effect, called a noncompletion effect, is the arbitrary curtailment of the production of output from identically initiated computations. An individual lurking bug might cause either a nonfunctionality effect, or a noncompletion effect, or both.

The following example describes a lurking bug causing a noncompletion effect. Suppose a bug in the procedure steps being executed by a clerk causes the clerk to change the procedure code being executed by a second clerk. During some computations from a given initial state, the second clerk has already executed the changed code, and so the change does not affect the second clerk's periodic execution of an output procedure step. During other computations from the same initial state, the second clerk executes the changed code, and thereupon begins executing a loop of procedure steps that excludes the output step the clerk would otherwise have periodically executed. The bug in the procedure steps being executed by the first clerk is a lurking bug that causes a noncompletion effect.

Facilities for multiprocessing that behave like the abstract machine to be described in Chapter II are proved in Chapter IV to prevent nonfunctionality effects. In Chapter V, such facilities are also proved to prevent noncompletion effects.

Contemporary Procedure Steps for Multiprocessing

To allow comparison between contemporary facilities for multiprocessing and the facilities to be discussed in subsequent Chapters, several types of procedure steps used in contemporary multiprocess programs are now described. In the next Section, procedure steps of some of these types are used in an exemplary program, into which a typing error could introduce a lurking bug.

The first type of procedure step is **fork** [1, 3, 10]. Execution of the step

```
    fork e;
```

creates a clerk that will start execution at the label e, and causes the clerk that executed the **fork** to pass to the next procedure step. The creation of a clerk means that there is an additional clerk participating in the computation; an immediately recruited processing unit might play the role of the new clerk, or a new entry might be placed in a list, called a *ready list* [10, 26], of clerks whose roles ought to be played.

Execution of the step

```
    quit;
```

[10] *deletes* the executing clerk. The deletion of a clerk means that

there is one less clerk participating in the computation; the processing unit that had played the role of a clerk being deleted might either join a pool of available processing units, or begin playing the role of the highest priority clerk on a ready list.

One of several schemes for programming the cooperation of simultaneously existing clerks is the scheme reported on by Dijkstra [11] involving *semaphore* quantities. Execution of the step

```
    V s;
```

adds one to a semaphore quantity, s, using an uninterruptible increment-memory instruction. Execution of the step

```
    P s;
```

causes the executing clerk to act as if it executed the following program:

1. If s is greater than zero then go to step 3.
2. Cease activity until s becomes greater than zero.
3. Subtract 1 from s without interruption.

If several clerks are at step (2) for s when s becomes greater than zero, then just one of these clerks, which is selected according to a priority discipline, proceeds to step (3).

Procedure steps of the types **V** and **P** are useful for directing the cooperation between a clerk producing data and a clerk receiving data, such as the clerks of Figure 1.1. In this application the semaphore quantity indicates the number of data values ready for consumption in a variable length buffer.

V and **P** procedure steps can also be used to restrict a *critical section* of a program's procedure steps to execution by one clerk at a time. Each critical section is preceded by a P and followed by a V, both referring to the section's unique semaphore quantity, which will only take the values 1 and 0.

Dijkstra has shown that critical section control can be achieved without using a procedure step whose execution requires uninterruptible read-alter-write action [12, 20]. The construction is so complicated, however, that it is mainly of theoretical interest.

Similar to the actions performed by P and V when used for critical section control are the actions of lockout in the D825 system [13], lock and unlock discussed by Dennis and Van Horn [10], and obtain and release described by Anderson [1]. Each procedure step of these types acts on a binary semaphore quantity that is associated with a data quantity specified by the procedure step.

As described by Conway [3], execution of the step

```
    join c;
```

subtracts 1 from c without interruption. If the result is negative or zero, the clerk proceeds to the next step; otherwise the clerk is deleted. The initial value of the quantity c is the number of clerks that must execute the join in order for one of them to proceed to the step following the join.

Dennis and Van Horn [10] describe a join similar to Conway's join, but which specifies a label to which a clerk should transfer rather than quit.

An Example of a Lurking Bug

To exemplify the use of some of the types of procedure steps described in the previous section, as well as to allow the illustration of a lurking bug, the following program is presented, which tells how to compute the expression

$$(AB)^{-1}((CD)(AB))$$

where A, B, C, and D are n by n matrices, and AB is assumed non-singular. The language in which the program is written is similar to the Algol language [2]. The join appearing at the label "last" is a Conway-type join. A flow-chart representation of the program is given in Figure 1.3.

```
begin     array T, U, Y, Z[1:n, 1:n];
          integer c, s;
          c := 2;
          s := 0;
          fork parpro;
          matrix multiply (A) times:(B) into:(T);
          V s;
          matrix inverse of (T) into:(U);
          goto last;
parpro:   matrix multiply (C) times:(D) into:(Y);
          P s;
          matrix multiply (Y) times:(T) into:(Z);
last:     join c;
          matrix multiply (U) times:(Z) into:(answer);
end
```

$$\begin{tikzcd}
 & 2 \rightarrow c \arrow[d] & \\
 & 0 \rightarrow s \arrow[d] & \\
 & \underline{\underline{\mathrm{fork}}} \arrow[dl] \arrow[dr] & \\
A \times B \rightarrow T \arrow[d] & & C \times D \rightarrow Y \arrow[d] \\
\underline{\underline{V}}\ s \arrow[d] & & \underline{\underline{P}}\ s \arrow[d] \\
T^{-1} \rightarrow U \arrow[dr] & & Y \times T \rightarrow Z \arrow[dl] \\
 & \underline{\underline{\mathrm{join}}}\ c \arrow[d] & \\
 & U \times Z \rightarrow \mathrm{answer} &
\end{tikzcd}$$

Figure 1.3. Flow-chart of a contemporary multiprocess program.

A clerk enters the program by passing to the symbol **begin**. The following activity ensues when exactly one clerk enters the program. First the entering clerk causes a new clerk to start execution at the label "parpro". Then the entering clerk, after computing the product AB, executes a **V** telling the parpro clerk that AB has been formed. Next the entering clerk inverts AB. The parpro clerk, after computing CD,* delays at a **P** until AB is available. Then the parpro clerk computes (AB)(CD). Finally, the two clerks join, and the last to arrive computes the final answer.

Consider the effect of a typographical error on this program: suppose that

```
          P s;
```

had been inadvertently typed as

```
          P c;
```

If the execution timing were such that the entering clerk proceeded all the way to the **join** before the parpro clerk reached the **P**, then a correct result would be obtained; otherwise an unreproducible incorrect result

*Applying the Algol rule of procedure body replacement to the above program, the simultaneous execution by both clerks of the procedure "matrix multiply" occurs as if each clerk executed a separate copy of the procedure. The need for separate copies of the procedure in an implementation can be avoided, however, through the use of *pure procedure* programming techniques [4], in which programs, instead of directing their own modification, direct the modification of data quantities *private* to each executing clerk [10].

could be expected. In particular, if the program were tested using a facility in which as many actions as possible were performed in one process before any actions were performed in another, then the effects of the typographical error would be hidden during debugging, but might be discovered on an application in which simultaneous processing occurred.

Chapter II

A Machine for Coordinated Multiprocessing

Introduction

This Chapter delineates a class of machines called machines for coordinated multiprocessing, or MCM's for short. An MCM is abstract, in the same sense that a Turing machine [8] or a finite automaton [24] is abstract. In principle, it is possible to construct an MCM; indeed, the structure and functioning of an MCM are best understood by imagining the existence of an actually constructed physical device. Nevertheless, it is desirable to keep in mind that an MCM is abstract, because an application of an MCM need not involve the MCM's straightforward construction as a physical device.

Although an extended discussion of MCM applications is postponed until Chapter III, it is useful to see examples of how an MCM can be applied in ways that do not constitute its straightforward construction. One example is the simulation of an MCM on a digital computer through the use of interpretive software. Another example is the use of an MCM as a model for analyzing, from an individual user's point of view, the behavior of a computing system's hardware and supervisory software. This last example can be expressed in another way: just as a CTSS [14] facility appears to a user like a virtual machine very similar to an IBM 7094, so also might some facility behave as if it were an actually constructed MCM.

In Chapter III, techniques and examples are presented to suggest that a facility which behaves like an MCM can be useful, and can be both constructed and programmed at reasonable cost. In Chapters IV and V, it is proved that nonfunctionality lurking bug effects and noncompletion lurking bug effects, respectively, do not occur in an MCM. That is, it is proved in these two Chapters that identically initiated MCM computations never produce arbitrary output, and that the production of output from identically initiated MCM computations never is arbitrarily cut short. The questions treated in Chapters III, IV, and V concern the sufficiency of an MCM for achieving various goals; questions concerning the necessity of an MCM remain open.

Preview

One of the purposes underlying the formulation of the MCM concept has been to bridge the gap, or provide a way station as it were, between the simplicity of automata theory and the complexity of contemporary computing systems. Specifically, the structure of an MCM is complex compared to the structure of a Turing machine or a finite automaton, but is clean and symmetrical compared to the structure of a contemporary computing system.

During the presentation of the details of MCM structure and operation, the reader may find it helpful to have in mind a broad framework upon which to sort out the details as they arrive. The following paragraphs establish this framework by mentioning the components of an MCM, and by indicating how these components work together to perform computations.

An MCM consists of a collection of _cells_, a _scheduler_, and a _count matrix_; Figure 2.1 shows an MCM with three cells. A cell might act like a main memory register of a contemporary computing system, or a cell might act like a processing unit of such a system. That is, a cell might store information passively, or a cell might actively perform a process. In performing a process, a cell performs a sequence of basic actions called _transactions_; in performing an individual transaction, a cell might read from or write into another cell, or might change an element of the count matrix.

The _computation state_ of an MCM is denoted by the information contained in the MCM's cells and count matrix. During the performance of a _computation_, an MCM takes on a succession of computation states: each transition from one computation state to the next is accomplished by the simultaneous performance of single transactions by one or more cells.

The cells that are to accomplish each computation state transition are selected by the scheduler. The scheduler's selection of such cells is affected by the immediately preceding computation state, and perhaps also by the unpredictable influences that make a facility for multiprocessing be arbitrarily-timed.

As was mentioned, a cell performing a transaction might change the count matrix. The count matrix, in turn, affects the scheduler's selecting of the cells that are to perform transactions. This loop of cause and effect -- from cells to count matrix to scheduler to cells --

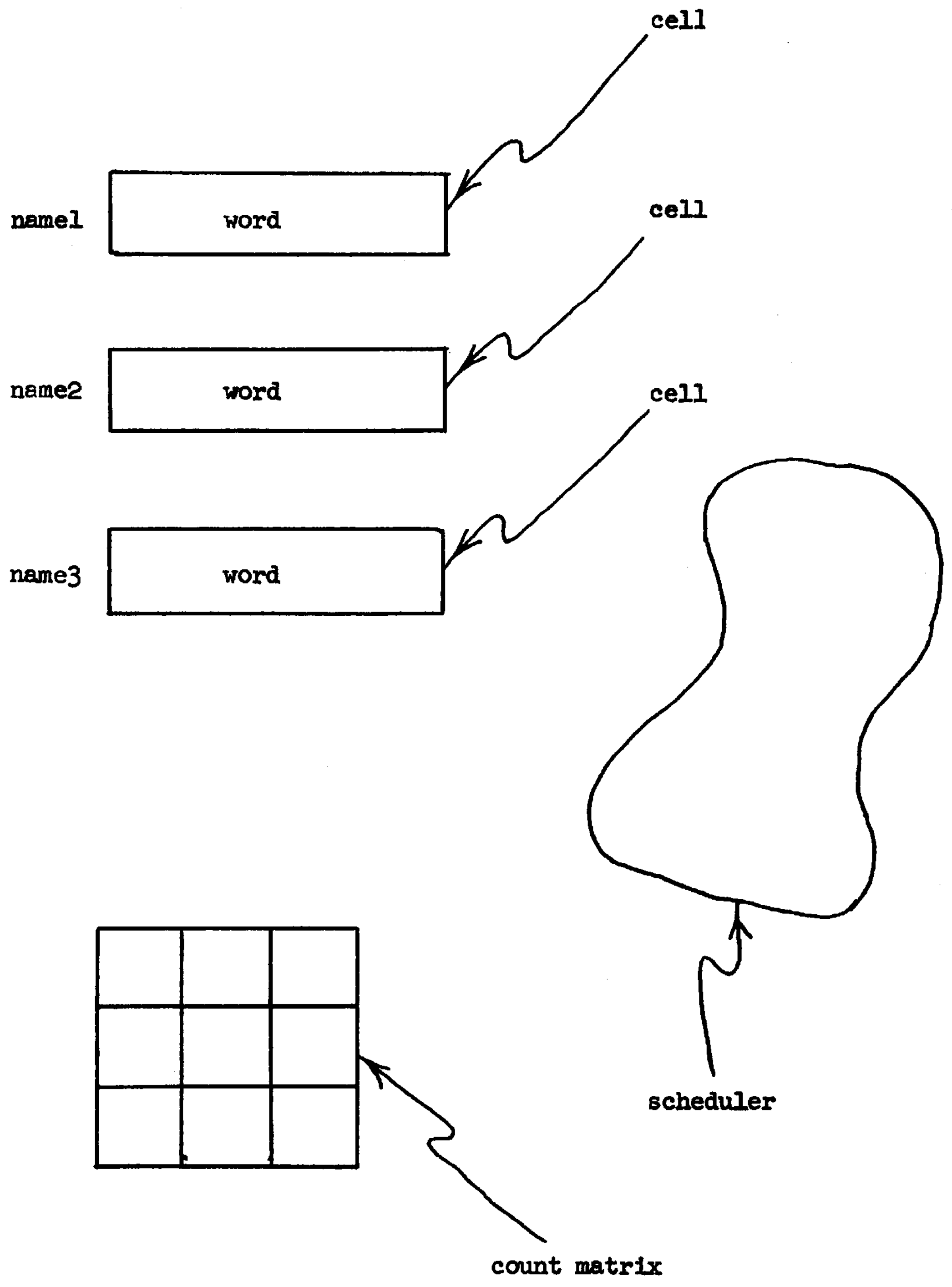


Figure 2.1. An MCM having three cells.

is the mechanism by which the activities of the cells are coordinated in order to prevent nonfunctionality and noncompletion lurking bug effects.

The author does not know of a contemporary facility for multiprocessing that behaves like an MCM in every respect. Nevertheless, there seems to be a certain similarity between MCM's and contemporary facilities, no doubt because the MCM concept grew out of an effort to model such facilities. Specifically, the cells of an MCM function in a way that is in many respects similar to the behavior of the processing units and main memory registers of a contemporary facility. Moreover, the scheduler of an MCM functions in a way that is somewhat similar both to the behavior of a supervisory computation that schedules processing units among programs to be executed, and to the behavior of electronic hardware that disciplines a queue of processing units awaiting access to a main memory unit. The count matrix of an MCM, however, does not appear to be similar to any feature of a contemporary facility.

Cells

A cell is unusual in that it can model either an active device like a processing unit, or a passive device like a memory register. A cell that models an active device is called a <u>clerk</u> cell, and a cell that models a passive device is called a <u>value</u> cell. A clerk cell performs a process that is a sequence of transactions, and a value cell stores information passively. As shown in Figure 2.1, the state of a cell is denoted by a <u>word</u>, and each cell is designated by a unique <u>name</u>.

It was mentioned that no contemporary facility is known that behaves like an MCM in every respect. Nevertheless, in order to understand the properties of the cells that occur in MCM's, it will be useful to observe how a collection of such cells might model the processing units and main memory registers of, say, an IBM 7090. Such a collection might contain 32,768 + 9 cells: one value cell to hold each memory word, one clerk cell to hold each data channel state word, and one clerk cell to hold the central processing unit state word. These cells might be named: 0, 1, ..., 32767, A, B, ..., H, CPU.

The state of an MCM's collection of cells can be described by means of a table in which the name of each cell is associated with the word held by the cell. Since each name is unique, such a table defines a single-valued function, called a <u>content function</u>, that takes the name of a cell into the word held by the cell; for example, if $c(\cdot)$ is the content function denoting some state of the 7090 cells described above, then the memory word at location 5 is $c(5)$, and the state word of channel B is $c(B)$. If $c(\cdot)$ is a content function and x is a name, then the word $c(x)$ is called the <u>content</u> of cell x. As an MCM passes from one computation state to the next, its collection of cells takes on a succession of states, each distinct state being described by a different content function.

An interesting property of an MCM is that the designation of a cell as either a clerk cell or a value cell can be a function of time. In this respect, the cells of the 7090 model mentioned above are not typical of MCM cells in general. Although each cell of the 7090 model is for all time either a clerk cell or a value cell, the general case is that a cell might alternate between clerk and value status as time progresses.

This alternation of cells between clerk and value status is similar to the alternation of modules between active and non-active roles in an abstract iterative computer described by Holland [16].

The clerk or value status of a cell is determined at any instant solely from the count matrix, in a manner to be described later. A clerk cell is changed into a value cell, or a value cell is changed into a clerk cell, by making an appropriate change in the count matrix.

Transactions

It was mentioned that the process each clerk cell performs is a sequence of basic actions called transactions. A clerk cell's performance of an individual transaction generally involves an interaction between the clerk cell and its environment. For example, suppose the CPU cell of the 7090 model mentioned above executes a store-accumulator instruction that does not specify indirect addressing. This execution is a sequence of two transactions: the first transaction reads from a value cell the encoded store-accumulator instruction, and the second transaction writes into perhaps some other value cell the accumulator information held in the CPU cell. These two transactions are of the types <u>get</u> and <u>put</u>, respectively. Gets, puts, and transactions of three other types are explained later in detail.

Each computation state transition is accomplished by the simultaneous performance of single transactions by one or more clerk cells. After the scheduler has selected the clerk cells that are to carry out a computation state transition, the scheduler simultaneously transmits a single <u>go pulse</u> to each clerk cell that has been selected. Upon receipt of a go pulse a clerk cell performs, not a series of transactions, but exactly one transaction; specifically, a clerk cell performs exactly one transaction when and only when it receives a go pulse. The scheduler's operation, which will be described later, is such that only clerk cells receive go pulses.

When a clerk cell receives a go pulse, the transaction the cell performs is determined both by the cell's identity, and by the cell's content. For example, during the previously mentioned store-accumulator execution, the first transaction performed by the CPU cell is determined both by the fact that the cell is the CPU cell rather than, say, the B cell, and by the fact that the content of the cell has cycle information saying that an instruction fetch activity is to take place next.

As shown in Figure 2.2 for the MCM of Figure 2.1, a cell's identity determines a transaction table, within which the cell's content determines a transaction. Specifically, in a cell's transaction table each possible content of the cell is associated with the type and parameters of a transaction; that is, if a cell might hold any one of n words, then there are exactly n word-transaction pairs in the cell's transaction table. At any instant, the transaction associated in this way with a cell's content is the transaction the cell would perform if it were to receive a go pulse at that instant. In other words, if the content of a cell is a certain word, and one wishes to know the transaction the cell would perform upon receipt of a go pulse, one need only look up the word in the cell's transaction table in order to obtain the type and parameters of the desired transaction.

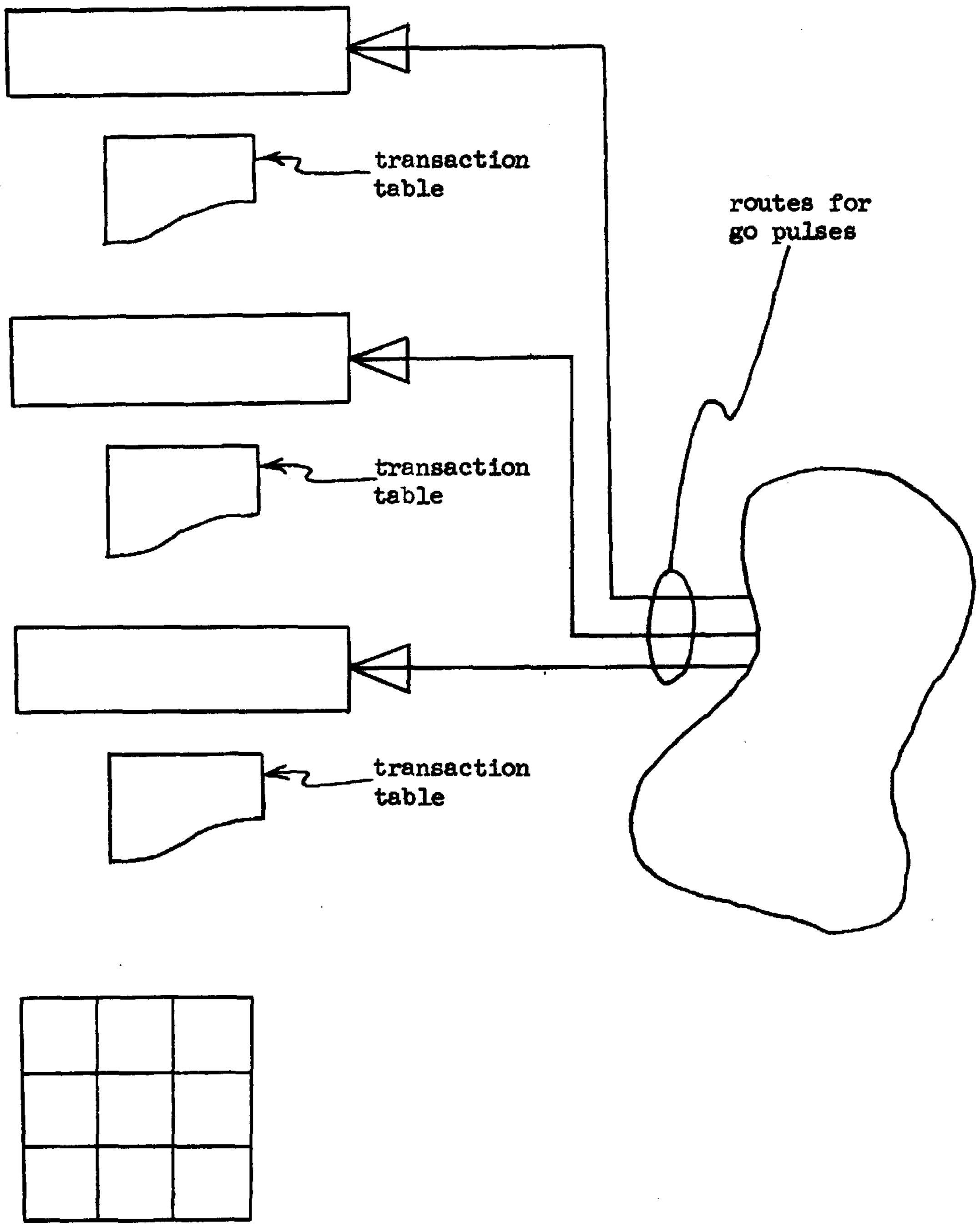


Figure 2.2. Additional properties of the MCM having three cells.

Every cell has a transaction table, because, in general, every cell is potentially a clerk cell. A cell's transaction table is permanently associated with the cell; in other words, a cell's transaction table does not vary with time. In the 7090 model, the transaction table of the CPU cell describes the wired-in properties of the 7090 central processing unit -- in the same way that the state transition table of a sequential machine describes certain wired-in properties of the machine.

Get Transactions

The first transaction of the previously mentioned store-accumulator execution is of the type get. A typical get transaction is described in a transaction table as follows.

$$\text{get of } i \text{ replace } f(\cdot) \tag{2.1}$$

The two parameters of a get are its operand name, such as i in (2.1), and its replacement function, such as $f(\cdot)$ in (2.1). In performing the transaction described by (2.1), a clerk cell reads the content of cell i, evaluates the function $f(\cdot)$ with this content as argument, and causes itself to hold the result of this evaluation. For example, suppose $c(\cdot)$ denotes the state of a collection of cells at some instant, and suppose (2.1) is the transaction that corresponds to $c(x)$ in clerk cell x's transaction table. If at this instant clerk cell x receives a go pulse, then clerk cell x will be caused to hold the word $f(c(i))$.

The performance of a get by clerk cell x changes the content of clerk cell x from some word, A, to some word, B. Let us review the way in which the word B is determined. First, the name x determines a transaction table. Second, the word A determines in this table a particular transaction -- in this case a get whose parameters are, say i and $f(\cdot)$. Third, the content of cell i is used as the argument of $f(\cdot)$ to determine the word B.

As noted above, the first transaction of the previously mentioned store-accumulator execution is a get. This get is determined by the word W that is held in the CPU cell just before the get's performance. The get's operand name is W's instruction location counter information, and the content of the cell named by this operand name is the encoded store-accumulator instruction. If the get's replacement function is evaluated with the encoded instruction as argument, the function yields for placement in the CPU cell a word X consisting of (1) instruction register information denoting the encoded instruction's operation part, (2) storage register information denoting the encoded instruction's address part, (3) cycle information saying that an instruction execution activity is to take place next, (4) instruction location counter information denoting a name numerically one greater than the name denoted by W's instruction location counter information, and (5) other information, such as accumulator and index register information, identical to the corresponding information of W. In other words, the replacement function of the get associated with W tells how the word X, which replaces W, depends on the encoded instruction.

The above example illustrates that in practice the state of a processing unit is usually the state of a collection of several information-storing elements, such as flip-flops or ferrite cores. When a processing unit performs a get, not every information-storing element need change state. In fact, the replacement function of some particular get may be such that the states of certain elements never change during the get's performance, regardless of the word read as operand. Thus the performance of a get may involve a change in only "part of" the word held in a processing unit. For present purposes, it will be convenient to think of the performance of a get as always changing the "entire" word held in a processing unit, and recognize that a get's performance might actually change the state of only some of a unit's information storing elements. In fact, for understanding the formal properties of an MCM, one may think of a word, not as a string of digits denoting states of separate elements, but as one symbol selected from some alphabet; with the latter point of view it is meaningless to speak of replacing "part of" a word.

Put Transactions

The second transaction of the previously mentioned store-accumulator execution is of the type put. A typical <u>put</u> transaction is described in a transaction table as follows.

$$\text{put of } i \text{ with } v \text{ replace } w \qquad (2.2)$$

The three parameters of a put are its operand name, such as i in (2.2), its <u>operand word</u>, such as v in (2.2), and its <u>replacement word</u>, such as w in (2.2). In performing the transaction described by (2.2), a clerk cell causes itself to hold the word w, and causes cell i to hold the word v; if the clerk cell and cell i are the same cell, then the clerk cell causes itself to hold w, and v is ignored. For example, suppose $c(\cdot)$ denotes the state of a collection of cells at some instant, and suppose (2.2) is the transaction that corresponds to c(x) in clerk cell x's transaction table. If at this instant clerk cell x receives a go pulse, then clerk cell x will be caused to hold the word w, and, if i is not equal to x, cell i will be caused to hold the word v.

As noted above, the second transaction of the previously mentioned store-accumulator execution is a put. This put is determined by the word X that is held in the CPU cell just before the put's performance. The put's operand name is X's storage register information. The put's operand word, which is the word to be placed in the cell named by the operand name, is X's accumulator information. The put's replacement word, which is the word to be placed in the CPU cell, is a word consisting of (1) cycle information saying that an instruction fetch activity is to take place next, and (2) other information identical to the corresponding information of X.

Procedure Steps

A program may be stored within a subset of an MCM's cells. When an MCM holds a program in this way, each procedure step of the program is encoded into one or more words, and each word is held by a distinct cell. For example, the previously mentioned 7090 model might hold a program that has as one of its procedure steps the previously mentioned store-accumulator instruction; this instruction is a procedure step that is encoded into exactly one word.

The execution of a procedure step begins with the performance of a get whose operand name is the name of a cell holding an encoded procedure step word. A procedure step execution might consist only of the performance of such a get, or a procedure step execution might consist of the performance of such a get followed by the sequential performance of one or more transactions, each of which might be either (1) a get whose operand name is the name of a cell holding an encoded procedure step word, or (2) a transaction, of any of the five types, whose performance is requested by the procedure step itself. For example, a program for the 7090 model might contain procedure steps such as: (1) a transfer-on-minus instruction, whose execution is the performance of a single get, (2) a store-accumulator instruction with indirect addressing, whose execution is the performance of a sequence of two or three gets followed by a put, and (3) a convert-by-replacement-from-the-MQ instruction, whose execution is the performance of a sequence of some number of gets from 1 through 256.

Reading and Writing

When one says that some cell reads another cell, the notion of "reading" is fairly clear from contemporary usage. Nevertheless, it is desirable to have a more precise notion of what it means to read a cell. In the MCM design as introduced thus far, the notion of reading is applicable in two contexts. The first application is intuitive: a clerk cell performing a get reads from the cell named by the get's operand name. The second application is more subtle: a clerk cell performing either a get or a put reads from itself, because the clerk cell's content just before the performing of the get or put determines the particular get or put to be performed. Based on these examples, a definition of the notion of reading may be given: a clerk cell x that is performing a transaction reads from a cell i just when x senses the content of i in order to help determine x's performance of the transaction.

Similar remarks may be made concerning the notion of "writing". In the MCM design as introduced thus far, the notion of writing is applicable in two contexts. The first application is intuitive: a clerk cell performing a put writes into the cell named by the put's operand name. The second application is more subtle: a clerk cell performing either a get or a put writes into itself. Based on these examples, a definition of the notion of writing may be given: a clerk cell x writes into a cell i just when x causes a new word to be held by i, where the new word might or might not be the same as the word previously held by i.*

Outputting

An explanation can now be given of the method by which an MCM produces output symbols. Some of an MCM's cells are output cells; the designation of a cell as an output cell does not vary with time, and is in addition to the time-varying designation of the cell as either a clerk cell or a value cell. There is a one-to-one correspondence between the output cells of an MCM and the output streams that are produced when the

*The concepts of reading and writing described here are no doubt the same as the concepts analyzed by Maurer in his discussion of the "input regions" and "output regions" of computer instructions [21].

MCM performs a computation. Whenever a word is written into an output cell, the word is not only caused to be held by the cell, but also is produced as the next symbol in the cell's output stream.

The Scheduler

At any instant when no transactions are being performed, the computation state of an MCM is denoted by the information contained in the MCM's cells and count matrix. When an MCM performs a computation, the MCM takes on a succession of computation states: each transition from one computation state to the next is accomplished by the simultaneous performance of single transactions by one or more clerk cells.

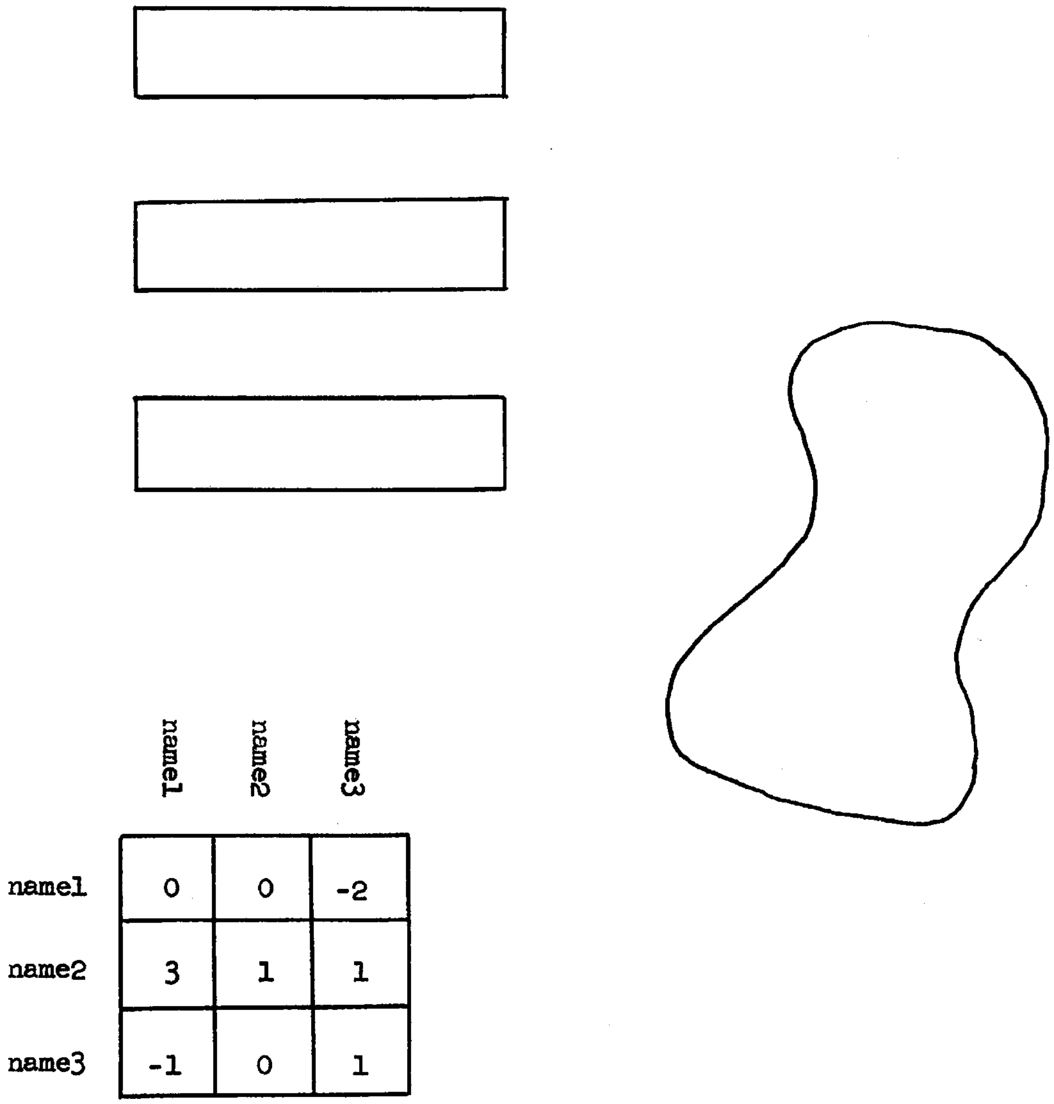
The set of clerk cells that accomplish a computation state transition is selected by the scheduler on the basis of the immediately preceding computation state, the transaction tables, and perhaps on the basis of unpredictable influences. In making such a selection, the scheduler behaves as if it carried out in sequence the following three activities: (1) the determination of an enable set, which is a set of names of enabled clerk cells, (2) the determination of a choice collection, which is a collection of subsets of the enable set, and (3) the selection of a member of the choice collection. The selected member of the choice collection is a set of names of cells; it is to these cells that the scheduler transmits simultaneous go pulses in order to initiate a computation state transition. The scheduler's activities are described later in more detail.

The performance of a computation proceeds as follows. At an instant when no transactions are being performed, i.e., when some computation state prevails, the scheduler selects a set of clerk cell names in the manner outlined above and transmits one go pulse simultaneously to each of the cells named in the set. After the transactions triggered by these go pulses are finished, a new computation state prevails. At a subsequent instant, the activity is repeated, i.e., the scheduler again selects and sends pulses to clerk cells, which then accomplish a transition to still another computation state. The succession of computation states produced in this way begins from an initial computation state specified by a user.

The Count Matrix

In determining an enable set, the scheduler refers to the count matrix. If an MCM contains n cells, then the MCM's count matrix is an n by n matrix. Each element of a count matrix is an integer, called a count, that can be positive or negative.

The information contained at some instant in an MCM's count matrix allows statements to be made such as, "cell x has read capability for cell i", and "cell x has write capability for cell i". Cell x has read capability for cell i just when the count at row x and column i of the count matrix is greater than zero. Cell x has write capability for cell i just when cell x is the only cell that has read capability for cell i. The capabilities of the cells of the MCM shown in Figure 2.3 are the following: name2 has read capability for name3, name2 has write capability for both itself and name1,



	name1	name2	name3
name1	0	0	-2
name2	3	1	1
name3	-1	0	1

Figure 2.3. Typical count matrix configuration of the MCM having three cells.

and name3 has read capability for itself. The significance of the fact that a cell has a read capability for another cell is explained in the following Section.

The Enable Set

At any instant between computation state transitions, the scheduler's enable set is just the set of names of those cells that are enabled clerk cells. In order to decide whether a particular cell is an enabled clerk cell, the scheduler first examines the cell's content and transaction table in order to discover the transaction the cell would perform if it were to receive a go pulse; then the scheduler makes its decision for the examined cell by applying the enabling rule for the discovered transaction. The enabling rules for gets and puts are described below. The enabling rules for transactions of the other three types will be described as these types are introduced.

The enabling rule for a get is the following: a cell that would perform a get upon receipt of a go pulse is an enabled clerk cell just when the cell has both write capability for itself, and read capability for the cell named by the get's operand name. In the example of Figure 2.3, suppose the name2 cell would perform a get of name3 upon receipt of a go pulse; then the name2 cell is an enabled clerk cell.

The enabling rule for a put is the following: a cell that would perform a put upon receipt of a go pulse is an enabled clerk cell just when the cell has both write capability for itself, and write capability for the cell named by the put's operand name. In the example of

Figure 2.3, suppose the name2 cell would perform a put of name1 upon receipt of a go pulse; then the name2 cell is an enabled clerk cell.

For gets and puts, the notions of reading and writing correspond in a simple way to the notions of read capability and write capability, respectively; this correspondence is seen in the following restatement of the above enabling rules. A cell that would perform either a get or a put upon receipt of a go pulse is an enabled clerk cell just when the cell has read capability for each cell it would read, and has write capability for each cell it would write.

Nomenclature for Cells

A cell is a clerk cell just when it has write capability for itself; otherwise the cell is a value cell. In other words, whether a cell is a clerk cell or a value cell is based, not on the cell's construction, but on information contained in the count matrix. In general, any cell can be a clerk cell; all it takes to make a value cell into a clerk cell or a clerk cell into a value cell is the performance of appropriate transactions, of types to be described later, that modify the count matrix.

Although "being a clerk" is not really a property of a cell itself, it is sometimes convenient to call a cell a "clerk cell". Specifically, it has been convenient, and will continue to be convenient, to call a cell a "clerk cell" whenever it is known or assumed that the cell has write capability for itself.

A cell can be a clerk cell and not be an enabled clerk cell; such a cell is called a disabled clerk cell. For example, a cell that would perform a get upon receipt of a go pulse is a disabled clerk cell just when the cell has write capability for itself and does not have read capability for the cell named by the get's operand name.

The Choice Collection for Gets and Puts

In the event that each member of an enable set is the name of a cell that would perform either a get or a put upon receipt of a go pulse, then the choice collection determined from this enable set is just the collection of the non-empty subsets* of the enable set.

At an instant between computation state transitions, the scheduler selects one set belonging to the choice collection derived from the current computation state. As mentioned previously, the selected member of the choice collection is a set of names of cells; it is to these cells that the scheduler transmits simultaneous go pulses in order to initiate a computation state transition. The method by which the scheduler selects a member of the choice collection will be described later.

*The subsets of the set $\{a, b\}$ are the sets: $\{\}$ (the empty set), $\{a\}$, $\{b\}$, and $\{a, b\}$.

Send, Done, and Bye Transactions

During a computation, the count matrix is set initially by the specification of an initial computation state, and thereafter is changed only by performance of transactions of the types send, done, and bye.

A typical send transaction is described in a transaction table as follows.

send of i to e replace w (2.3)

The three parameters of a send are its operand name, such as i in (2.3), its sendee name, such as e in (2.3), and its replacement word, such as w in (2.3). In performing the transaction described by (2.3), a clerk cell both causes itself to hold w, and adds 1 to the count at row e and column i of the count matrix. For example, let K be the count at row e and column i of the count matrix just before the performance of (2.3); then the performance of (2.3) has one of three effects: (1) if K is 0, the performance gives cell e read capability for cell i, (2) if K is less than 0, the performance reduces the number of sends of i to e that might be performed to give cell e read capability for cell i, or (3) if K is greater than 0, the performance increases the number of dones of i, described below, that cell e might perform to relinquish its read capability for cell i.

The enabling rule for a send is similar to the enabling rule for a get: a cell that would perform a send upon receipt of a go pulse is an enabled clerk cell just when the cell has both write capability for itself, and read capability for the cell named by the send's operand name. To perform a send, a cell need not have any capability for the cell named by the send's sendee name.

A typical done transaction is described in a transaction table as follows.

done of i replace w (2.4)

The two parameters of a done are its operand name, such as i in (2.4), and its replacement word, such as w in (2.4). In performing the transaction described by (2.4), a clerk cell named x both causes itself to hold w, and subtracts 1 from the count at row x and column i of the count matrix. For example, let K be the count at row x and column i of the count matrix just before clerk cell x's performance of (2.4); then the performance of (2.4) has one of three effects: (1) if K is 1, the performance takes away cell x's read capability for cell i, (2) if K is less than 1, the performance increases the number of sends of i to x that might be performed to give cell x the read capability for cell i, or (3) if K is greater than 1, the performance reduces the number of dones of i that cell x might perform to relinquish its read capability for cell i.

The enabling rule for a done is the following: a cell that would perform a done upon receipt of a go pulse is an enabled clerk cell just when the cell has write capability for itself. To perform a done, a cell need not have any capability for the cell named by the done's operand name.

A typical bye transaction is described in a transaction table as follows.

bye to e replace w (2.5)

The two parameters of a bye are its sendee name, such as e in (2.5), and its replacement word, such as w in (2.5). In performing the transaction described by (2.5), a clerk cell named x causes itself to hold w, subtracts 1 from the count at row x and column x of the count matrix, and adds 1 to the count at row e and column x of the count matrix; if e is equal to x, no alteration is made to the count matrix.

The enabling rule for a bye is similar to the enabling rule for a done: a cell that would perform a bye upon receipt of a go pulse is an enabled clerk cell just when the cell has write capability for itself. To perform a bye, a cell need not have any capability for the cell named by the bye's sendee name.

The effect of clerk cell x's performance of (2.5) would seem to be equivalent to the effect of clerk cell x's performance of the sequence

done of x replace w_1 (2.6)

send of x to e replace w (2.7)

where w_1 is a word to which (2.7) corresponds in cell x's transaction table. Although this equivalence holds in many situations, nevertheless transactions of the type bye are not redundant. In particular, if bye transactions were omitted from the MCM design, then it would not be possible for the capability to write into a cell to be held at one instant by the cell itself, and to be held at a later instant by some other cell. The non-redundancy of bye transactions is explained more fully in Appendix A.

An example showing how send and done transactions can be used to coordinate processes will be given later in this Chapter. A discussion of the use of bye transactions is postponed until Chapter III.

The Choice Collection

The rule by which the choice collection is determined from an enable set has been given for the case in which each member of the enable set is the name of a cell that would perform either a get or a put upon receipt of a go pulse. The complete rule for the determination of the choice collection is the following: the choice collection that is determined from an enable set is just the collection of the non-empty subsets of the enable set, but excluding those subsets that contain the names of two or more cells that, upon receipt of a go pulse, would alter the same element of the count matrix.

The enabling rules and the rule for determining a choice collection guarantee that simultaneously performed transactions do not conflict; that is, during a computation state transition, the following three statements are true: (1) each cell is written by no more than one clerk cell, (2) each cell that is both read and written is written by the same clerk cell that read the cell, and (3) each element of the count matrix is altered by no more than one clerk cell.

Scheduling Strategies

The scheduler makes selections from the choice collections derived from successive computation states in such a way that the scheduler's strategy is reasonable. In order to understand the notion of a reasonable scheduling strategy, one must observe that once a clerk cell becomes enabled, it remains enabled at least until it receives a go pulse. This statement is true because a clerk cell can only become disabled as the result of its own performance of a send or bye transaction.

The scheduler's strategy is reasonable if and only if each enabled clerk cell always receives a go pulse a finite time after the clerk cell becomes enabled. The fact that the scheduler's strategy is reasonable is used in the proof, given in Chapter V, that an MCM prevents noncompletion lurking bug effects.

Subject to the restriction that the scheduler's strategy must be reasonable, the scheduler's selections from successive choice collections may be made according to any rules whatsoever, and in accordance with any causes or influences whatsoever. That is, any strategy at all for making selections from choice collections is an acceptable strategy for making such selections, provided the strategy is reasonable in the above sense. For example, an acceptable scheduling strategy may involve remembering information about the scheduler's previous selections, or about the previous activity of the MCM. Furthermore, the selections of an acceptable scheduling strategy may be affected by unpredictable influences, such as the influences that make a facility for multiprocessing be arbitrarily-timed.

Specifying a Well-Defined MCM

The description of the structure and functioning of an MCM is now complete. Within the framework that has been described, many different MCM's may be specified, differing, for example, in the number of cells, or in the transactions that cells can perform. In general, an MCM is specified by giving (1) a set of cell names, (2) a set of output cell names that is a subset of the set of cell names, (3) a transaction table for each cell, and (4) a set of computation states that are the states which may serve as initial computation states.

For present purposes it is desirable that an MCM be <u>well-defined</u>: an MCM is well-defined if and only if during no computation performed by the MCM is an evaluation of a transaction table or a replacement function ever attempted with an undefined argument. That is, in a well-defined MCM, the transaction tables and replacement functions provide information sufficient to guide the MCM's activity in every possible circumstance. Every MCM discussed in the Thesis is assumed to be well-defined. Appendix B gives both a more precise definition of a well-defined MCM, and a condition sufficient for an MCM to be well-defined.

Coordination of Processes

Throughout the Chapter, MCM behavior has been explained in terms of the actions taken to accomplish an individual computation state transition. In order to achieve additional insight into the way in which an MCM coordinates processes, let us now examine MCM behavior from a different point of view: let us focus attention on just a few clerk cells, and observe, over the course of several computation state transitions, how these clerk cells might interact with each other as they perform their individual processes, i.e., as they perform their individual sequences of transactions.

The specific example to be considered concerns three clerk cells x, y, and z that communicate using a cell i. At time t_1 let clerk cell x have write capability for cell i, and let $k_1(\cdot, \cdot)$ describe the state of the count matrix, i.e., let $k_1(x, i)$ be the count at row x and column i of the count matrix. Let

$$k_1(x, i) = 1$$
$$k_1(y, i) = 0$$
$$k_1(z, i) = 0$$

As MCM operation proceeds subsequent to time t_1, clerk cell x might write a word w into cell i, and then, as shown in Figure 2.4, might make w available to clerk cell y by performing a send of i to y. At the time t_2 just after this performance, the count matrix $k_2(\cdot, \cdot)$ would be such that

$$k_2(x, i) = 1$$
$$k_2(y, i) = 1$$
$$k_2(z, i) = 0$$

Then clerk cells x and y, but not clerk cell z, might read from cell i any number of times, but no clerk cell would be allowed to perform a write into cell i.

Next, clerk cell y, in the midst of its use of the word w made available to it by clerk cell x, might make the word w available to clerk cell z by performing a send of i to z. At the time t_3 just after this performance, the count matrix $k_3(\cdot, \cdot)$ would be such that

$$k_3(x, i) = 1$$
$$k_3(y, i) = 1$$
$$k_3(z, i) = 1$$

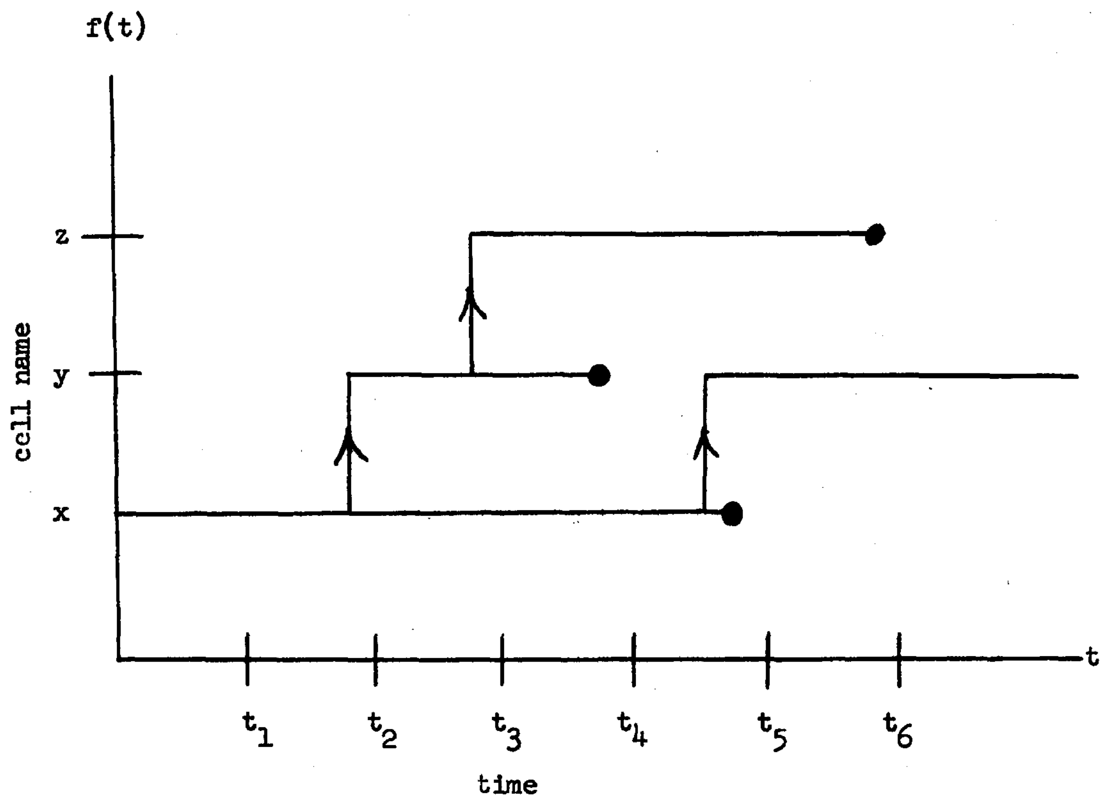


Figure 2.4. Locus of read capabilities for a cell i in an example of process coordination. The multiple-valued function $f(\cdot)$ defined by solid lines is such that $f(t) = x$ if and only if the count in row x and column i of the count matrix equals 1 at time t. An arrow from x to y at time t denotes the performance by cell x of a send of i to y at time t. A dark circle for x at time t denotes the performance by cell x of a done of i at time t.

Then all three clerk cells might read from cell i any number of times, but none would be allowed to write into cell i.

Next, clerk cell y might perform a done of i, thus indicating it has no more actions to take with respect to the current content, w, of cell i. At the time t_4 just after this performance, the count matrix $k_4(\cdot, \cdot)$ would be such that

$$k_4(x, i) = 1$$
$$k_4(y, i) = 0$$
$$k_4(z, i) = 1$$

Later, clerk cell x, to allow the next content of cell i to be written by clerk cell y, might perform a send of i to y, followed by a done of i. At the time t_5 just after the performance of the done, the count matrix $k_5(\cdot, \cdot)$ would be such that

$$k_5(x, i) = 0$$
$$k_5(y, i) = 1$$
$$k_5(z, i) = 1$$

Next clerk cell z might perform a done of i, thus indicating it has no more actions to take with respect to the current content, w, of cell i. At the time t_6 just after this performance, the count matrix $k_6(\cdot, \cdot)$ would be such that

$$k_6(x, i) = 0$$
$$k_6(y, i) = 1$$
$$k_6(z, i) = 0$$

Then clerk cell y, having write capability for cell i, would be able to write a new content into cell i, and would be able to make this new content available to other clerk cells. Notice that if clerk cell y had tried to write a new content into cell i before time t_6, then clerk cell y would have been disabled until t_6, and so would not have performed the write of cell i until after t_6.

The behavior of the clerk cells in the example given in the preceding paragraphs is typical of the manner in which clerk cells communicate during an MCM computation. Additional occurrences of interest can be seen in variations of this example. Specifically, to observe an occurrence of a negative count matrix element, refer to Figure 2.5, and recall in the example the instant t_2, just after clerk cell y was first given read capability for cell i. If after t_2 clerk cell z had proceeded much more rapidly than clerk cell y, and did not attempt to read w, then before time t_3 clerk cell z would have performed a done of i, and the resulting configuration of the count matrix $k_a(\cdot, \cdot)$ prior to t_3 would have been such that

$$k_a(x, i) = 1$$

$$k_a(y, i) = 1$$

$$k_a(z, i) = -1$$

This variation of the example shows that the possibility of counts being less than zero allows a clerk cell to anticipate, in effect, its lack of need for a read capability before being given the capability.

To see an occurrence of a count greater than one, refer to Figure 2.6, and recall in the example the instant t_3, just after clerk cell z was given read capability for cell i. If after t_3 clerk cell x had proceeded much more rapidly than clerk cell y, then before time t_4 clerk cell x would have performed a send of i to y

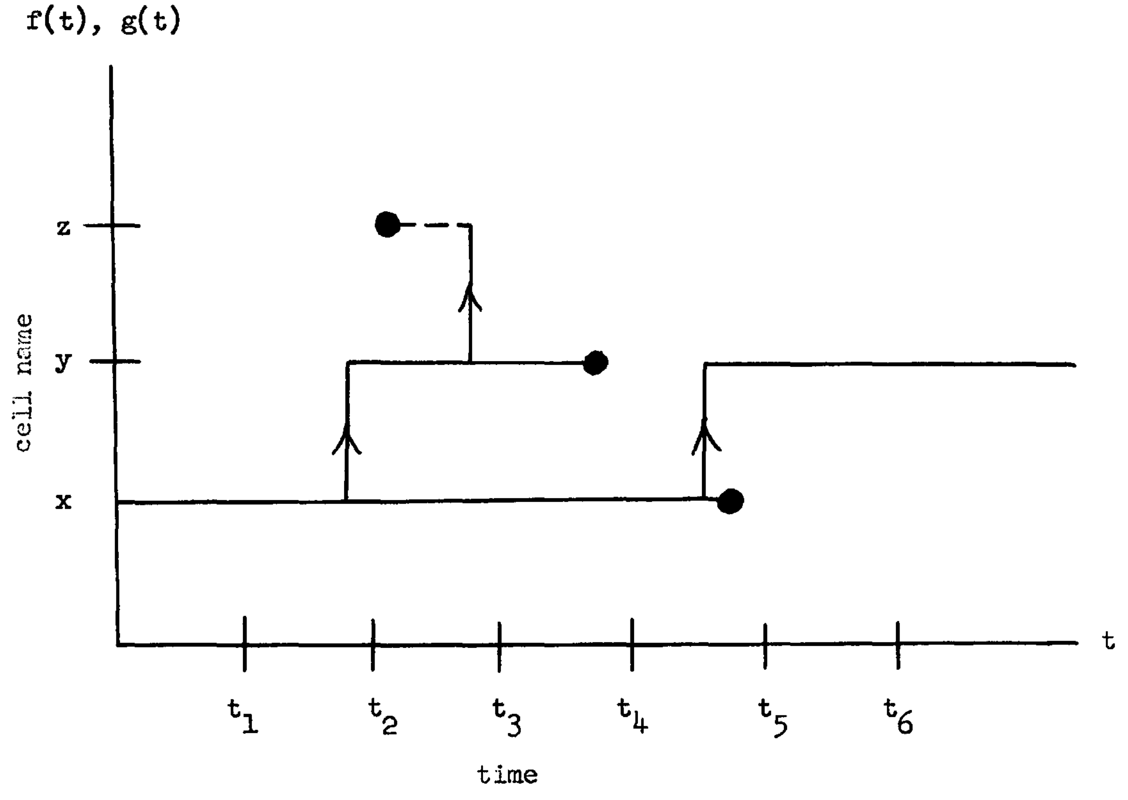


Figure 2.5. Locus of counts for a cell i in a variation of the example of Figure 2.4. The multiple-valued function f(·), defined by solid lines, and the arrows and dark circles are as described for Figure 2.4. The function g(·), defined by a dashed line, is such that g(t) = x if and only if the count at row x and column i of the count matrix equals -1 at time t.

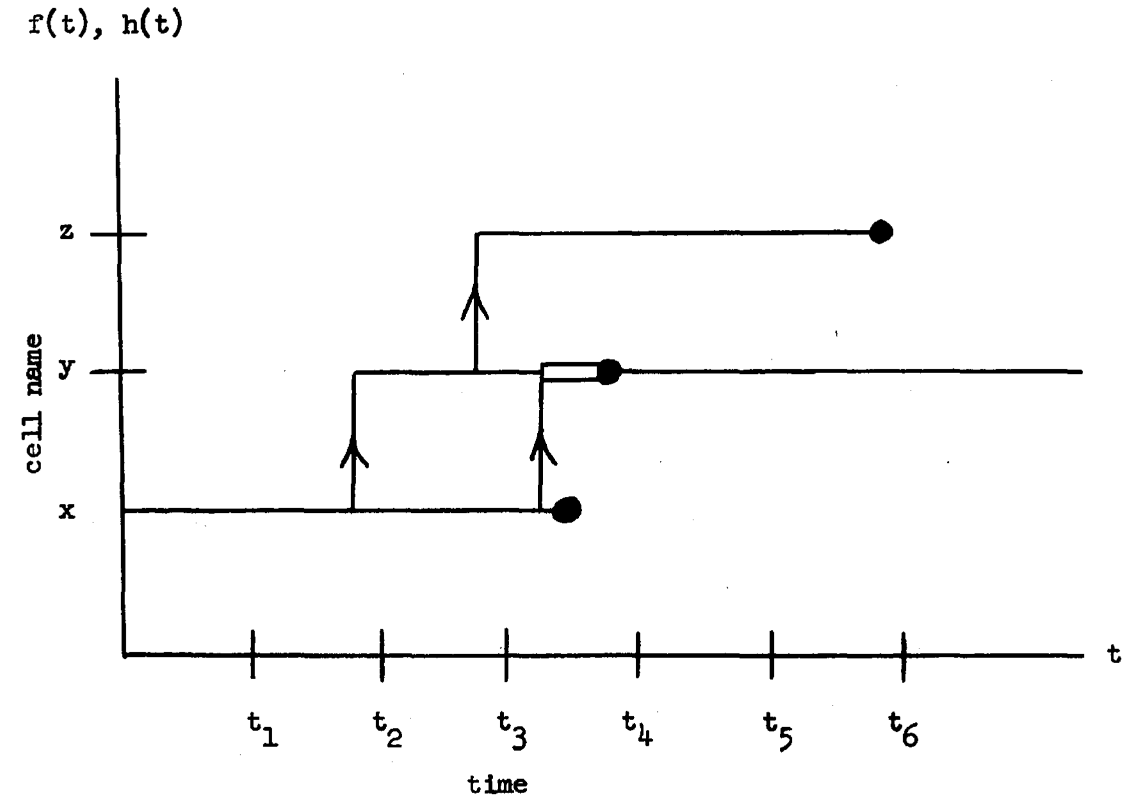


Figure 2.6. Locus of counts for a cell i in a variation of the example of Figure 2.4. The multiple-valued function f(·), defined by single solid lines, and the arrows and dark circles are as described for Figure 2.4. The function h(·), defined by a double solid line, is such that h(t) = x if and only if the count at row x and column i of the count matrix equals 2 at time t.

followed by a done of i, and the resulting configuration of the count matrix $k_b(\cdot, \cdot)$ prior to t_4 would have been such that

$$k_b(x, i) = 0$$
$$k_b(y, i) = 2$$
$$k_b(z, i) = 1$$

This variation of the example shows that the possibility of counts being greater than one allows a clerk cell to anticipate, in effect, a slower clerk cell's need for a read capability.

Tabulation of MCM Properties

Given here is a summary concerning three aspects of the MCM design: (1) the properties of the transactions, (2) the enabling rules, and (3) certain items of nomenclature. The summary is intended both to help the reader coalesce in his mind the diverse facts about MCM's, and to provide the reader with a tabulation of these facts for convenient reference during the reading of the Chapters to follow.

Suppose an MCM, whose set of names is N, holds a computation state given by the ordered pair $\langle c(\cdot), k(\cdot, \cdot)\rangle$, where $c(\cdot)$ is a content function describing the state of the MCM's collection of cells, and where $k(\cdot, \cdot)$ describes the state of the MCM's count matrix, i.e., where $k(x, i)$ is the count at row x and column i of the count matrix. Now suppose a computation state transition occurs in which only cell x receives a go pulse. Let the computation state prevailing after the transition be given in a similar way by the pair $\langle c'(\cdot), k'(\cdot, \cdot)\rangle$.

If during the transition cell x performed

get of i replace $f(\cdot)$

then

$c'(x) = f(c(i))$

$c'(a) = c(a)$; $a \neq x$

$k'(a, b) = k(a, b)$; a, b belonging to N

If during the transition cell x performed

put of i with v replace w

then

$c'(i) = v$; $i \neq x$

$c'(x) = w$

$c'(a) = c(a)$; $a \neq i, a \neq x$

$k'(a, b) = k(a, b)$; a, b belonging to N

If during the transition cell x performed

send of i to e replace w

then*

$c'(x) = w$

$c'(a) = c(a)$; $a \neq x$

$k'(e, i) = k(e, i) + 1$

$k'(a, b) = k(a, b)$; $\langle a, b \rangle \neq \langle e, i \rangle$

*The ordered n-tuple $\langle a_1, a_2, \ldots, a_n \rangle$ is equal to the ordered m-tuple $\langle b_1, b_2, \ldots, b_m \rangle$ if and only if $n = m$, and $a_1 = b_1$, and $a_2 = b_2$, and ..., and $a_n = b_n$. Thus, for example, $\{a, b\} = \{b, a\}$ always, but $\langle a, b \rangle = \langle b, a \rangle$ only if $a = b$.

If during the transition cell x performed

    done of i replace w

then

$$\begin{aligned}
&c'(x) = w \\
&c'(a) = c(a) && ;\ a \neq x \\
&k'(x, i) = k(x, i) - 1 \\
&k'(a, b) = k(a, b) && ;\ \langle a, b\rangle \neq \langle x, i\rangle
\end{aligned}$$

If during the transition cell x performed

    bye to e replace w

then

$$\begin{aligned}
&c'(x) = w \\
&c'(a) = c(a) && ;\ a \neq x \\
&k'(e, x) = k(e, x) + 1 && ;\ e \neq x \\
&k'(x, x) = k(x, x) - 1 && ;\ e \neq x \\
&k'(x, x) = k(x, x) && ;\ e = x \\
&k'(a, b) = k(a, b) && ;\ \langle a, b\rangle \neq \langle e, x\rangle \text{ and} \\
& && \ \langle a, b\rangle \neq \langle x, x\rangle
\end{aligned}$$

Just before the transition described above, cell x is an enabled clerk cell. The capabilities that a cell x must have in order to be an enabled clerk cell depend on the transaction that cell x would perform upon receipt of a go pulse. This dependence is given by five enabling rules, which are tabulated in Figure 2.7.

If an MCM holds the computation state $\langle c(\cdot), k(\cdot, \cdot)\rangle$, then cell x has read capability for cell i just when $k(x, i) > 0$. A cell x has write capability for cell i just when cell x is the only cell to have read capability for cell i.

Transaction cell x would perform upon receipt of a go pulse	Capabilities cell x must have in order to be an enabled clerk cell	
	Read	Write
get of i replace f(·)	i	x
send of i to e replace w	i	x
put of i with v replace w		x, i
done of i replace w		x
bye to e replace w		x

Figure 2.7. The enabling rules.

A cell that has write capability for itself is called a clerk cell. A cell that is not a clerk cell is called a value cell. A clerk cell that is not an enabled clerk cell is called a disabled clerk cell. Some cells are permanently designated to be output cells.

In the transactions listed in Figure 2.7,

i is called an operand name

e is called a sendee name

v is called an operand word

w is called a replacement word

$f(\cdot)$ is called a replacement function

Chapter III

A Facility That Behaves Like an MCM

Purpose

This Chapter discusses the feasibility of constructing and programming a computing facility that behaves toward a user as if it were an actually constructed MCM. The techniques and examples presented in the Chapter suggest that such a facility can be useful, and can be both constructed and programmed at a cost having the same order of magnitude as the cost of constructing and programming a contemporary facility for multiprocessing. The Chapter does not provide complete specifications for a facility that behaves like an MCM, but instead treats only some major issues concerning such a facility. One of the Chapter's goals is to indicate that further study of MCM applications of this type would be justified.

A facility of the type discussed in this Chapter is an example of but one kind of MCM application. There are other ways of designing facilities that behave like MCM's, and moreover, there are also MCM applications that do not involve the design of facilities. An example of the latter kind of application is a compiler whose generated code coordinates processing units according to the rules by which the clerk cells of an MCM are coordinated. A user who requests execution only of programs translated by such a compiler will never observe a nonfunctionality or noncompletion lurking bug effect, even if the executing facility does not itself prevent these effects.

The foregoing example, which will not be discussed further, indicates the variety of MCM applications that might be discussed. The particular application that will be discussed in this Chapter has been chosen both to illustrate the basic mechanism by which an MCM coordinates processes, and to show exemplary solutions for some of the problems that can arise in MCM applications generally. Thus the Chapter not only describes a particular application, but concomitantly provides an inventory of techniques that would be useful for developing other applications.

Method

The Chapter discusses a specific exemplary facility whose external characteristics are those of an MCM. This exemplary facility, called EF for short, does not exist, but might serve as a guide for the construction of an actual facility. The facility EF is realized by a combination of hardware and supervisory software, in the same way that each of the few dozen facilities of a typical CTSS system [14] is realized.

In order to highlight the differences between EF and contemporary facilities, and in order to avoid discussing details that are easily filled in by applying contemporary techniques, the Chapter describes EF primarily from the user's or programmer's point of view. In particular, the characteristics which EF presents to a user are described as those of a virtual, or apparent, machine, called VM for short. That is, EF behaves toward the user as if it were an actual construction of VM, in the same way that each of the few dozen facilities of the CTSS system behaves toward a user as if it were an actual construction of a machine very similar to an IBM 7094.

One of the tasks of the Chapter is to verify that EF behaves like an appropriately chosen MCM. This task will be accomplished by showing that VM behaves like an MCM. Since by definition EF behaves like VM, the demonstration that VM behaves like an MCM verifies that EF behaves like an MCM, and therefore verifies, by way of the proofs given in Chapters IV and V, that both nonfunctionality and noncompletion lurking bug effects do not occur in EF.

In the verification that EF behaves like an MCM, the machine VM serves as a tutorial intermediary. The correspondence between EF and an MCM is not completely trivial, and it turns out to be much easier to understand the relationship between VM and an MCM than it is to understand directly the relationship between EF and an MCM.

A description of VM is not sufficient to show that EF is feasible. For this purpose several additional topics are discussed: (1) the manner in which EF might be programmed using a language similar to the Algol language [2], (2) the properties of the compiler used to translate programs written in this language, (3) the character of EF's physical units, and (4) the supervisory programs that EF's processing units occasionally execute in order to perform activities not explicitly "wired into" EF's hardware. The discussions of these topics suggest methods for dealing with some of the major issues that concern EF's design and use. It is hoped that the remaining problems of EF's design and use can be solved by applying contemporary techniques.

Dynamic Allocation

The relationship between the facility EF and EF's virtual machine VM may be better understood if one keeps in mind that dynamic allocation of physical units among objects of program reference is assumed to occur in EF. One of the reasons for describing the properties of EF in terms of an abstraction such as VM rather than in terms of EF's physical units is to avoid a discussion of the complex, but fairly well understood, methods for implementing dynamic allocation; such a discussion would tend to obscure the primary points being made in the Chapter. It will be seen, however, that for EF to be feasible using contemporary technology, dynamic allocation is a practical necessity.

As mentioned in Chapter I, dynamic allocation implies that as time progresses there might occur in EF several distinct correspondences between physical units and the parts of VM. In other words, the role of any particular part of VM might be played by a succession of physical units, and during occasional intervals the role of a VM part might not be played by any unit. On account of dynamic allocation activity, it cannot be said that the role of a particular VM part is always played by a certain physical unit; it can only be said that when the role of a particular VM part is played, it is always played by one of a class of physical units.

Overview of VM

The parts of VM may be thought of as objects of program reference, because a program for EF might refer directly to these parts. On the other hand, the parts of VM are not the only objects to which a program written for EF might refer: a program for EF might refer to objects whose relationship to the parts of VM is established by the way in which the program is compiled or interpreted. An analogous situation prevails in the CTSS system: a program for a CTSS facility either might be, say, a FAP [18] program that refers directly to the words in the facility's 7094-like virtual machine or might be, say, a LISP [22] program that refers to objects called S-expressions, which are related in a non-trivial way to the words in the facility's virtual machine.

The parts of VM are of five kinds: segments, clerks, input devices, output devices, and control matrix elements. Clerks, input devices, and output devices were discussed in Chapter I. A segment is an ordered set of quantities, each quantity being a value set quantity of the type discussed in Chapter I. Control matrix elements are unique to the present development, and so have no counterpart in Chapter I. Each of these five kinds of VM parts will be described later in detail.

A Viewpoint toward Secondary Storage References

In order to see the general nature of both EF and VM, it is desirable to draw a sharper analogy between EF and a CTSS facility [6]. The user of a CTSS facility sees a virtual machine similar to a 7094, i.e., a machine having 32,768 virtual memory registers and having a virtual CPU with accumulator, index registers, etc. It is not often recognized in this context, however, that the user of a CTSS facility actually sees a much larger virtual storage system than just 32,768 main memory registers: supervisory programs provide the user's computation with access to secondary storage units, such as disc and drum units.

A user's secondary storage information is organized into ordered sets of words, each set of words being called a file. From the point of view of the user, each file is identified, not by a physical location, but by a BCD name that consists of twelve alphanumeric characters. A user program refers to a file using the file's BCD name, together with one or more integers indicating the words of interest in the file.

References to files in CTSS are accomplished when the physical CPU executes a supervisory program. This program causes the CPU to perform activities such as associating the name of the file being referenced with the file's physical location, and such as keeping the file "open", an activity which involves, for example, remembering the file's name so that subsequent references can be made using a small integer tag as an abbreviation of the file's name.

A user's computation makes a reference to secondary storage by executing a particular instruction that "calls the supervisor". The system is designed so that a user's computation cannot change the supervisory program; therefore, as far as the user is concerned, the function performed by the supervisory program might as well be "wired into" the hardware of the physical CPU. For that matter, the user would be equally satisfied if the CPU responded to a secondary storage reference by executing a microprogram stored in the CPU itself. What actually happens is that such a "microprogram" is in fact executed; the "microprogram" is stored in a memory unit separate from the CPU, and is written in terms of a set of instructions almost identical to the set of instructions in terms of which a user program may be written.

The functions which the supervisory "microprogram" can cause to be performed for a user are extremely complex; nevertheless these functions are a property of the user's virtual machine. Thus, for example, the virtual machine of a CTSS facility has the ability to access a large body of information that is referred to by invariant BCD names. Because the supervisory "microprogram" can remember information between calls, not every reference to a file need involve the use of the file's full BCD name.

The files that CTSS stores on behalf of a user are just as much a part of the virtual machine of the user's facility as are the virtual core memory and CPU. The instructions that refer to files are instructions that invoke execution of the supervisory "microprogram". Those of these instructions that, when executed, read and write information from and into files are actually instructions that move information between files and virtual core memory, and are thus similar in function to the "move" instructions found in many contemporary computers. The fact that files are not accessible in any way other than by execution of "move" instructions is not a good reason for excluding files from a CTSS virtual machine, for otherwise, in deciding generally whether something should or should not be a part of a virtual machine, where would one draw the line between "accessibility" and "non-accessibility"?

Introduction to Segments

Just as files having BCD names are part of the virtual machine of a CTSS facility, so also are named files of information part of the virtual machine VM of the facility EF. To follow contemporary usage [4, 9], these files of information will, in VM, be called segments. Since in CTSS a file and a virtual core memory are both ordered sets of words, it is natural to merge the concept of a file and virtual core memory into a single concept, namely, that of a segment; this is exactly the course that has been followed in the design of the Multics system [4], and will be the course followed here with respect to EF. In other words, in VM there are not both files and a virtual core memory, but only segments; each segment is accessed using a full instruction set similar to the set of instructions used to access a virtual core memory.

Just as in a CTSS facility, and even more as in a Multics facility, EF uses a supervisory "microprogram" to help make references to segments be effective. One of the functions performed by processing units executing this supervisory program is the dynamic allocation of parts of segments among storage units, such as core memory, drum, and disc units. Not only does the supervisory program aid in the implementation of segments, but it also aids in the implementation of other VM parts; various properties of the supervisory program will be explained later as the need arises.

Segments

The first VM part to be explained is the segment [4, 9]. As mentioned above, a segment is similar in function both to a CTSS file and to a virtual core memory. A segment is an ordered set of words, each word being what a programmer of a contemporary facility might call a "memory word". That is, each word denotes information, usually a few dozen bits in amount, that might be held in a core memory register. The number of words in a segment is called the length of the segment.

Each segment of VM is designated by a distinct name, which is a string of BCD characters. Each word in a segment is, in turn, designated by a distinct address, which is a binary integer between zero and the length of the segment minus one, inclusive. With respect to VM as a whole, therefore, each word is designated by a concatenation of segment name and address; this concatenation constitutes the word name of the word.

When a processing unit makes a reference to a segment, say to read some word in the segment, EF might allow the reference to proceed, or might take some action, such as typing an error message, to indicate that the reference is invalid. Let us examine this latter occurrence more closely. When a reference is made to a word, the name of the word is employed, either directly or indirectly. When a facility responds to a reference by saying that the reference is invalid, it is saying, "There is nothing associated with this name," or to use a more contemporary locution, "The object having this name does not exist." When a word does not exist, then either the segment that would contain the word does not exist, or else the word is not an existing word of the segment, i.e., the word's address, which is never negative, is greater than or equal to the segment's length.

Segments may be brought into and out of existence, i.e., created and deleted, respectively. Likewise, segments may be lengthened and shortened; such lengthening and such shortening are also activities of creation and deletion, respectively. Creation and deletion activities in VM will be described later in detail.

In understanding what a segment is, it is useful to observe that segmentation is just a way of naming words. That is, words are not intrinsically grouped into segments, but belong to segments only because we choose to structure their names in a certain way. The advantage of giving words numerically consecutive names is that some of the most powerful data processing techniques, such as additive table look-up, involve computing with names, or with parts of names.

Although segmentation is just a naming technique, it is reasonable to refer to a segment as something distinct from its individual words. For example, by performing an appropriate procedure step, a computation might tell EF, "Segment ABC will not be read or written for a while, so you may allocate its words to secondary storage if you like." In the light of the above interpretation of segmentation as a naming technique, the foregoing message to EF can be interpreted as, "You may allocate to secondary storage any word whose segment name is ABC." Such a message is thus seen to have significance even if segment ABC exists but happens to be of zero length when the message is given: the significance of the message in these circumstances is that if ABC should ever be lengthened, the resulting words may be allocated to secondary storage. Notice that if the foregoing message is given at a time when segment ABC does not exist, as opposed to when it exists and is of zero length, then EF might indicate that the message is invalid by saying in effect, "I know of no such segment."

Clerks

The second VM part to be explained is the clerk. A clerk is that kind of VM part whose role is played by a processing unit; as was mentioned in Chapter I, a clerk is an object that passes from one procedure step of a program to the next, obeying the directions encountered at each step. The sequence of actions that a clerk performs is called a process. A multiprocess program is a program that directs the activities of several clerks. A program for EF may be either a single-process program or a multiprocess program.

An unusual feature of VM is that each clerk is designated by a distinct name. The name of a clerk, like the name of a segment, is a string of BCD characters.

Clerks can be created and deleted, just like segments can be created and deleted. Examples of procedure steps whose executions in a contemporary facility would create and delete clerks are procedure steps of the types *fork* and *quit*, respectively; these procedure step types were described near the end of Chapter I.

Input Devices

The third kind of VM part to be explained is the input device. An input device is that kind of VM part whose role is played by an input unit, such as a card reader unit, or typewriter keyboard unit. As an object of program reference, an input device is similar to what is called in a FORTRAN [19] program a "symbolic" input device. That is, a program refers consistently to, say, "card reader device number 4" in spite of the fact that the role of this device might be played on different occasions by distinct physical units.

For each computation, there is associated with every input device an <u>input stream</u>, i.e., a sequence of input symbols. For example, a computation's input stream from a card reader device is the information punched in those cards, or portions of cards, that are read during the computation. In the case of a typewriter keyboard device, a computation's input stream from the device is the sequence of characters that are typed on the keyboard and that are read during the computation.

During a computation, a reference is made to an input device for one or more of three distinct purposes: (1) to read from the device's input stream, (2) to either sense or change the state of the physical unit playing the role of the device, e.g., to ask,"Hopper empty?" or to say, "Lock keyboard," and (3) to refer to the device as a whole, as for example to say, "I am not going to use this device for a while, and so this device's physical unit may be allocated to play the role of some other device."

Just like a clerk, each input device is designated by a distinct name, which is a string of BCD characters. Also just like a clerk, an input device can be created or deleted by the execution of an appropriate procedure step.

Output Devices

The fourth kind of VM part to be explained is the output device. An output device is that kind of VM part whose role is played by an output unit, such as a printer unit. Like an input device, an output device is referred to during a computation without regard to the identity of the physical unit playing the role of the device.

For each computation, there is associated with every output device an <u>output stream</u>, i.e., a sequence of output symbols. For example, a computation's output stream from a printer device is the sequence of characters produced by the printer during the computation. As with an input device, a reference to an output device during a computation is made for one or more of three distinct purposes: (1) to write into the device's output stream, (2) to sense or change the state of the physical unit playing the role of the device, and (3) to refer to the device as a whole.

Like input devices, output devices are designated by distinct BCD names, and may be created and deleted by the execution of appropriate procedure steps.

Control Matrix Elements

The fifth kind of VM part to be explained is the <u>control matrix element</u>. The control matrix elements of VM form, not surprisingly, VM's <u>control matrix</u>, whose function in VM is similar to that of the count matrix in an MCM. As shown in Figure 3.1, each row or column coordinate of VM's control matrix is a BCD name that might be the

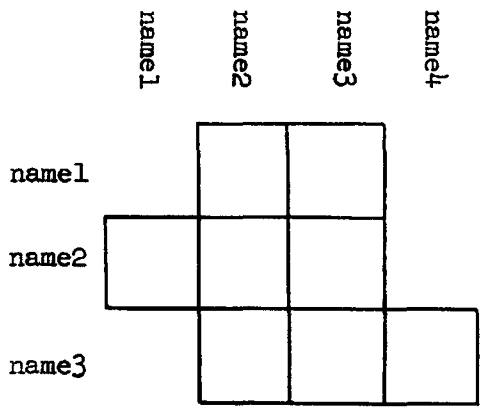


Figure 3.1. A typical control matrix. The names name1 through name4 are BCD names, each of which might be the name of a segment, clerk, input device, or output device.

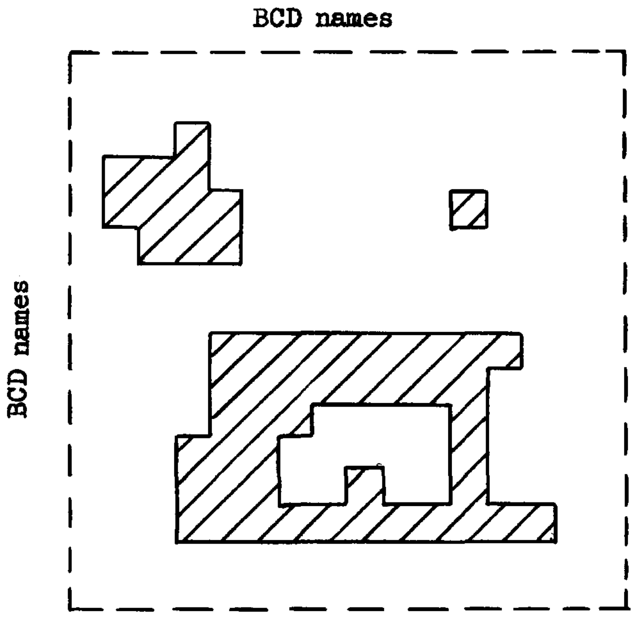


Figure 3.2. A typical set of existing control matrix elements. Dashed lines enclose a full control matrix. Hatched regions are a typical set of existing control matrix elements.

name of a segment, clerk, input device, or output device. Just like an element of the count matrix of an MCM, each element of VM's control matrix is an integer that might be positive or negative.

Unlike the count matrix of an MCM, the control matrix of VM does not necessarily have the usual rectangular shape of a matrix. Furthermore, the shape of VM's control matrix may vary during a computation. In order to understand the properties of VM's control matrix, let us imagine a <u>full</u> control matrix having a fixed, square shape and consisting of just one row and one column to correspond to each BCD name that might be the name of a segment, clerk, input device, or output device. Each element of the full control matrix might or might not exist as an element of VM's control matrix. Thus, as shown in Figure 3.2, VM's control matrix might have jagged edges, isolated sets of elements, and "holes".

If a control matrix element exists, and has the value n, then the element of the full control at the same position also has the value n. On the other hand, if a control matrix element does not exist, then the element of the full control matrix at the same position has a value that is given by a convention. This convention is established to facilitate the creation and deletion of VM parts, and will be explained later in the Chapter.

Control matrix elements are not created and deleted by clerks executing procedure steps that order their creation and deletion, but instead, control matrix elements are created and deleted automatically, perhaps by a processing unit obeying EF's supervisory program. Whenever a clerk makes a control matrix element have a value that does <u>not</u> conform to the convention for non-existing elements, then the control matrix element is created if it does not already exist, and is given the new value. Furthermore, whenever a clerk makes a control matrix element have a value that does conform to the convention for non-existing elements, then the control matrix element can be deleted. Thus the clerks of VM behave as if the full control matrix existed. For the sake of economy, only those full control matrix elements that do not conform to the convention for non-existing elements need be physically realized at any instant in the control matrix of VM.

A Sketch of VM's Operation

A sketch of the basic way in which VM operates when it performs a computation can now be presented. This sketch is accurate, but not complete; the details omitted will be filled in throughout the rest of the Chapter.

When VM performs a computation, each clerk of VM performs a process that is a sequence of actions. At any instant, a clerk does or does not have permission to proceed with its next action, depending on whether or not it has at that instant all of the capabilities necessary to perform the action.

In performing an action, a clerk always writes into itself. A clerk x has capability to write into itself just when the integer in row x and column x of the full control matrix is greater than zero and the other elements in column x of the full control matrix are less than or equal to zero. Unless mentioned otherwise, it may be assumed that any clerk has write capability for itself.

A clerk x has capability to read any word in a segment n just when the integer at position $\langle x, n\rangle$ of the full control matrix, i.e., in row x and column n of the full control matrix, is greater than zero. A clerk has capability to write any word in a segment just when it is the only clerk that has capability to read a word in the segment.

The action of a clerk might change a control matrix element. Let e be a clerk name, and let n be a segment name. A clerk x has capability to add 1 to the integer at position $\langle e, n\rangle$ of the full control matrix just when clerk x has capability to read any word of segment n, i.e., just when the integer at position $\langle x, n\rangle$ of the full control matrix is greater than 0. On the other hand, a clerk x always has capability to subtract 1 from the integer at position $\langle x, n\rangle$ of the full control matrix.

The above mechanism for changing the control matrix allows capabilities to read and write segments to be passed from one clerk to another, and to be relinquished by clerks when no longer needed; this mechanism is similar to that by which processes are coordinated in an MCM.

Coordination Procedure Steps

Before an explanation is given of how VM behaves like an MCM, an example will be presented to show how EF might be programmed using an Algol-like language. In preparation for this example, procedure steps of the types **send** and **done** are now introduced.

Before defining procedure steps of the type **send**, two examples of the use of this type of step will be given. First, suppose a programmer wishes to direct a clerk to add 1 to the integer at position $\langle e, n \rangle$ of the full control matrix. Then the programmer may write

send 'n', 'e';

For a second example, suppose again that a programmer wishes to direct a clerk to add 1 to the integer at position $\langle e, n \rangle$ of the full control matrix, and suppose that the values of the quantities a and b are the names n and e, respectively. Then the programmer may write

send a, b;

In general, if the expressions α and β evaluate to the names n and e, respectively, then the execution of

send α, β;

adds 1 to the integer at position $\langle e, n \rangle$ of the full control matrix. According to the rule given in the preceding Section, a clerk x has permission to add 1 to the integer at position $\langle e, n \rangle$ of the full control matrix just when the integer at position $\langle x, n \rangle$ of the full control matrix is greater than 0.

The single quotation marks used in the first example above are the first operators to be introduced for use in expressions whose values are names. Quotation marks inhibit evaluation of the expression they enclose; that is, the value of the expression

'α'

is the expression α itself. Additional operators for use in expressions whose values are names will be introduced later.

If the expression α evaluates to the name n, then the execution of

$$\underline{\underline{\text{done}}}\ \alpha;$$

by clerk x subtracts 1 from the integer at position $\langle x, n \rangle$ of the full control matrix. According to the rule given in the preceding Section, a clerk x always has permission to subtract 1 from any integer in row x of the full control matrix.

An Example -- Matrix Manipulation Again

The procedure steps just introduced will now be used in a program that directs the same computational activity as was directed by the program given at the end of Chapter I. This computational activity is the computing of

$$(AB)^{-1}((CD)(AB))$$

where A, B, C, and D are n by n matrices and AB is non-singular. Like the program of Chapter I, the program given here is written in a language similar to the Algol language, and uses four temporary storage matrices: T, U, Y, and Z. The program also uses the procedures "matrix multiply" and "matrix inverse of". Each of the matrices A, B, C, D, T, U, Y, and Z is stored in a distinct segment having the corresponding name.

The coding that would correspond to the fork appearing in the program of Chapter I is omitted, because the notion of fork in EF has not yet been discussed. Instead, the program is given in two compound statements: one directing the activity of a clerk named "alpha", and the other directing the activity of a clerk named "beta". The clerks may enter their respective compound statements simultaneously, or in arbitrary order.

Just before the entry into a compound statement by the first clerk to enter its compound statement, the control matrix is assumed to have the configuration depicted partially in Figure 3.3. It may be assumed that unless Figure 3.3 indicates otherwise, each element in columns A through beta of the full control matrix is 0. Thus, for example, Figure 3.3 indicates that clerk beta has write capability for segment Y. Figure 3.3 does not show the entire control matrix: clerks alpha and beta are assumed to have read capability for the segments into which the procedure steps of the program are encoded, and each clerk is assumed to have write capability for one or more temporary storage segments.

The compound statement entered by clerk alpha is the following.

```
begin   send 'C', 'beta';
        send 'D', 'beta';
        matrix multiply (A) times:(B) into:(T);
        send 'T', 'beta';
        matrix inverse of (T) into:(U);
        matrix multiply (U) times:(Z) into:(answer);
        done 'Z';
end
```

	A	B	C	D	T	U	Y	Z	alpha	beta
alpha	1	1	1	1	1	1	0	0	1	0
beta	0	0	0	0	0	0	1	1	0	1

Figure 3.3. Part of the control matrix upon entry into an example program.

The compound statement entered by clerk beta is the following.

```
begin   matrix multiply (C) times:(D) into:(Y);
        done 'C';
        done 'D';
        matrix multiply (Y) times:(T) into:(Z);
        done 'T';
        send 'Z', 'alpha';
end
```

A clerk enters its compound statement by passing to the symbol begin. The following activity ensues when exactly two clerks, named alpha and beta, enter their respective compound statements. Clerk beta might have to wait to be given read capability for C and D. Then alpha and beta might compute T and Y simultaneously*. Next beta might have to wait to be given read capability for T. Then alpha and beta might compute U and Z simultaneously. Finally, alpha might have to wait to be given read capability for Z in order to compute the final answer. After both clerks exit from their respective compound statements, the control matrix will be the same as it was before the clerks entered the compound statements.

*As during the execution of the program of Chapter I, the simultaneous execution by both clerks of the procedure "matrix multiply" occurs as if each clerk executed a separate copy of the procedure.

Introduction to the Correspondence between VM and an MCM

The purpose of the next several Sections is to show that VM behaves like an MCM. This will be accomplished by establishing a correspondence between the parts of VM and the components of a certain MCM. In order to establish this correspondence, additional details concerning VM's operation will be introduced as necessary. The correspondence itself, together with the description of MCM operations given in Chapter II, will serve to define VM precisely.

The particular MCM to which VM will be shown to correspond is designated by the letter M. The machine M, like any MCM, is described by four quantities: (1) a set of cell names, (2) a set of output cell names that is a subset of the set of cell names, (3) a transaction table for each cell, and (4) a set of initial computation states. The machine M is an abstraction employed to show, via the proofs of Chapters IV and V, that EF has certain properties, namely, that EF prevents nonfunctionality and noncompletion lurking bugs.

Consider the set $\mathcal{D}$ of the BCD names by which the segments, clerks, input devices, and output devices of VM can be designated. The set $\mathcal{D}$ contains all the names that might designate such VM parts; at any instant only a few of the names belonging to $\mathcal{D}$ might designate parts that actually exist. The names belonging to $\mathcal{D}$ are not partitioned according to the kind of VM part that they designate; on the contrary each name belonging to $\mathcal{D}$ can, on separate occasions, be the name of a segment, clerk, input device, or output device.

Corresponding to each name in $\mathcal{V}$ there is a <u>cluster</u> of cells in M. Every cell of M belongs to exactly one cluster. If n is a name belonging to $\mathcal{V}$, then the cells in the cluster corresponding to n have the names

$n.\mu$

$n.\alpha$

$n.\sigma$

$n.\pi\pi$

$n.\pi\sigma$

$n.0$

$n.1$

$n.2$

.

.

The symbols μ, α, σ, $\pi\pi$, and $\pi\sigma$ stand for "mode", "output", "state", "primary pointer", and "state pointer", respectively. The numerals 0, 1, 2, ... are addresses.

The number of cells in a cluster is finite, and is the same for all clusters. The number of cells in M is the number of cells in a cluster times the number of names in $\mathcal{V}$. Notice that the number of cells in M is fixed, even though the structure of VM changes from time to time as the result of creation and deletion activity.

As was mentioned, each name in $\mathcal{V}$, at any instant, is either used as the name of an existing segment, clerk, input device, or output device, or is not used as the name of any existing object. The usage of the cells in a cluster depends on which of these five conditions prevails for the name to which the cluster corresponds. The usage of the cells in a cluster will now be explained for each of these five conditions.

<u>The Cluster Corresponding to a Segment</u>

If n is the name of a segment of length, say, 3, then the cluster corresponding to n is shown in Figure 3.4. Cell $n.\mu$ always holds a word that denotes the "mode" of the cluster n. In Figure 3.4 the cell $n.\mu$ holds the symbol s, denoting "segment". When cluster n is a segment cluster, cells $n.\alpha$, $n.\sigma$, $n.\pi\pi$, and $n.\pi\sigma$ are not used, and so hold a word that is the symbol ∅, denoting "empty" or "undefined".

To each address with which a word is associated in a segment there corresponds in the segment's cluster a cell having the address and holding the associated word. For example, in Figure 3.4 the word having address 1 in segment n is the word x, and in cluster n the cell named n.1 holds the word x. Moreover, to each address, between 0 and the length of a segment minus 1 inclusive, with which <u>no</u> word is associated in a segment there corresponds in the segment's cluster a cell having the address and holding the word ∅. For example, in Figure 3.4 there is no word associated with address 3 in segment n, and in cluster n the cell named n.3 holds the word ∅. It is assumed that ∅ is not a word that can belong to a segment.

<u>The Cluster Corresponding to a Clerk</u>

If n is the name of a clerk, then every cell of the cluster corresponding to n holds the word ∅, except cells $n.\mu$ and $n.\sigma$.

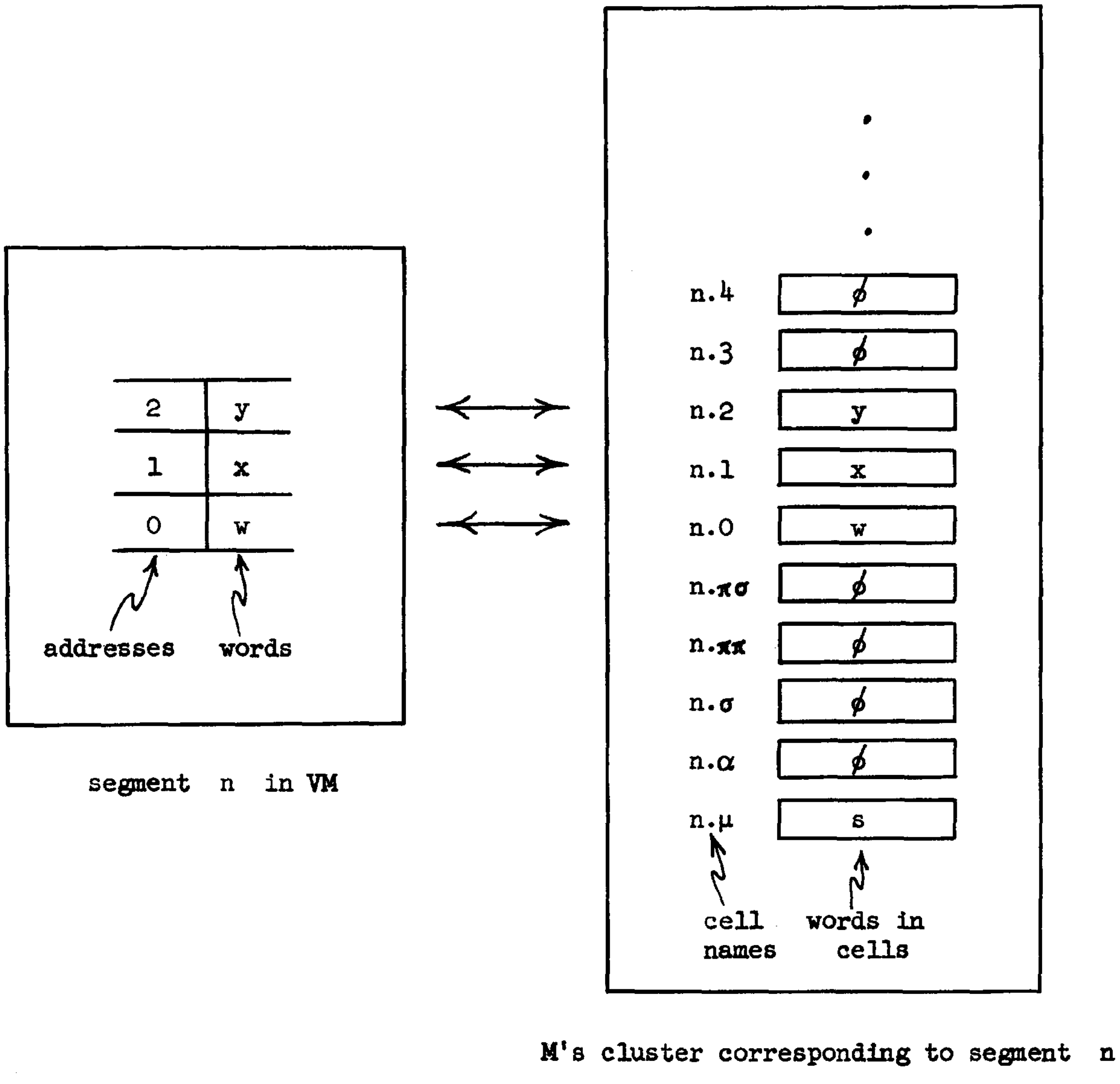


Figure 3.4. The cluster corresponding to a segment.

Cell $n.\mu$ holds the word c, denoting "clerk", and cell $n.\sigma$ holds the word that is the state word of clerk n in VM.

Performance Correspondences

Before the configuration of a cluster corresponding to an input device, output device, or unused name is described, an explanation will be given concerning the correspondence between the actions performed by clerks in VM and the transactions performed by clerk cells in M. When clerk x performs an action in VM, the corresponding activity in M is the performance of one or more transactions by clerk cell $x.\sigma$. For example, when clerk x reads the word at address 6 in segment i, clerk cell $x.\sigma$ performs a get of $i.6$.

The way in which the execution of a procedure step in VM corresponds to the performance of a sequence of transactions in M may be understood by means of the following example. Suppose that in VM the procedure step

Add one to segment i, address 6

is encoded into the word at address 14 in segment xp, in a code appropriate for clerk x. The execution of this procedure step by clerk x corresponds in M to the performance by clerk cell $x.\sigma$ of the following sequence of transactions.

$$\text{get of } xp.14 \text{ replace } f_1(\cdot) \tag{3.1}$$

$$\text{get of } i.6 \text{ replace } f_2(\cdot) \tag{3.2}$$

$$\text{put of } i.6 \text{ with } v \text{ replace } w \tag{3.3}$$

If the content of cell $xp.14$ just before clerk cell $x.\sigma$ performs (3.1) is $c_1(xp.14)$, then $f_1(\cdot)$ is such that $f_1(c_1(xp.14))$ is a word to which (3.2) corresponds in clerk cell $x.\sigma$'s transaction table. Similarly, if the content of cell $i.6$ just before clerk cell $x.\sigma$ performs (3.2) is $c_2(i.6)$, then $f_2(\cdot)$ is such that $f_2(c_2(i.6))$ is a word to which (3.3) corresponds in clerk cell $x.\sigma$'s transaction table. In (3.3), the word v equals $c_2(i.6) + 1$, and the word w is a word to which some transaction θ corresponds in clerk cell $x.\sigma$'s transaction table. This transaction θ, which might be, for example, a get of $xp.15$, is such that when clerk cell $x.\sigma$ performs θ, clerk cell $x.\sigma$ will read the next encoded procedure step word.

From the above example is may be seen how the step-by-step execution of procedure steps by a clerk in VM corresponds to the performance of a sequence of transactions by a clerk cell in M. This correspondence is achieved by an appropriate specification of the transaction tables of M. Thus the transaction tables of M are a description of the sets of executable procedure steps, or "instruction sets", of the clerks of VM.

The Cluster Corresponding to an Input Device

As was mentioned previously, a reference to an input device is made for one or more of three purposes: (1) to read from the device's input stream, (2) to sense or change the state of the device, and (3) to refer to the device as a whole. A reference to an input device as a whole occurs in a communication between clerks, or in a communication between a clerk and a processing unit that is executing a supervisory program. Such a communication might say, for example, "Obtain your next input from device n," or "Device n is no longer needed." A reference to an input device as a whole always speaks about the device and its properties, and never actually manipulates the device. A reference to a device as a whole may occur regardless of whether the device exists, and regardless of its actual properties. Since such references are not necessarily related to the properties of input devices, then the occurrence of such references tells us nothing about the properties of input devices, and so the possibility of such references may be ignored in establishing a correspondence between input devices and clusters of cells. Thus, in establishing a correspondence between an input device and a cluster of cells, only two types of reference need be accounted for: (1) reading from the device's input stream, and (2) sensing or changing the state, or configuration, of the device. These same considerations apply as well to the establishing of a correspondence between an output device and a cluster of cells.

The design of an MCM is such that the content of a cell, once set initially, can be changed only by a clerk cell writing into the cell, and not by external influences. Any external influences that affect the performance of a computation must affect the computation through information encoded into the initial computation state. Thus an input device's entire input stream, consisting of symbols on, say, punched cards, must be encoded into the initial computation state. This encoding is accomplished as follows.

Let n be the name of an input device. The first symbol in the input stream of n is held in cell n.0, the second symbol is held in cell n.2, the third symbol is held in cell n.4, etc. (The usage of the cells having odd addresses will be explained later.) Any even-addressed cells in cluster n that are not needed to hold input stream symbols for device n hold the word ϕ. Since there is always a practical limit on the number of symbols in an input stream, then the finite number of cells in a cluster can be chosen so that there is no chance for the number of symbols in an input stream to exceed the number of available cells. Since M is an abstraction that does not have to be actually constructed, then the number of cells in a cluster can be arbitrarily large. A number of cells in a cluster equal to

$$10^{(10^{10})}$$

should be sufficient to assure that any input stream found in practice can be encoded into the cells of a cluster.

For the input device n, the cell $n.\mu$ holds the word i, denoting "input device". Also, the cell $n.\alpha$ is not used and so holds the word ϕ. Finally, the cell $n.\pi\pi$ holds a non-negative even integer giving the address of the cell that holds the next symbol to be read from the input stream during a computation.

The way in which the reading of a symbol from the input stream of an input device by a clerk of VM corresponds to the performance of a sequence of transactions by a clerk cell of M may be understood by means of the following example. When clerk x of VM reads from the input stream of input device n, the corresponding sequence of transactions performed in M by clerk cell $x.\sigma$ is the following. First, clerk cell $x.\sigma$ performs a get of $n.\pi\pi$ to discover the address, say 4, of the cell to be read. Then clerk cell $x.\sigma$ performs a put of $n.\pi\pi$ to place into cell $n.\pi\pi$ an address that is always two greater than the previous address held in that cell, and that is in this case the address 6. Then clerk cell $x.\sigma$ performs a get of $n.4$ to obtain, in this case, the third symbol in the input stream of input device n. The fact that a sequence of transactions of the foregoing kind is always performed to correspond to the reading of an input stream is a property of M's transaction tables.

The state or configuration of an input device is distinct from the input stream of the device. In general, the state of an input device might be affected either by the action of clerks in VM, or by external influences. For example, a clerk can issue an order to lock a typewriter keyboard, or an operator can make a card reader "ready" by placing a deck of cards into a hopper. The giving of an order by a clerk of VM to change the state of input device n corresponds in M to the writing of a word into the cell $n.\sigma$. The words written into cell $n.\sigma$ express in coded form the particular orders appropriate to the device n.

When a clerk senses the state of an input device, the clerk is, in effect, reading a symbol from an input stream, because the state of the input device may, in general, be affected by influences external to VM. This <u>state</u> input stream of an input device is distinct from the previously discussed <u>primary</u> input stream of an input device. The successive symbols in a state input stream denote the successive device states that are sensed during a computation. Specifically, the i-th symbol in the state input stream of an input device is a symbol denoting the information sensed on the i-th occasion that the state of the input device is sensed, by any clerk, during a computation.

The state input stream of an input device is encoded into odd-addressed cells in the same way that the primary input stream of the device is encoded into even-addressed cells. For input device n, the first symbol in the state input stream is held in cell n.1, the second symbol is held in cell n.3, the third symbol is held in cell n.5, etc. Any odd-addressed cells in cluster n that are not needed to hold input symbols in the state input stream for device n hold the word $\emptyset$. The cell $n.\pi\sigma$ holds a non-negative odd integer giving the address of the cell that holds the next symbol to be read from the state input stream during a computation. A transaction sequence that reads from the state input stream of an input device is analogous to a transaction sequence that reads the primary input stream of an input device.

The treatment of a state input stream on a par with a primary input stream is unusual. The similarities between these two kinds of input streams are seldom emphasized, because in practice there is an important distinction between them: a user can completely specify a primary input stream, but usually cannot completely specify a state input stream. For present purposes, however, this distinction is not important; the successive input device states sensed during a computation must be considered an input stream to VM, because these states are, in general, affected by influences external to VM. The fact that we have had to consider a symbol sequence not completely specifiable by a user to be an input stream is a fact that will be discussed further in Chapter VI.

The description of a cluster corresponding to an input device is now complete. Let us review the roles of the cells in a cluster corresponding to an input device n. The primary input stream is held in the even-addressed cells, starting with cell n.0. The state input stream is held in the odd-addressed cells, starting with cell n.1. Even- or odd-addressed cells that are not used hold the word ϕ. The cells $n.\pi\pi$ and $n.\pi\sigma$ hold the addresses of the next cells in cluster n to be read from the primary and state input streams, respectively. The cell $n.\sigma$ holds a word that is an encoding of the most recent order for change of state issued by a clerk to device n. The cell $n.\alpha$ is not used, and so holds the word ϕ. The cell $n.\mu$ holds the word i, denoting "input device".

The Cluster Corresponding to an Output Device

The symbols that the clerks of VM place in the output stream of output device n are, in M, written into cell $n.\alpha$, which is designated in M as an output cell. An order given by one of the clerks of VM to change the state of output device n without affecting the output stream corresponds in M to the writing of a word into the cell $n.\sigma$.

Since a clerk of VM can sense the state of an output device, and since this state might be affected by external influences, therefore an output device has a state input stream similar to that of an input device. Just as for an input device, the symbols in the state input stream of the output device n are held in the cells n.1, n.3, etc., with any unneeded odd-numbered cells holding the word ϕ. The cell $n.\pi\sigma$ holds the address of the next cell in cluster n to be read from the state input stream during a computation. A transaction sequence that reads from the state input stream of an output device is analogous to a transaction sequence that reads from either input stream of an input device.

For the output device n, the even-addressed cells, and the cell $n.\pi\pi$ are not used, and so hold the word ϕ. The cell $n.\mu$ holds the word o, denoting "output device".

The Cluster Corresponding to an Unused Name

If the name n is not used in VM, then every cell in cluster n holds the word ϕ.

The Correspondence between Control and Count Matrices

At some instant during a VM computation, let $k_f(\cdot, \cdot)$ denote VM's full control matrix; that is, let $k_f(x, n)$ be the integer at position $\langle x, n \rangle$ of the full control matrix. Similarly, let $k_m(\cdot, \cdot)$ denote the corresponding count matrix of M. Recall that the value of an element of $k_f(\cdot, \cdot)$ is either the value of the corresponding element of VM's actual control matrix, or is dictated by a convention, to be described later, that gives the value of a non-existent control matrix element.

Figure 3.5 shows the count matrix corresponding to a 2 by 2 full control matrix. In general, the correspondence between $k_f(\cdot, \cdot)$ and $k_m(\cdot, \cdot)$ is given as follows. Let $k_m(x.a, n.b)$ be an arbitrary

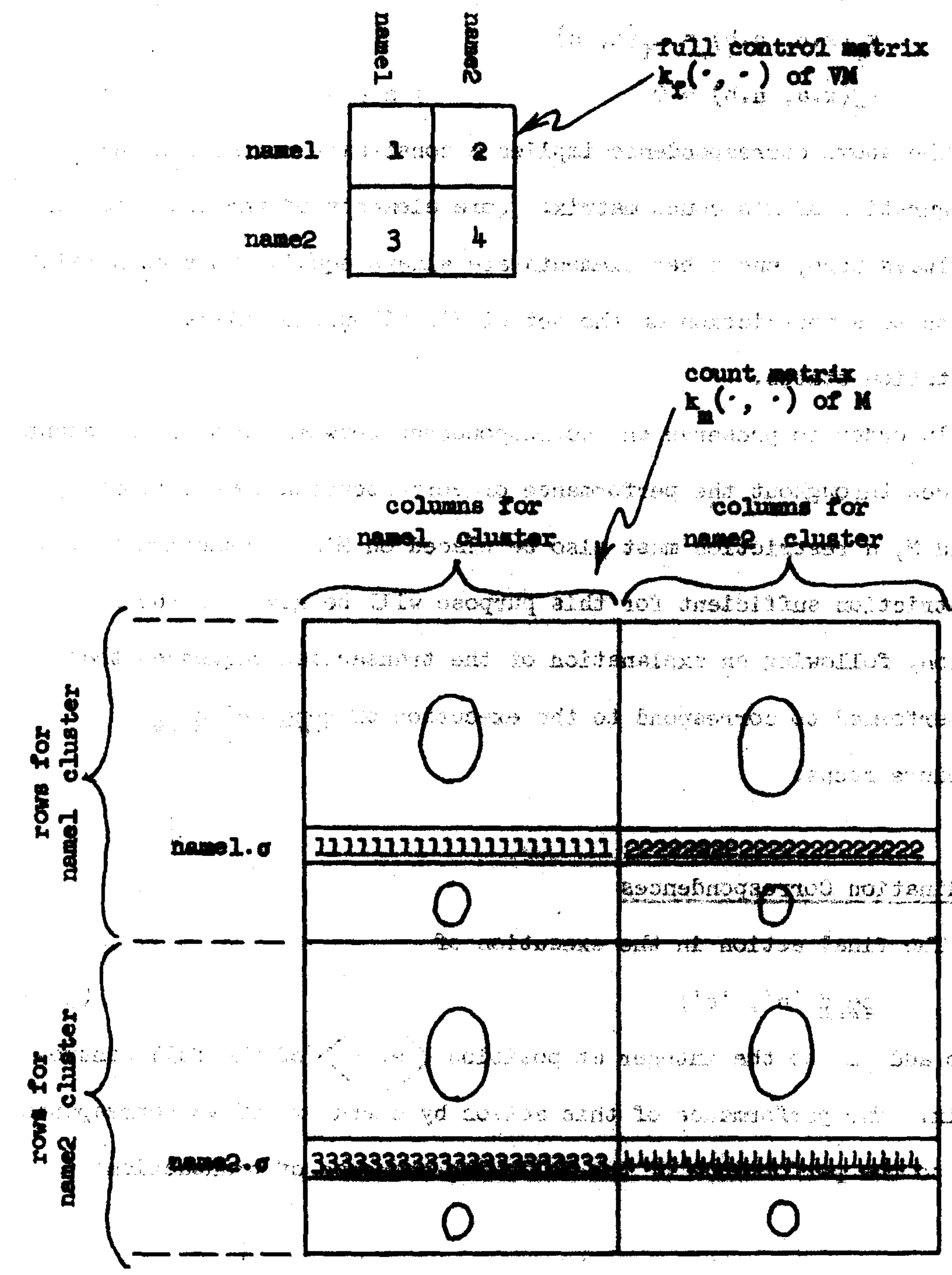


Figure 3.5. Correspondence between a full control matrix and a count matrix.

element of $k_m(\cdot, \cdot)$. Then

$$k_m(x.\sigma, n.b) = k_f(x, n)$$
$$k_m(x.a, n.b) = 0 \qquad ; a \neq \sigma$$

The above correspondence implies a constraint on the initial configuration of M's count matrix: some elements of the count matrix are always zero, and other elements are always equal. This constraint amounts to a restriction on the set of M's allowable initial computation states.

In order to preserve the correspondence between control and count matrices throughout the performance of corresponding computations by VM and M, a restriction must also be placed on M's transaction tables. A restriction sufficient for this purpose will be given in the next Section, following an explanation of the transaction sequences that are performed to correspond to the execution of send and done procedure steps.

Coordination Correspondences

The final action in the execution of

$$\text{send } 'n', 'e'; \tag{3.4}$$

is to add 1 to the integer at position $\langle e, n \rangle$ of the full control matrix. The performance of this action by clerk x of VM corresponds in M to the performance of the following sequence of transactions by clerk cell x.σ.

send of n.μ to e.σ replace w_1 (3.5)

send of n.α to e.σ replace w_2 (3.6)

send of n.σ to e.σ replace w_3 (3.7)

.

.

send of n.m to e.σ replace w_p (3.8)

In (3.5), w_1 is a word to which (3.6) corresponds in cell x.σ's transaction table. Likewise in (3.6), w_2 is a word to which (3.7) corresponds in the same transaction table. In (3.8), m is the maximum address of a cell in a cluster, and p is the number of cells in a cluster. One might say that in correspondence to the execution of (3.4), the whole cluster n is sent to cell e.σ.

Likewise, the final action performed by a clerk x executing

done 'n';

is to subtract 1 from the integer at position $\langle x, n \rangle$ of the full control matrix. The performance of this action corresponds in M to the performance of a sequence of dones by clerk cell x.σ, one done for each cell in cluster n.

The above correspondences for send's and done's suggest the following restriction on M's transaction tables. For each cell x, a send, done, or bye transaction may appear in cell x's transaction table only as part of a transaction sequence, such as (3.5)-(3.8), which when performed by cell x would make identical alterations to count matrix elements in one row and several columns, the row corresponding to a cell whose name ends in ".σ", and the columns corresponding just to the cells in one cluster. In conjunction with the timing correspondence discussed in the next Section, this restriction is sufficient to maintain during a computation a correspondence between VM's control matrix and M's count matrix.

The Timing Correspondence

The object of this Section is to describe how the strategy of M's scheduler can be established so as to maintain a correspondence between the behavior of VM and the behavior of M. As was mentioned in Chapter II, the strategy of M's scheduler is the scheme by which the scheduler makes selections from successive choice collections. It may be recalled that a choice collection is derived from a computation state by applying both the enabling rules and a rule to prevent clerk cells from altering the same count matrix element simultaneously. From out of the choice collection derived from a given computation state, the scheduler selects, in accordance with its strategy, one set of names to be the set of the names of the cells that will accomplish the next computation state transition.

It is assumed that each clerk in VM proceeds autonomously, and that the actions of several clerks may overlap in time in an arbitrary way. The autonomous behavior of clerks in VM may be contrasted with the synchronous behavior of clerk cells in M; in a computation performed by M, all transactions of one computation state transition are completed before the next computation state transition is begun. The main task in explaining the strategy of M's scheduler is to establish a method by which the autonomous clerk behavior of VM may be modeled by the synchronous clerk cell behavior of M.

The modeling of VM's behavior in terms of M's behavior is achieved by letting the performance of transactions in M occur both instantaneously, and immediately upon receipt of a go pulse. It is legitimate to talk about instantaneous transaction performances in M, since M is an abstraction that does not have to be actually constructed. An alternative to letting the performance of a transaction be instantaneous is to let the performance of a transaction have a duration much smaller than any duration of interest in VM.

The process that a clerk in VM performs may be divided into a sequence of actions, each of which has non-zero, finite duration, and each of which involves the sensing and/or changing of the state of another VM part. For example, the reading by a clerk x of the word at address 7 in segment n is such an action. As shown in Figure 3.6, this action has a certain duration for the word being read and a certain duration for clerk x. During the interval when the word at address 7 in segment n is in use for this action, no other clerk reads or writes the word. Likewise, during the interval when clerk x is in use for this action, no other clerk reads or writes the state word of clerk x.

In the example of Figure 3.6, the transaction that corresponds in M to the read action in VM is a transaction performed by clerk cell x.σ, and is a get of n.7. For convenience, let us imagine that corresponding computations occur in VM and M simultaneously. Then in the example of Figure 3.6, the transaction corresponding in M to the read action in VM is performed in M at an instant during the interval when in VM both clerk x and the word being read are in use for the read action.

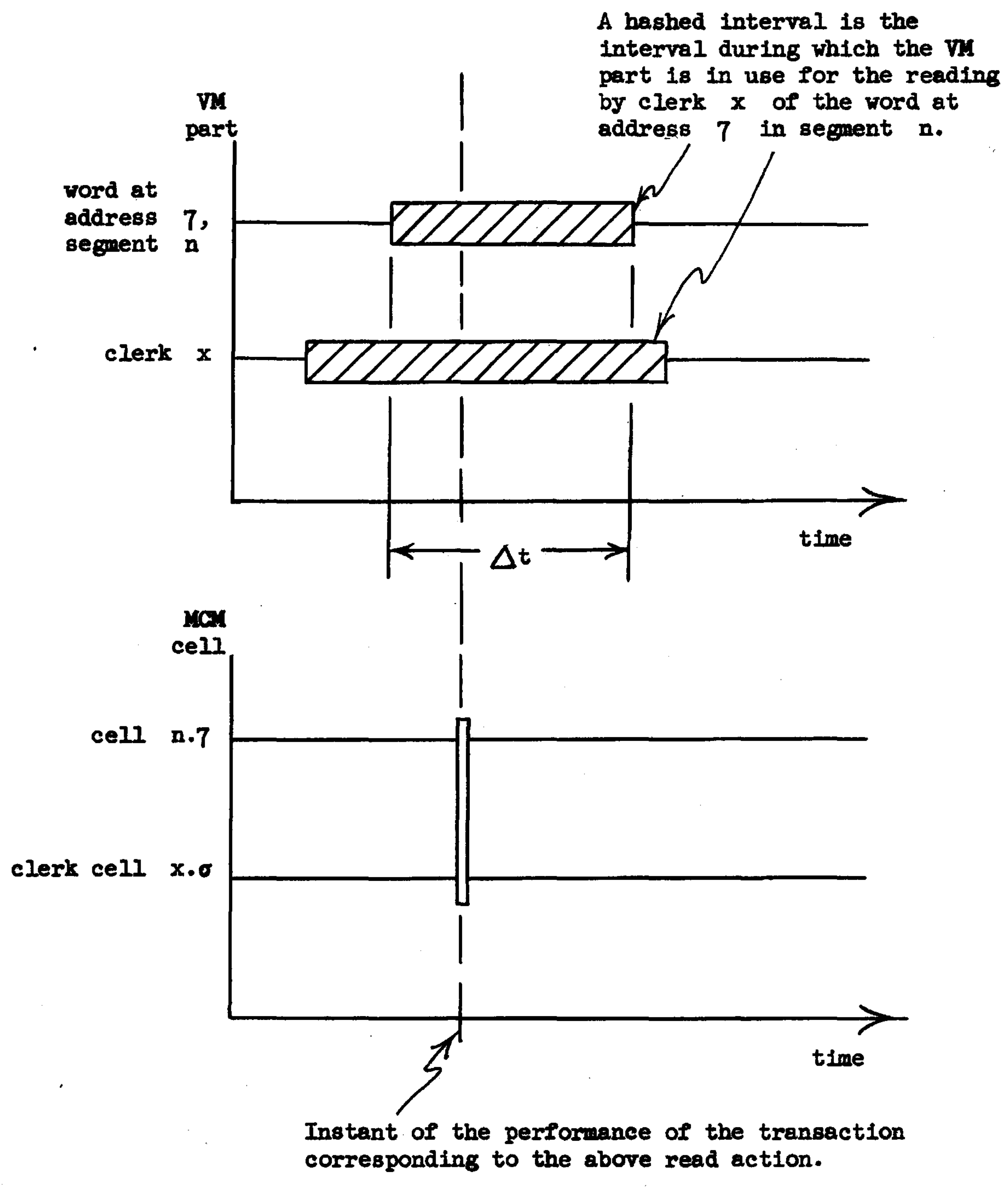

A hashed interval is the interval during which the VM part is in use for the reading by clerk x of the word at address 7 in segment n.
VM part
word at address 7, segment n
clerk x
time
Δt
MCM cell
cell n.7
clerk cell x.σ
time
Instant of the performance of the transaction corresponding to the above read action.

Figure 3.6. The performance in M of the transaction that corresponds to a read action in VM.

The action-transaction correspondence in the example of Figure 3.6 is in one respect not typical of action-transaction correspondences in general, because in this example the performance of the read action in VM corresponds to the performance of just one transaction in M. In general, the performance of an action by a clerk in VM corresponds to the performance of one or more transactions by a clerk cell in M. For example, to an action that reads from an input stream in VM there corresponds in M the performance of three transactions, two of which manipulate an address pointer. For another example, to an action that adds one to a control matrix element of VM there corresponds in M the performance of a number of send transactions equal to the number of cells in a cluster.

For some action performed by a clerk in VM, consider the interval Δt during which all parts of VM that participate in the action are in use for the performance of the action. The transaction or transactions, always finite in number, that correspond to this action are performed in M during the interval Δt. This timing correspondence holds for every action performed in VM. The scheduler of M makes selections from choice collections, and schedules the transmission of go pulses, so that this timing correspondence is maintained.

The rules by which clerks of VM obtain permission to proceed are constructed to correspond to the rules for the formation of choice collections in M in such a way that it is possible for M's scheduler to maintain the above timing correspondence. A sketch indicating the general nature of VM's permission rules was given previously. A complete list of the rules for the actions discussed so far is given in Figure 3.7. Additional permission rules will be given as new actions are introduced. Notice that each permission rule can be inferred from the correspondence between VM and M. For example a clerk, having write capability for itself, must have, not read capability, but write capability, for an input device in order to read from the input stream of the device. This is because the reading of an input stream corresponds in M to the performance of three transactions, one of which is a put that increments the pointer indicating the next symbol to be read.

A capability possessed by a clerk of VM can be taken away from the clerk only by an action performed by the clerk itself, specifically, by the clerk's decrementing of a control matrix integer. Thus, whenever a clerk has permission to perform an action, it continues to have that permission until the action is performed. It is assumed that once a clerk of VM has permission to perform an action, it does perform that action after a finite time. This assumption means that the strategy by which M's scheduler maintains the timing correspondence between VM and M is a strategy that is reasonable, in the sense described in Chapter II.

The timing correspondence between VM and M implies that there is not a simple correspondence between the state of VM at some instant and the computation state of M at the same instant. For example, M can have a definite computation state at an instant when in VM actions are still being completed. This lack of a simple correspondence between machine states seems to be a consequence of the modeling of autonomous clerk behavior by synchronous clerk cell behavior.

Clerk x has read capability for part n just when the integer at position $\langle x, n\rangle$ of the full control matrix is greater than zero. Clerk x has write capability for part n just when clerk x has sole read capability for part n.

Action clerk x will perform	Capabilities clerk x must have to perform the action.	
	Read	Write
read from a word in segment n	n	x
write into a word in segment n		x, n
read from the primary input stream of input device n		x, n
read from the state input stream of input device n		x, n
command a change of state of input device n		x, n
write into the output stream of output device n		x, n
read from the state input stream of output device n		x, n
command a change of state of output device n		x, n
add one to the integer at position $\langle e, n\rangle$ of the full control matrix	n	x
subtract one from the integer at position $\langle x, n\rangle$ of the full control matrix		x

Figure 3.7. Permission rules for several actions of a clerk in VM.

Creation and Deletion Correspondences

The creation of a VM part n corresponds in M to the performance of a sequence of puts that change cluster n from the cluster of an unused name into the cluster of the VM part being created. Likewise, the deletion of a VM part n corresponds in M to the performance of a sequence of puts that change cluster n from the cluster of the VM part being deleted into the cluster of an unused name.

The creation of a VM part corresponds in M to at least one writing of a non-ϕ word into a cell that previously held the word ϕ. Likewise, the deletion of a VM part corresponds in M to at least one writing of the word ϕ into a cell that previously held a non-ϕ word. Therefore, in order to have permission to perform an action that creates or deletes a VM part n, a clerk x of VM must have write capability for part n, as well as, of course, write capability for itself.

Epilogue for the Correspondence between VM and M

The transaction sequence correspondences for certain creation-deletion procedure steps to be introduced later have yet to be described. Except for the discussion of these correspondences, the explanation of the correspondence between VM and M is now complete. The machine M is constructed to correspond internally to VM. Therefore the external behavior of VM, i.e., the relationship between the initial state and output streams of VM, is identical to the external behavior of M. Thus after the transaction sequence correspondences mentioned above have been discussed, it will have been shown that there is an MCM which VM behaves like, and so the proofs of Chapters IV and V will verify that nonfunctionality and noncompletion lurking bugs do not occur in VM.

Attention now turns again to the way in which EF might be programmed using an Algol-like language. The specific concern of the next three Sections is the explanation of some procedure steps whose executions create and delete VM parts.

The Problem of Choosing Names

Although the previously mentioned permission rule for creation and deletion actions is sufficient to assure a correspondence between VM and M, it remains to be shown that it is easy to write useful VM programs that direct creation and deletion activity. The major problem in writing such programs concerns the choice of names to be given to VM parts being created.

From the correspondence between VM and M, the following statement about VM may be inferred: the name of a VM part being created is never chosen arbitrarily, in accordance with influences external to VM, at the instant of the part's creation. To understand why this statement is true, consider the transactions that correspond in M to a creation action performed by a clerk x in VM. These transactions are puts into some particular cluster, say cluster n. In M, the name n is determined by the word held in clerk cell $x.\sigma$ just before the creation action begins in VM. Therefore, in VM, the name n is determined from the corresponding state word of clerk x, i.e., from the state word of clerk x just before the creation action. Thus the name n is the only name that may be given to the VM part being created, because the choice of any other name would fail to preserve the correspondence between VM and M. Thus the selection of the name n

must not be made arbitrarily, but must be "programmed": the name n can either be specified explicitly in the program being executed, or be computed as part of the computational activity being performed.

When choosing a name for a VM part to be created, or when programming such a choice, a computation's planner, either a programmer or a compiler, must avoid two potential difficulties. First, the name of the part to be created might inadvertently be chosen so that the clerk executing the creation procedure step will never receive write capability for the name and hence never have permission to create the part. Second, the name of the part to be created might inadvertently be chosen to be the same as the name of a VM part that already exists and that will not be deleted before the part being created is required. It is desirable to have at hand a method for choosing names, or for programming the choice of names, so that these two difficulties can be systematically avoided. Such a method is described in the next Section.

Naming Conventions Suggested to Facilitate Creation and Deletion

This Section describes four programming and design conventions, the adoption of which will allow the two problems mentioned above to be circumvented in a practical manner. The conventions are interrelated, and accomplish their intended purpose only if all four are adopted. It may be noted, however, that adherence to neither any nor all of these conventions is required in order to maintain a correspondence between VM and an MCM; the conventions have been formulated merely to facilitate the programming of creation and deletion activity. The second of the four conventions is the previously promised convention that gives values for non-existing control matrix elements.

The first convention is the following: the name of each created VM part should contain as a largest explicitly delimited prefix the name of the creating clerk. For example, if clerk x were to create another clerk, the new clerk might be named

x:y

and if clerk x:y were to create a segment, the segment might be named

x:y:m

This convention allows the generation of arbitrarily long names. References using such long names can be made practical using the attachment scheme discussed by Dennis [9]. In this scheme a clerk has available to it a number of attachment tags. After a correspondence has been established between a tag and some portion of a name, subsequent references can be stated more compactly using the tag and the remainder of the name. During the performance of a computation, tag correspondences can be changed, saved, and restored in a manner similar to the manipulation of quantities in index registers. An example of an application of the attachment concept is the use in CTSS of a small integer tag to refer to a file that has been "opened" [6]. Two more examples of applications of the attachment concept are found in the Multics system: a segment number is a tag abbreviating a segment's tree name, and a base register number is a tag abbreviating a segment number [15]. Notice that a programmer working in an Algol-like language need not be aware that the attachment concept is being employed in the execution of a compiled program.

The second convention gives values for non-existing control matrix elements. At some instant during a VM computation, let $k_f(\cdot, \cdot)$ be the full control matrix, and let $k_a(\cdot, \cdot)$ be the actual control matrix. As mentioned previously, for each $\langle x, n \rangle$ such that $k_a(x, n)$ exists, then

$$k_f(x, n) = k_a(x, n)$$

The convention to be described now gives $k_f(x, n)$ for each $\langle x, n \rangle$ such that $k_a(x, n)$ does not exist. The explanation of this convention requires consideration of two separate cases.

The first case in defining $k_f(x, n)$ where $k_a(x, n)$ does not exist is the case in which n contains a colon. Let y be the longest explicitly delimited prefix in n. Then $n = y{:}m$, where m does not contain a colon. If $x = y$, then

$$k_f(y, y{:}m) = 1$$

and otherwise

$$k_f(x, y{:}m) = 0$$

Thus a VM name, considered as a potential clerk name, has write capability for any name having its own name as largest explicitly delimited prefix, unless VM's actual control matrix indicates otherwise.

The second case in defining $k_f(x, n)$ where $k_a(x, n)$ does not exist is the case in which n does not contain a colon. Here it is assumed that there is some master name, say p, such that if $x = p$ then

$$k_f(p, n) = 1$$

and otherwise

$$k_f(x, n) = 0$$

Thus the master name, which might be the name of a clerk not expected to be deleted, has write capability for any name that does not contain a colon, provided VM's actual control matrix does not indicate otherwise.

For stating the third and fourth conventions, let n be any name in VM, and let x be the name in VM that is given as follows: if n contains a colon, then x is the largest explicitly delimited prefix of n, otherwise, x is the master name.

The third naming convention is the following. Let n and x be as mentioned above, and let y be the name of a clerk such that $y \neq x$. When clerk y is programmed to delete a VM part having the name n, clerk y should be programmed to do the following two things just after the deletion: (1) add one to the integer at position $\langle x, n \rangle$ of the full control matrix, and (2) make the integer at position $\langle y, n \rangle$ of the full control matrix less than or equal to zero. A sequence of actions performed according to this third convention is sufficient to restore write capability for name n to the name x, provided that the fourth convention is adhered to.

The fourth convention is the following. Let n and x be as mentioned above. Whenever x is the name of a clerk, and clerk x is programmed to give write capability for n to some other clerk, then clerk x should be programmed to make the integer at position $\langle x, n \rangle$ of the full control matrix be exactly equal to zero. Furthermore, clerk x should be programmed to perform no more subtractions from this integer until clerk x itself has performed an action requiring read capability for name n.

The effects of these four conventions are best explained using an example. Suppose clerk x has not yet performed any action involving the name x:m, where m does not contain a colon, and suppose column x:m of the control matrix does not exist initially. Suppose clerk x now creates the segment x:m. Clerk x has permission to perform this creation action because clerk x is guaranteed to have write capability for the name x:m. Since clerk x has always had write capability for the name x:m, then it is certain that no other clerk has previously created a VM part having the name x:m. Notice that elements of column x:m of the control matrix either may be created at the time clerk x creates segment x:m, or may be created as they are needed.

Once segment x:m exists, capabilities for it may be passed among clerks. When segment x:m is deleted, write capability for name x:m will be passed back to clerk x, which may then create another VM part having the name x:m.

The four conventions imply that a clerk x which has write capability for itself always has permission to create a VM part with the name x:m, where m does not contain a colon, provided that the name x:m is currently unused. In order to guarantee that the name n of a part being created by a clerk x is currently unused, the planner of a computation need only arrange that the smallest explicitly delimited suffix in the name n is distinct from the suffixes of this sort in the names of the currently existing VM parts that were created, not by any clerk, but by clerk x itself. A clerk may be programmed to achieve this distinctness in a name's suffix through the use of a counter, or by any other means appropriate to the calculation being performed.

Creation and Deletion Procedure Steps

Four types of procedure steps will now be introduced. The execution of any one of these procedure steps either creates or deletes a VM part. These procedure steps make use of the naming conventions just discussed, and are formulated for convenient use in the programming example to be given in the next Section. In the following definitions of these procedure steps, let x be the name of the executing clerk.

The first type of procedure step is

create segment α;

The expression α must evaluate into a name that does not contain a colon, say the name m. Execution of this procedure step creates a segment named x:m. The segment is initially of zero length; the subsequent writing of words into the segment will cause its length to be increased as required.

The second type of procedure step is

fork α, β , e;

The expression α must evaluate into a name that does not contain a colon, say the name m. The expression β must evaluate into the name of a segment, say the name s. The string e must be a label. Execution of this procedure step creates a clerk named x:m. The state word of clerk x:m is made identical to that of clerk x, with two exceptions:

(1) clerk x:m is set of find its first procedure step at the label e, and (2) segment s is made the *private segment*, sometimes called the stack segment [4], of clerk x:m. The private segment of a clerk is a segment into which the clerk may store temporary results; a clerk normally retains write capability for its private segment. Following the creation of the clerk x:m, the clerk x, as part of the execution of the **fork** procedure step, acts as if it executed the following sequence of procedure steps.

```
        send 'x:m', 'x:m';
        done 'x:m';
        send 's', 'x:m';
        done 's';
        send 'r', 'x:m';
```

Here r is the name of the segment that contains the encoded **fork** procedure step which clerk x is executing. The execution of the **fork** procedure step might be sufficient to cause the clerk x:m to begin execution at the label e, even as clerk x simultaneously continues execution by passing to the procedure step following the **fork**.

The third type of procedure step is

```
        delete α;
```

The expression α must evaluate into a name, say the name n. If n contains a colon, then let the name p be the largest explicitly delimited prefix in n, and otherwise, let p be the master name. If $n \neq x$, then execution of the procedure step deletes the VM part n. Following this deletion, and as part of the execution of the **delete** procedure step, the clerk x acts as if it executed the following sequence of procedure steps.

```
        send 'n', 'p';
        done 'n';
```

If n = x, then the execution of the delete procedure step is equivalent to the execution of the procedure step quit, described below.

The fourth type of procedure step is

```
        quit;
```

If x contains a colon, let the name p be the largest explicitly delimited prefix in x, and otherwise, let p be the master name. In addition, let r be the name of the segment containing the encoded quit procedure step. The execution of the quit procedure step consists of the following actions: (1) the subtracting of 1 from the integer at position $\langle x, r\rangle$ of the full control matrix, (2) the adding of 1 to the integer at position $\langle p, x\rangle$ of the full control matrix, (3) the subtracting of 1 from the integer at position $\langle x, x\rangle$ of the full control matrix, and (4) the deleting of clerk x.

The permission rule for the execution of any of the four types of procedure steps defined above is the following. To execute one of the steps, a clerk must have three capabilities: (1) write capability for itself, (2) read capability for the segment containing the encoded procedure step, and (3) write capability for the name of the VM part being created or deleted. In addition, the execution of a fork requires read capability for the segment that is to become the private segment of the created clerk.

The general nature of the transaction sequences in M that correspond to creation and deletion actions in VM was mentioned in a previous Section. Descriptions of the exact transaction sequence correspondences for the four types of procedure steps introduced above are omitted, because, except as discussed below, these correspondences may be easily deduced from the correspondences between VM parts and clusters of cells, and from the previously mentioned transaction sequence correspondences for procedure steps of the types send and done.

The one non-trivial aspect of the transaction sequence correspondences for the procedure steps just introduced is the fact that the transaction sequence corresponding to a quit must end with a bye transaction. Let x be the name of the clerk executing a quit, and let p be as it was defined in the above definition of a quit. Then the transaction sequence that corresponds to clerk x's execution of quit concludes with

bye to $p.\sigma$ replace $\emptyset$

In the event that execution of the quit causes clerk x to give up write capability for itself, then, as shown in Appendix A, a sequence of sends and dones cannot be substituted for this bye transaction. This is the only circumstance in which the use of a bye transaction is required in order to maintain the correspondence between VM and M.

Another Example -- Macro Expansion

To illustrate how EF might be programmed to perform a reasonably complex data processing task, a program, written in an Algol-like language, will now be presented that directs the expansion of nested macro calls [18] in a source program character string. It is assumed that all macros have been previously defined, but that the expansion of one macro may reveal calls on other macros, and so on, to arbitrary depth. The nature of the computation is such that at any instant the number of clerks in existence is roughly equal to the depth of the macro nest being processed at that instant.

Strategy. The strategy of the program is to feed the source string to a first clerk, which expands one layer of macro calls and generates a first intermediate string. If the first clerk detects additional macro calls as it is generating its intermediate string, it creates, and feeds its intermediate string to, a second clerk, which might for a similar reason create, and feed its intermediate string to, a third clerk, and so on.

Each of the strings, source and intermediate, is stored in a series of segments, each segment containing a piece of a string of perhaps several hundred characters. When the n-th clerk is able to generate an entire segment in which no further macro calls exist, it outputs this segment instead of sending it on to the (n+1)-st clerk. In this circumstance, if there already exist clerks of ordinality higher than n, the n-th clerk generates an "end message", which causes the higher-order clerks to eventually generate their output and then quit. Meanwhile, the permission rule mechanism automatically prevents outputting by the n-th clerk until the higher order clerks have finished their outputting.

Language Conventions. Some language conventions to be used in the program will now be explained. Later the program itself will be stated and discussed in detail.

First, the operators that appear in name expressions will be introduced. As has been mentioned, single quotation marks inhibit evaluation of the string they enclose. For example, the value of the name expression

```
'x:ab'
```

is the string

```
x:ab
```

Any arithmetic expression whose value is a non-negative integer is a name expression whose value is the string of decimal numerals giving the arithmetic value of the expression. For example, the value of the name expression

```
16 + 8
```

is the string of two numerals

```
24
```

The character "|" appearing in a name expression, and not belonging to a quoted string, is a concatenation operator. For example, the value of the name expression

```
'sam'|(16 + 8)
```

is the string

```
sam24
```

For another example, the value of the name expression

```
'sam:'|'(16 + 8)'
```

is the string

```
sam:(16 + 8)
```

In the evaluation of a name expression, after any arithmetic expressions have been evaluated, and after any quotation marks and concatenation operators have been applied, the result might be a string beginning with a colon. If the result of arithmetic, quotation, and concatenation evaluation is a string beginning with a colon, then the name of the executing clerk is placed to the left of the string.*
For example, if x is the name of the executing clerk, the string

```
:ab
```

becomes the name

```
x:ab
```

After a string is examined for a leading colon, and modified if such a colon is found, the string is searched from left to right for occurrences of the sub-string ":*". Whenever such a sub-string is found, the longest string that lies immediately to the left of the sub-string and that does not contain a colon is deleted, along with the sub-string ":*" itself. For example, the string

```
x:sam:*:a
```

becomes the name

```
x:a
```

This usage of colon, and the following usage of ":" were inspired by the notation that has been suggested by Daley and Neumann [7] for referring to hierarchically structured files.

If an initial or final colon remains after this operation, the colon is deleted. For example, the two strings

```
x:*:ab:c          and          x:y:*
```

become, respectively, the names

```
ab:c              and          x
```

A more complicated example of a name expression, whose evaluation invokes almost all of the above rules, is the expression

```
':*:sam'|(2 + 2)
```

If x:y is the name of the executing clerk, then this expression evaluates to the name

```
x:sam4
```

The explanation of name expression operators is now complete, and some other conventions will now be discussed. Variables in arithmetic expressions refer to words in the executing clerk's private segment. For example, execution of the procedure step

```
alpha := alpha + 1
```

adds 1 to the quantity alpha in the executing clerk's private segment. Two different clerks executing this statement will modify two distinct, possibly unequal, quantities. For another example, if x:y is the name of the executing clerk, and if the quantity alpha in clerk x:y's private segment is equal to 5, then the value of the name expression

```
':sam'|(alpha + 3)
```

is the name

```
x:y:sam8
```

Two different clerks evaluating this name expression might generate two different names.

References to words in other segments besides the executing clerk's private segment have the form

$\alpha . \beta$

where α is a name expression, and β is a variable. The meaning of such a reference is best explained using an example. The quantity referred to by

'a'.alpha

is the word in segment a at the address that is the same as the address of the word alpha in the executing clerk's private segment. Thus, if the quantity alpha in the executing clerk's private segment is the word at address 7 in that segment, then execution of the procedure step

'a'.alpha := 'a'.alpha + 1

adds 1 to the word at address 7 in segment a.

Variables declared agreed are assigned to the same address in the private segment of every clerk; this convention makes it easy for a clerk initializing the activities of a new clerk to store quantities into what will become the private segment of the new clerk.

Two final conventions are the following. First, for legibility, the Algol operator ":=" is replaced by "=". Second, for compactness, the procedure step

give α, β;

where α and β are name expressions, is defined to be equivalent to the sequence of procedure steps

> send α, β;
>
> done α;

The Program. The coding of the macro expansion program can now be presented. On a first reading, it is suggested that the reader read into the program as far as he can easily, then skip to the explanation that follows the program, and then refer back to the program as he reads the explanation.

Assumed to be declared outside the program are the procedures "read input into", "write output from", "end of input segment", "initialize analyze", "analyze", "add end message to", and "remove end message from". All of these procedures and the program itself are assumed to be encoded into the words of one segment. A single clerk, whose name is not the master name, is assumed to enter the program at the label "expand". At the time of this clerk's entry, the control matrix is assumed to have two rows and five columns: the row for the entering clerk is a row of five 1's, and the row for the master name is a row of five 0's. The five columns correspond to (1) the entering clerk itself, (2) the entering clerk's private segment, (3) the procedure segment, (4) a segment named "inputseg" containing the computation's entire input string, and (5) a segment named "outputseg" into which the computation's entire output string is to be assembled.

```
expand:  begin agreed integer in, out;
         agreed Boolean next clerk started;
         switch switch = input, output, more, done;
         out = 1;
         create segment 'priv';
         ':priv'.in = out;
         fork 'scanner', ':priv', scan;
         give 'outputseg', ':scanner';
read:    create segment 'piece'|out;
         if end of input segment (inputseg) then goto end;
         read input into (':piece'|out) from:(inputseg);
         give ':piece'|out, ':scanner';
         out = out + 1;
         goto read;
scan:    out = 1;
         next clerk started = false;
         initialize analyze;
first:   create segment 'piece'|out;
second:  goto switch[analyze (':*:piece'|in) generate:(':piece'|out)];
input:   delete ':*:piece'|in;
         in = in + 1;
         goto second;
output:  if ¬ next clerk started then goto putout;
         next clerk started = false;
         add end message to (':piece'|out);
         give ':piece'|out, ':scanner';
         remove end message from (':piece'|out);
```

```
putout:   write output from (':piece'|out) into:(outputseg);
          goto second;
more:     if next clerk started then goto again;
          create segment 'priv';
          ':priv'.in = out;
          fork 'scanner', ':priv', scan;
          give 'outputseg', ':scanner';
          next clerk started = true;
again:    give ':piece'|out, ':scanner';
          out = out + 1;
          goto first;
done:     if ¬ next clerk started then goto doneout;
          next clerk started = false;
          add end message to (':piece'|out);
          give ':piece'|out, ':scanner';
doneout:  delete ':piece'|out;
          give ':*:piece'|in, ':*';
          give 'outputseg', ':*';
          delete ':*:priv';
          quit;
end:      add end message to (':piece'|out);
          give ':piece'|out, ':scanner';
          delete ':piece'|out;
end
```

Explanation. A detailed explanation of the execution of the program will now be given. The input string is divided into segmented pieces by a clerk whose name is, say, "compiler". The pieces, which are named

```
compiler:piece1
compiler:piece2
      •
      •
```

are given to the first created clerk, which is named

```
compiler:scanner
```

The private segment of this first scanner clerk is named

```
compiler:priv
```

The integer-valued procedure

```
analyze (A) generate:(B)
```

absorbs input pieces through the parameter A, performs one layer of macro expansion, and emits segmented output pieces through the parameter B. These output pieces for the first scanner clerk are named

```
compiler:scanner:piece1
compiler:scanner:piece2
              •
              •
```

The analyzer procedure, like each of the procedures called during the program's execution, is assumed to not direct its own modification, but is assumed to store temporary quantities in the calling clerk's private segment. In addition, the analyzer procedure is assumed to store private-to-clerk **own** quantities [2] in the calling clerk's private segment; thus with respect to each clerk, the analyzer procedure "remembers" its status between calls. These **own** quantities are initialized for each clerk when the clerk calls the procedure "initialize analyze".

A clerk returning from the analyzer is switched to the label "input" just when another input piece is needed. The previous input piece is deleted, and 1 is added to an input count "in" so that the analyzer will read the next piece of its input string.

A clerk returning from the analyzer is switched to the label "output" just when the analyzer has filled an output piece segment with a string in which there are no additional macro calls. If a clerk of next higher order exists, an end message is sent to this clerk. After any such end message has been returned, the current clerk writes the data of the current output piece into the computation's output segment, but retains the segment of this output piece to collect further output.

A clerk returning from the analyzer is switched to the label "more" just when an output piece contains additional macro calls. The output piece is given to the clerk of next highest order, which is created if necessary. Then 1 is added to an output count "out", and a new output piece segment is created before another call is made to the analyzer.

A clerk returning from the analyzer is switched to the label "done" just when the analyzer has detected that its input piece contains an end message, and before any other processing of the input piece is performed, but only after the analyzer has first switched the clerk to either "output" or "more" in order to complete the processing of a perhaps partially filled output piece segment. At the label "done", an end message is sent to and returned from a higher order clerk, if a higher order clerk exists. Then both the input piece, containing the original end message, and the capability to write into the computation's output segment are returned to the clerk of next lowest order, and the current clerk quits.

There are no other labels to which a clerk returning from the analyzer can be switched. When the "compiler" clerk detects the end of the computation's input segment, the "compiler" clerk sends an end message to the first scanner clerk, but does not exit from the program until the first scanner clerk returns this message. When the "compiler" clerk exits from the program, the full control matrix is the same as it was when the "compiler" clerk entered the program.

Control Matrix Implementation

This Section suggests how EF's physical units and supervisory programs might be designed in order to implement the most novel feature of VM, namely, the control matrix. To speed the communication of ideas, a suggested design for control matrix implementation will be developed within the general framework of the Multics system [4]. Specifically, the Section will discuss how the current design for the Multics system might be modified so as to provide to a user both a control matrix, and the permission rule mechanism associated with the control matrix. These remarks concerning a modification to Multics are made only for illustrative purposes; no claim is made that this modification would be an appropriate change in the existing plans for Multics.

First the notion of a process in Multics will be related to the notions of process and clerk that have been used in the Thesis. Since every Multics process is a sequence of actions, the notion of a process in Multics is compatible with the notion of process that has been used in the Thesis. In the Multics literature, however, one often finds statements such as, "Process x does such-and-such." Although it is possible to give a reasonable interpretation to an assertion that a sequence of actions does something, nevertheless the viewpoint that has been adopted in the Thesis is that of attributing an action in a process to an entity, called a clerk, that performs the action, and not to the process itself. Thus statements in the Multics literature like, "Process x does such-and-such" will become here, "Clerk x does such-and-such." Notice that there is a one-to-one correspondence between processes and clerks.

To simplify the discussion, it will be assumed that a user employs in a computation just a fixed number of clerks, a fixed number of segments, and a control matrix; it will also be assumed that each clerk always has write capability for itself, but not read capability for any other clerk. In other words, input devices, output devices, creation-deletion activity, and inter-clerk state word accessing will be assumed to not be employed in a user's computation. The present discussion may, however, be extended in a straightforward manner to take into account these omitted topics.

Working under the above assumptions, let us review some of the features of Multics in order to define a viewpoint toward the system. Associated with each clerk x is a descriptor segment [15], each descriptor of which contains access control information and allocation information for a specific segment used by clerk x. Although clerk x makes its first reference to a segment n using what would in VM be called the name of segment n, subsequent references to segment n occur indirectly through segment n's descriptor in x's descriptor segment. The address of segment n's descriptor in x's descriptor segment is called x's segment number for n. After x first refers to n, x may make subsequent references to n using x's segment number for n. Thus x's segment number for n constitutes an attachment tag [9] abbreviating segment n's name. Further abbreviation of segment references through attachment is allowed through the use of the base registers.

Control Segments. In order to implement the control matrix of a multiprocess computation running within Multics, let there exist for each clerk of the computation, in addition to a descriptor segment, a control segment. Part of the function of a clerk's control segment is to realize the clerk's row in the computation's control matrix. Each entry in a clerk's control segment corresponds to a segment.

As indicated in Figure 3.8, the entries for a segment n in the control segments of a computation all occur at the same address. This situation is in contrast to the situation that prevails in a computation's descriptor segments. The entries, i.e., descriptors, for a segment n in the descriptor segments of a computation may occur at different addresses. That is, each segment of a computation has only one control segment address, but each segment of a computation might have many distinct segment numbers, one number for each clerk of the computation.

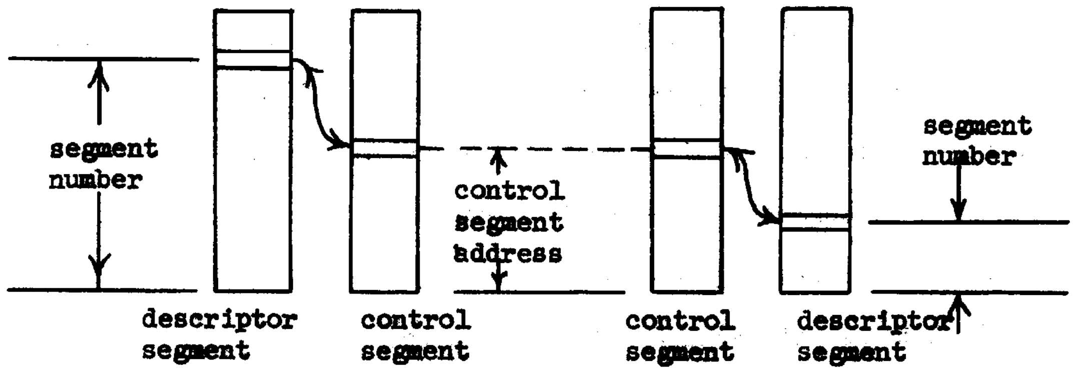


Figure 3.8. Segment numbers and the control segment address of a segment n in a two-clerk computation.

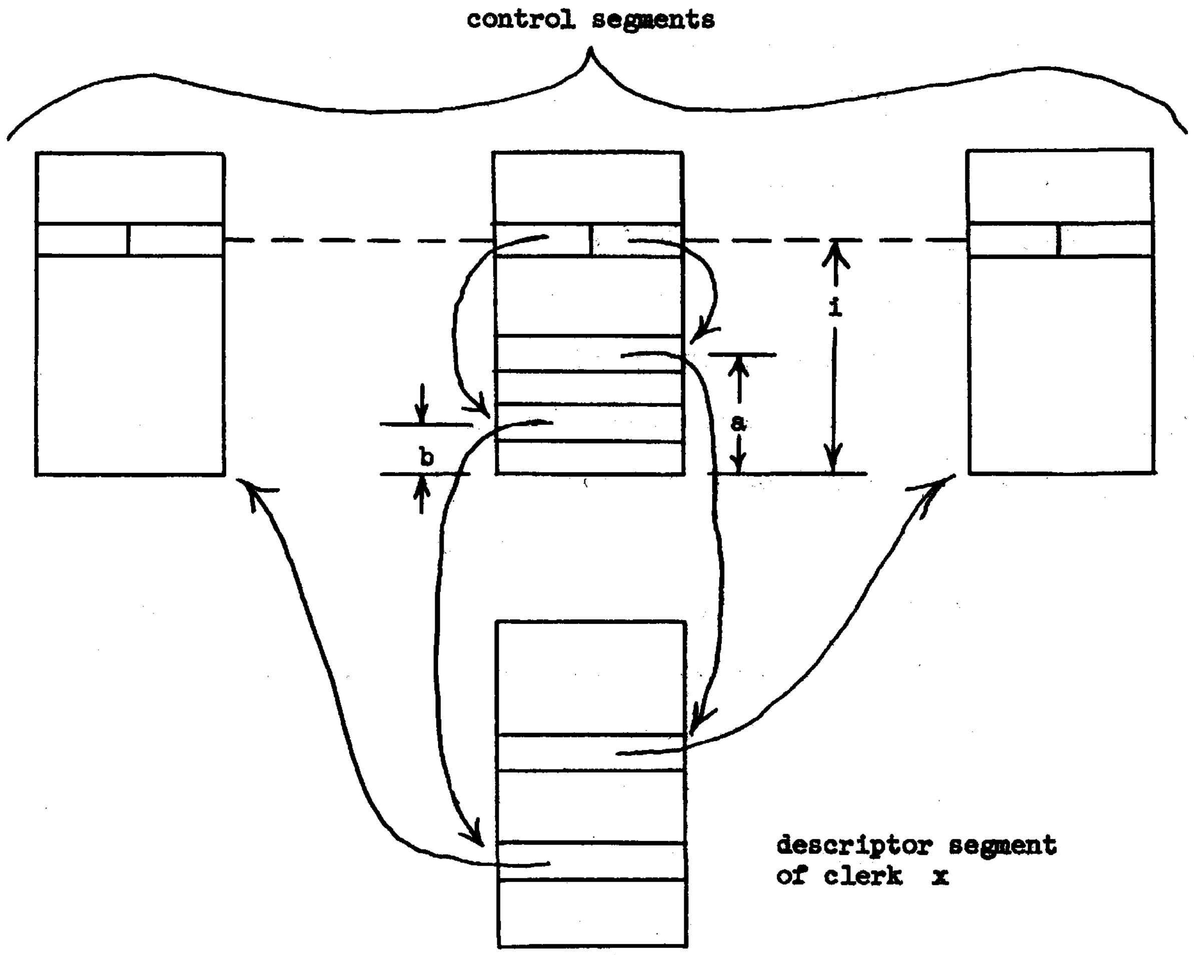


Figure 3.9. List structure linking clerks having read capability for a segment. Notation is that used in the text.

A control segment address is assigned to a segment n by the computer system just when n is first accessed by any one of a computation's clerks. At that time also, a segment number is assigned to n; but for the accessing clerk only. A subsequent first access to n by a second clerk will cause the assignment of perhaps a different segment number to n for the second clerk, but in order to update the control segment of the second clerk, the original control segment address assignment must be discovered and used.

In Multics a clerk becomes attached through its descriptor segment to any segment that it accesses. No modification to this rule is being proposed. Since it will be necessary for a clerk, often in obedience to a supervisory "microprogram", to make access to its own control segment, and to the control and descriptor segments of other clerks, therefore control and descriptor segments will possess, after first access, segment numbers with respect to each accessing clerk.

In addition, control and descriptor segments will be assigned not only segment numbers, but also control segment addresses. The assignment of control segment addresses to control and descriptor segments is subject only to the convention that if i is the control segment address of the control segment of a clerk x, then i+1 must be the control segment address of the descriptor segment of clerk x.

The Structure after First Accesses. The main requirement of a design for control matrix implementation is that operations on the matrix be efficient, in terms of both time and storage, after first accesses have been made. First accesses to segments trigger the incremental construction of the control matrix data structure about to be described. It is assumed that contemporary data processing techniques, such as hash-addressing and list structures, can be applied to carry out this incremental construction in a satisfactory way. To show that a control matrix implementation is indeed feasible, it will be assumed now, and throughout the rest of the Section, that all first accesses have occurred. The form of the resulting control matrix data structure will now be described.

If segment n has control segment address i, then the entry at address i in clerk x's control segment holds three items: (1) the integer K at position $\langle x, n \rangle$ of the computation's control matrix, (2) clerk x's segment number for n, and (3) a double pointer. If segment n is a control or descriptor segment, then items (1) and (3) have no significance. Let us assume that segment n is not a control or descriptor segment. If K is less than or equal to zero, then the double pointer field is empty, i.e., contains some conventional bit pattern denoting "empty", and otherwise, the double pointer field is part of a bi-directional circular list tying together the control segments of all clerks that have read capability for segment n. In this list, a portion of which is diagrammed in Figure 3.9, the pointers at address i in clerk x's control segment give the control segment addresses, call them a and b, of the two control segments adjacent to x's control segment on the list. To gain access to the next double pointer in the direction of, say, the address a, clerk x first accesses its own control segment at address a to obtain its own segment number for the control segment pointed to. In that control segment at address i is found the double pointer that is next on the list in the "a" direction.

Since the entries in the double pointer field of a control segment are control segment addresses, their designation of specific control segments is independent of their residency in any particular control segment. Thus, ordinary techniques of list manipulation may be used to splice a control segment in or out of a list.

If a user could program in a general way the manipulation of control segment addresses or control matrix elements, then he might experience nonfunctionality and noncompletion lurking bug effects, because the values of control segment addresses and control matrix elements can depend on the unpredictable influences that affect the progress of clerks with respect to each other.* Therefore control segment addresses and control matrix elements must be utilized "behind the user's back" -- in the same sense that the state of memory allocation in Multics is hidden from the user. The suggested method for hiding control segment addresses and control matrix elements from the user is to employ these quantities only within descriptor and control segments, and to restrict user access to control segments in the same way that user access to descriptor segments is restricted in Multics.

*It is useful to understand that the programmed manipulation of segment numbers does not give rise to lurking bugs, because segment numbers are assigned to segments on the basis of the progress of each individual clerk, just as index registers are assigned to data quantities.

Descriptor Segment Modifications. The suggested modification to Multics calls for each descriptor in a descriptor segment to contain not two but three major items: access control information, allocation information, and the control segment address of the segment the descriptor describes. Also, additional access control information is required in each descriptor in the form of four additional bits: a read enable bit, a write enable bit, a read blocked bit, and a write blocked bit.

Suppose that i is the control segment address of a segment n, and that j is clerk x's segment number for n. From the discussion so far, one can conclude that the i-th entry in x's control segment contains j, and that the j-th entry in x's descriptor segment contains i. After a clerk y has first made access to the control and descriptor segments of clerk x, then clerk y can, and presumably nearly always does, make subsequent access to these segments using its own segment numbers for these segments. Therefore, from the point of view of a clerk such as clerk y, there exists a bi-directional link between the entries for segment n in clerk x's control and descriptor segments. Two such bi-directional links are shown in Figure 3.8.

The Permission Rule Mechanism. An explanation will now be given of the implementation of four types of actions: (1) reading from an ordinary segment, i.e., from a segment that is neither a control segment nor a descriptor segment, (2) writing into an ordinary segment, (3) adding 1 to a control matrix element, and (4) subtracting 1 from a control matrix element.

A clerk reads from an ordinary segment in the following way. If the Multics <u>read permit</u> bit [15] is on, and the read enable bit is also on, then the reading proceeds normally. If the read enable bit is off, then the read blocked bit is turned on, and the clerk goes into the <u>blocked</u> status [26].

A clerk writes into an ordinary segment in the following way. If the Multics <u>write permit</u> bit is on, and the write enable bit is also on, then the writing proceeds normally. If the write enable bit is off, then the write blocked bit is turned on, and the clerk goes into the blocked status.

A clerk x adds 1 to the integer at position $\langle e, n \rangle$ of the control matrix by acting as if it executed the following program.

1. If the read enable bit in x's descriptor of n is off, turn on the read blocked bit in the same descriptor, and go into the blocked status; otherwise proceed to step (2).

2. Determine the control segment address of n by referring to x's descriptor of n.

3. Add 1 to the control matrix integer for n in e's control segment. If the result is +1, perform the following steps.

   a) Splice e's control segment into the read capability list for n.

   b) Turn on the read enable bit in e's descriptor of n.

   c) If the read blocked bit is on in the same descriptor, turn it off and take e out of the blocked status.

A clerk x subtracts 1 from the integer at position $\langle x, n \rangle$ of the control matrix by acting as if it executed the following program.

1. Determine the control segment address of n by referring to x's descriptor of n.

2. Subtract 1 from the control matrix integer for n in x's control segment. If the result is 0, perform the following steps.

   a) Turn off the read enable and write enable bits in x's descriptor of n.

   b) Splice x's control segment out of the read capability list for n.

   c) Examine the double pointer field for n in x's control segment. If the field is not empty, and if its two addresses are equal, say to some address a, then perform the following steps.

      i) Examine x's control segment at a to obtain x's segment number for the control segment of some clerk, p. Examine x's control segment at a+1 to obtain x's segment number for the descriptor segment of p.

      ii) Examine p's control segment at the control segment address of n to determine p's segment number for n.

      iii) Turn on the write enable bit in p's descriptor of n.

      iv) If the write blocked bit is on in the same descriptor, turn it off, and take p out of the blocked status.

It is clear that the incrementing and decrementing of control matrix elements can be implemented either by hardware, or by a supervisory program.

<u>The Associative Memory</u>. Each processing unit in the GE 645 computing system, on which Multics is currently being implemented, contains an associative memory in which information obtained during recent accesses to a clerk's descriptor segment, and to page tables, is remembered [15]. The problem of detecting invalid entries in this associative memory suffers only a small augmentation because of the

modification that has been outlined here. The augmentation to this problem is small because a read or write capability possessed by a clerk can only be denied to the clerk by an action performed by the clerk itself, specifically, by the clerk's decrementing of a control matrix integer.

Conclusion. Many issues concerning the modification of Multics to realize a control matrix have not been discussed. Nevertheless, it is clear that the implementation of a control matrix in Multics is very likely to be feasible, and that this implementation would be worth exploring further in the event that the goals to be achieved by such an implementation should prove desirable.

Similar remarks may be made concerning the Chapter's overall topic, the facility EF. It is clear that the construction and use of a facility that behaves like an MCM are both very likely to be feasible, and that a further exploration of the problems involved would be justified in the event that the goals to be achieved were deemed desirable.

Chapter IV

The Output Functionality of an MCM

Introduction

This Chapter presents a proof of the fact that nonfunctionality lurking bug effects do not occur in an MCM, or in other words, that every MCM is output-functional. According to the definition given in Chapter I, an MCM is output-functional just when each output symbol produced in every output stream of the MCM is a function only of the MCM's initial computation state. It may be recalled from the description in Chapter II that an MCM produces an output symbol just when a clerk cell writes into one of the MCM's designated output cells.

In pursuit of the proof of output functionality, a set-theoretic entity called a run is introduced as a formal description of a computation. Then the notion of a history array is introduced to describe certain properties of a run. Using runs and history arrays, a theorem, called the functionality theorem, is stated and proved; the truth of the functionality theorem implies that every MCM is output-functional.

Specialization to a Single Arbitrary MCM

The remarks to be made in this Chapter concern one specific, but arbitrary, MCM. This MCM, which will be designated by the letter M, is described by four quantities: (1) a set N of cell names, (2) a set Q of output cell names that is a subset of N, (3) a transaction table J(i) for each i belonging to N, and (4) a set I of initial computation states. It is assumed that this machine M is well-defined, in the sense described near the end of Chapter II. Since M is arbitrary, the remarks to be made about M, including the functionality theorem, are true for any well-defined MCM.

The Run as a Formal Description of a Computation

A computation is an instance of the behavior of the machine M; in other words, a computation is that activity which ensues when M is started up from an initial computation state. The problem of describing a computation performed by M may be compared with the problem of describing an instance of the behavior of, say, a finite automaton [24]. Since the identity of each successive state of a finite automaton is determined by the immediately preceding state, then an instance of the behavior of a finite automaton is determined solely by the automaton's initial state, and therefore is described completely by this initial state.

A computation performed by M is determined, in general, not only by M's initial computation state, but also by the successive selections that M's scheduler makes in response to perhaps unpredictable influences. A suitable description of a computation performed by M must therefore both give the initial computation state, and also identify the clerk cells that participate in each computation state transition. One kind of description that meets these specifications is the run. A run is an ordered pair such as

$$\langle S, T \rangle$$

where S is an initial computation state, and T is a transition sequence, in which each T_i is a set containing the names of the clerk cells that accomplish the i-th computation state transition of the denoted computation. Thus, for example, a run with T equal to

$$\left\langle T_1, T_2, \ldots, T_m \right\rangle$$

is a run that describes a computation in which m computation state transitions occur. The number of elements in a run's transition sequence is called the length of the run; the length of the run $\langle S, T \rangle$, above, is therefore m.

Not every set-theoretic entity having the form of a run describes a computation that M might perform. Specifically, if $\langle S, T \rangle$ is a run, then S must belong to I, the set of initial computation states of M, and each element T_i of the sequence T must be a member of the choice collection that is derived from the computation state immediately preceding the i-th computation state transition. It may be recalled from Chapter II that the choice collection derived from a computation state is a collection of sets of cell names; it is from this collection that the scheduler selects one set to be the set of the names of the cells that will accomplish the next computation state transition.

The foregoing restrictions on the initial computation state and transition sequence of a run may be expressed concisely by saying that every run must be possible. A run is possible just when it describes a computation that M might perform. In order that the notion of a possible run might be thoroughly understood, an alternate, more formal definition of a possible run will now be developed.

To aid in the defining of a possible run, let two functions, $e(\cdot)$ and $n(\cdot, \cdot)$, be defined in the following way. Suppose M holds some arbitrary computation state S, not necessarily belonging to I. Let e(S) denote the choice collection derived from S, and if A is a set of names of cells, let n(S, A) denote the computation state that prevails after each of the cells named in A has performed its next transaction.

A run, $\langle S_1, T \rangle$, is possible if and only if*

$$S_1 \in I$$

and**

$$T_i \in e(S_i) \qquad ; \ i \in \mathcal{D}T$$

where

$$S_{i+1} = n(S_i, T_i) \qquad ; \ i \in \mathcal{D}T$$

It is convenient to define the predicate p(·, ·) so that p(S, T) is true just when $\langle S, T \rangle$ is a possible run. Henceforth, every run discussed is assumed to be possible.

A useful concept is that of a prefix run. A run, P = $\langle U, V \rangle$, is a prefix of the run, R = $\langle S, T \rangle$, if and only if S = U and V is

*The proposition $A \in B$ is true if and only if A belongs to B. For example, $a \in \{a, b\}$. The set-theoretic and logical notation being introduced in footnotes is summarized in Appendix C.

**The set $\mathcal{D}F$ is the domain of the function F, i.e., the set of arguments for which F is defined. A sequence, such as T above, may be thought of as a function that takes an integer i into the i-th element of the sequence. Thus, for example, the domain of the sequence $\langle T_1, T_2, \ldots, T_m \rangle$ is the set $\{1, 2, \ldots, m\}$.

an initial subsequence of T. For example, the prefix runs of

$$\langle S, \langle T_1, T_2\rangle\rangle$$

are

$$\langle S, \langle\rangle\rangle$$
$$\langle S, \langle T_1\rangle\rangle$$
$$\langle S, \langle T_1, T_2\rangle\rangle$$

where $\langle\rangle$ is the empty sequence. Clearly, if R is a possible run, then every prefix of R is also a possible run.

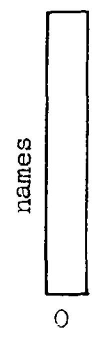
The History Array of a Run

Let us recall from Chapter II the notion of writing into a cell. A clerk cell x writes into a cell i just when x causes a new word to be held by i, where the new word might or might not be the same as the word previously held by i. The only circumstances in which a write into a cell occurs during a computation are the following two circumstances: A clerk cell performing a put writes into the cell named by the put's operand name, and a clerk cell performing any transaction writes into itself. The notion of writing is vital for understanding the notion of a history array; the latter notion is now introduced.

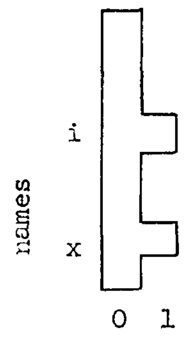
Let R be a run, describing, of course, a computation that might be performed by the machine M. If M has n cells, then there are exactly n rows in the <u>history array</u> H of the run R; that is, to each cell there corresponds a row in H, and vice-versa. If i is a cell name, then the number of elements in row i of H is one greater than the number of writes into cell i that occur during the computation denoted by R. Specifically, the elements of the array H are defined as follows: H_{ij} is the word written into cell i by the j-th write into cell i during the computation denoted by R, and H_{i0} is the word held initially in cell i, that is, before any writes into cell i have occurred. Thus row i of H is a "history" of the words that are caused to be held by cell i, either by writing, or by M's being placed in an initial computation state.

Before explaining further the notion of a history array, it is convenient to introduce a more concise way of talking about runs: events that occur during the computation denoted by a run will be said to occur "during the run", even though the latter locution is not, strictly speaking, correct.

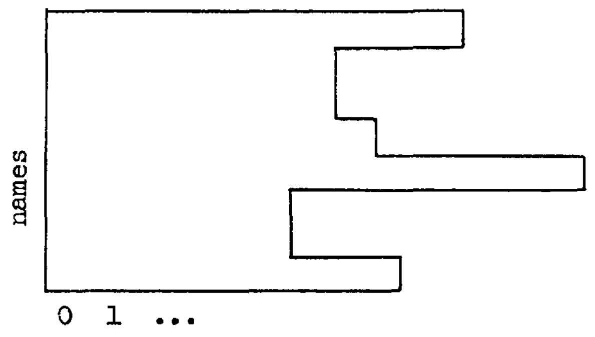
Returning to the explanation of history arrays, let us examine, with the aid of Figure 4.1, the appearance of a few typical history arrays. If no computation state transitions occur during a run, i.e., if the run is of length zero, then as shown in Figure 4.1(a), the history array of the run consists of just one column. If exactly one transaction occurs during a run, and if this transaction is a put of i that is performed by a cell x where x is not equal to i, then as shown in Figure 4.1(b), exactly two rows of the run's history array contain two elements each, and any remaining rows of the array contain one element each. As shown in Figure 4.1(c), the history array of a run of substantial length would most likely have a jagged right edge, since the maximum column subscript in any row is equal to the number of writes that have occurred into the cell to which the row corresponds.



(a) For a run of length zero.



(b) For a run during which exactly one transaction is performed, namely, a put of i performed by a cell x where x is not equal to i.



(c) For a run of substantial length.

Figure 4.1. Three typical history arrays.

One can think of the array H as a set of ordered pairs of the form [25]

$$\langle\langle i, j\rangle, H_{ij}\rangle$$

Using this convention for the history arrays G and H, the statement*

$$G \subseteq H$$

means, considering an array to be a function whose domain is a set of ordered pairs, that

$$\mathcal{D}G \subseteq \mathcal{D}H$$

$$G_{ij} = H_{ij} \qquad ; \langle i, j\rangle \in \mathcal{D}G$$

In other words, H is defined everywhere that G is defined, and each element of G is equal to the corresponding element of H.

A useful property of history arrays is that if G and H are the history arrays of the runs P and R, respectively, and if P is a prefix of R, then

$$G \subseteq H$$

That this inclusion is true may be verified in three steps. First, P and R both have the same initial computation state; therefore the zeroth columns of G and H are equal. Second, if the j-th write into cell i actually occurs in P, i.e., if G_{ij} is defined, then since P is a prefix of R, the j-th write into cell i must occur in R, i.e., H_{ij} must be defined. Therefore H is defined everywhere that G is

*The proposition $A \subseteq B$ is true if and only if A is included in B, i.e., if and only if A is a subset of B. For example, $\{a\} \subseteq \{a, b\}$, $\{a, b\} \subseteq \{a, b\}$, and $\{\} \subseteq \{a, b\}$, where $\{\}$ is the empty set.

defined. Third, since the events in P up to and including the j-th write into cell i are identical to the events in R up to and including the j-th write into cell i, then the j-th word written into cell i must be the same in both P and R, and so G_{ij} must equal H_{ij}. Therefore, G and H are equal everywhere that G is defined. The truth of the inclusion is thus verified.

Let us define the proposition $\#H_{ij}$ to be true if and only if the history array element H_{ij} is defined, i.e., if and only if $\langle i, j \rangle \in \mathfrak{D}H$.

The relation of <u>similarity</u>, $\cong$, for two history arrays G and H is defined as follows.

$$G \cong H$$

if and only if*

$$(i)(j)(\#G_{ij} \wedge \#H_{ij} \rightarrow G_{ij} = H_{ij}) \tag{4.1}$$

In other words, two history arrays are similar if and only if they are equal at every position where both are defined.

It is convenient to define the function $h(\cdot, \cdot)$ so that $h(S, T)$ is the history array of the run $\langle S, T \rangle$. In other words, if H is the history array of $\langle S, T \rangle$, then

$$H = h(S, T)$$

*The proposition (x)A, where A is usually a function of x, is true if and only if A is true for every x. The proposition $A \wedge B$ is true if and only if both A and B are true. The proposition $A \rightarrow B$ is true if and only if A implies B, i.e., if and only if either A is false, or both A and B are true.

Statement of the Functionality Theorem

The functionality theorem is stated as follows: for the machine M, any two possible runs that have the same initial computation state have similar history arrays. In symbols, the functionality theorem is

$$(S)(T)(V)[p(S, T) \wedge p(S, V) \rightarrow h(S, T) \cong h(S, V)] \quad (4.2)$$

It must be emphasized again that since the machine M is arbitrary, the functionality theorem, when proved, will hold for any MCM.

What is the relationship between the functionality theorem and the notion of output functionality? Output functionality holds just when each word written into every output cell is a function only of the initial computation state. The functionality theorem says that each word written, not just into every output cell, but into every cell, is a function only of the initial computation state. That is, the functionality theorem asserts <u>complete</u> functionality, not just output functionality. It is clear that the functionality theorem implies output functionality. One way of understanding the functionality theorem is to observe that if every cell were an output cell then the theorem would not just imply, but be equivalent to output functionality.

It was mentioned at the beginning of Chapter II that questions concerning the necessity of the MCM design remain open. A particularly intriguing open question is, "Is complete functionality necessary for output functionality?" A proper answer to this question requires a broader framework than that being developed here, and is beyond the scope of the Thesis. Speculations concerning how this question might be answered are indulged in in Chapter VI. It may be noted that complete functionality is a worthwhile design goal in its own right, because complete functionality materially facilitates the debugging of programs.

The remainder of the Chapter is devoted to a proof of the functionality theorem, (4.2).

The Augmented Array

The proof of (4.2) will be given as an inductive proof of a proposition that implies (4.2), namely

$$(S)(T)(V)[p(S, T) \wedge p(S, V) \rightarrow a(S, T) \cong a(S, V)] \quad (4.3)$$

The function $a(\cdot, \cdot)$ is defined so that if $\langle S, T\rangle$ is a run, then $a(S, T)$ is the augmented array of the run $\langle S, T\rangle$. Just as for history arrays, two augmented arrays are similar if and only if they are equal at every position where both are defined.

The augmented array of a run has the same form as the history array of the run; that is, if an array is considered as a function defined on ordered pairs, then a run's augmented array and history array both have the same domain. The augmented array of a run contains all the information contained in the history array of the run, plus additional information. Specifically, if $\langle S, T\rangle$ is a run and

$$H = h(S, T)$$

$$A = a(S, T)$$

then

$$A_{ij} = \langle H_{ij}, x_{ij}, y_{ij}\rangle \qquad ; \langle i, j\rangle \in \mathfrak{D}H \text{ and } j \neq 0$$

$$A_{i0} = \langle H_{i0}, \phi, \phi\rangle \qquad ; \langle i, 0\rangle \in \mathfrak{D}H$$

where x_{ij} and y_{ij} are explained below, and where the symbol ϕ, which is assumed to not be the name of a cell of M, means "empty" or "meaningless".

In the above definition, position $\langle x_{ij}, y_{ij} \rangle$ of A "writes" position $\langle i, j \rangle$ of A. That is, the quantity x_{ij} is the name of the cell that performs the j-th write into cell i during the run $\langle S, T \rangle$, and the quantity y_{ij} is the number of writes that have occurred into the cell x_{ij} during the run $\langle S, T \rangle$ up to the instant just before the cell x_{ij} performs the j-th write into cell i. In other words, if $A_{ij} = \langle a, b, c \rangle$, and $j \neq 0$, then one might say that during $\langle S, T \rangle$, A_{bc} "writes" A_{ij}; what actually happens is that the j-th write into cell i is a write of the word a into cell i, and this write is performed by cell b at an instant when exactly c writes have occurred into cell b. Thus concerning each write that occurs during $\langle S, T \rangle$, the array A not only tells the word written, but also tells both the name of the cell that did the writing, and the ordinality of the writing cell's content.

Just as for a history array, let the proposition $\# A_{ij}$ be true if and only if the augmented array element A_{ij} is defined. Then the definition of similarity, (4.1), applies to augmented arrays as well as to history arrays. If $\langle S, T \rangle$ and $\langle S, V \rangle$ are runs, then clearly

$$a(S, T) \cong a(S, V) \rightarrow h(S, T) \cong h(S, V)$$

and so (4.3) implies the functionality theorem, (4.2).

If $\langle S, V \rangle$ is a prefix of $\langle S, T \rangle$, then

$$a(S, V) \subseteq a(S, T)$$

This statement is true because for each $\langle i, j \rangle$ such that $(a(S, V))_{ij}$ is defined, the three components of $(a(S, V))_{ij}$ and the three components of $(a(S, T))_{ij}$ depend only on the events in $\langle S, V\rangle$ and $\langle S, T\rangle$, respectively, up to and including the j-th write into cell i; since $\langle S, V\rangle$ is a prefix of $\langle S, T\rangle$, then these events in $\langle S, V\rangle$ are identical to these events in $\langle S, T\rangle$.

Introduction to the Proof

The proof of (4.3) will be an inductive proof that uses as the variable of induction the length of successively longer prefixes of the run $\langle S, V\rangle$. The initial step of the induction is to show that a(S, T) is similar to the augmented array of $\langle S, V\rangle$'s prefix of length 0. The inductive step is to show that if a(S, T) is similar to the augmented array of $\langle S, V\rangle$'s prefix of length n, then a(S, T) is similar to the augmented array of $\langle S, V\rangle$'s prefix of length $n + 1$.

After both the initial step, and the simplest case of the two cases in the inductive step have been proved, a digression will be made to introduce several auxiliary concepts that will be useful in proving the second case of the inductive step. Following the digression, the proof of the second case of the inductive step will be given. Since this proof of the second case is complex, the proof of the second case will be divided, for convenience, into three stages. In turn, the first stage will be divided into five tasks, and the third stage will be divided into three tasks.

The Inductive Formulation

In order to define the inductive proof of (4.3) precisely, the function $u(\cdot, \cdot)$ is now introduced so that if V is a transition sequence and n is a non-negative integer, then*

$$u(V, 0) = \langle\rangle$$
$$u(V, n) = \langle V_1, V_2, \ldots, V_n\rangle \qquad ; n \in \mathfrak{D}V$$
$$u(V, n) = V \qquad ; n \notin \mathfrak{D}V \text{ and } n > 0$$

Let the predicate $\pi(\cdot)$ be defined so that $\pi(n)$ is true if and only if

$$a(S, T) \cong a(S, u(V, n))$$

For example, if $n \in \mathfrak{D}V$, then $\pi(n)$ is true if and only if the augmented array of $\langle S, T\rangle$ is similar to the augmented array of $\langle S, V\rangle$'s prefix of length n.

By the induction principle, (4.3) is equivalent to

$$(S)(T)(V)\{p(S, T) \wedge p(S, V) \rightarrow \pi(0) \wedge (n)[n > 0 \rightarrow (\pi(n) \rightarrow \pi(n+1))]\} \qquad (4.4)$$

Proposition (4.4) is easily transformed into

$$(S)(T)(V)[p(S, T) \wedge p(S, V) \rightarrow \pi(0)] \qquad (4.5)$$
$$\wedge (S)(T)(V)(n)[(p(S, T) \wedge p(S, V) \wedge n > 0 \wedge \pi(n)) \rightarrow \pi(n+1)]$$

The initial step and the inductive step in the inductive proof of (4.3) will be established by proving the first and second conjuncts, respectively, of (4.5).

*The proposition $A \notin B$ is true if and only if A does not belong to B.

The Initial Step

The truth of the first conjunct of (4.5) follows from the fact that the runs $\langle S, T\rangle$ and $\langle S, V\rangle$ have a common initial computation state S. To verify this first conjunct, observe that during $\langle S, u(V, 0)\rangle$ no writes are performed into any cell, and so the augmented array a(S, u(V, 0)) consists of just the 0-th column. If c(·) is the content function denoting the state of M's cells when M has the computation state S, and if N is the set of M's cell names, then

$$[a(S, u(V, 0))]_{i0} = \langle c(i), \emptyset, \emptyset\rangle \qquad ; i \in N$$

But it is also true that

$$[a(S, T)]_{i0} = \langle c(i), \emptyset, \emptyset\rangle \qquad ; i \in N$$

Therefore, the arrays a(S, T) and a(S, u(V, 0)) are equal everywhere that elements are defined in both, and so we have

$$a(S, T) \cong a(S, u(V, 0))$$

which is $\pi(0)$. Thus, in the inductive proof of (4.3), the initial step has been proved.

Beginning the Inductive Step

In presenting the inductive step in the proof of (4.3), i.e., in presenting the proof of the second conjunct of (4.5), the following notation will be used.

$$R = \langle S, u(V, n)\rangle$$

$$R' = \langle S, u(V, n+1)\rangle$$

$$R^{+} = \langle S, T\rangle$$

$$A = a(S, u(V, n))$$

$$A' = a(S, u(V, n+1))$$

$$A^{+} = a(S, T)$$

Thus in the inductive step we are given

$$p(S, T) \wedge p(S, V) \wedge n > 0 \wedge A \cong A^{+} \tag{4.6}$$

and we must prove

$$A' \cong A^{+} \tag{4.7}$$

The environment of the inductive step is diagrammed in Figure 4.2.

According to the definition of similarity, (4.1), the job of the inductive step is to show that

$$(i)(j)(\#A'_{ij} \wedge \#A^{+}_{ij} \rightarrow A'_{ij} = A^{+}_{ij}) \tag{4.8}$$

The truth of (4.8) will be established by showing that $A'_{ij} = A^{+}_{ij}$ for an arbitrary array position, $\langle i, j\rangle$, at which both A' and A^{+} are defined.

The simpler of the two cases to be considered is the case in which position $\langle i, j\rangle$ is defined in A. Since R is a prefix of R', we have

$$A_{ij} = A'_{ij}$$

By similarity, we have

$$A_{ij} = A^{+}_{ij}$$

and so

$$A'_{ij} = A^{+}_{ij}$$

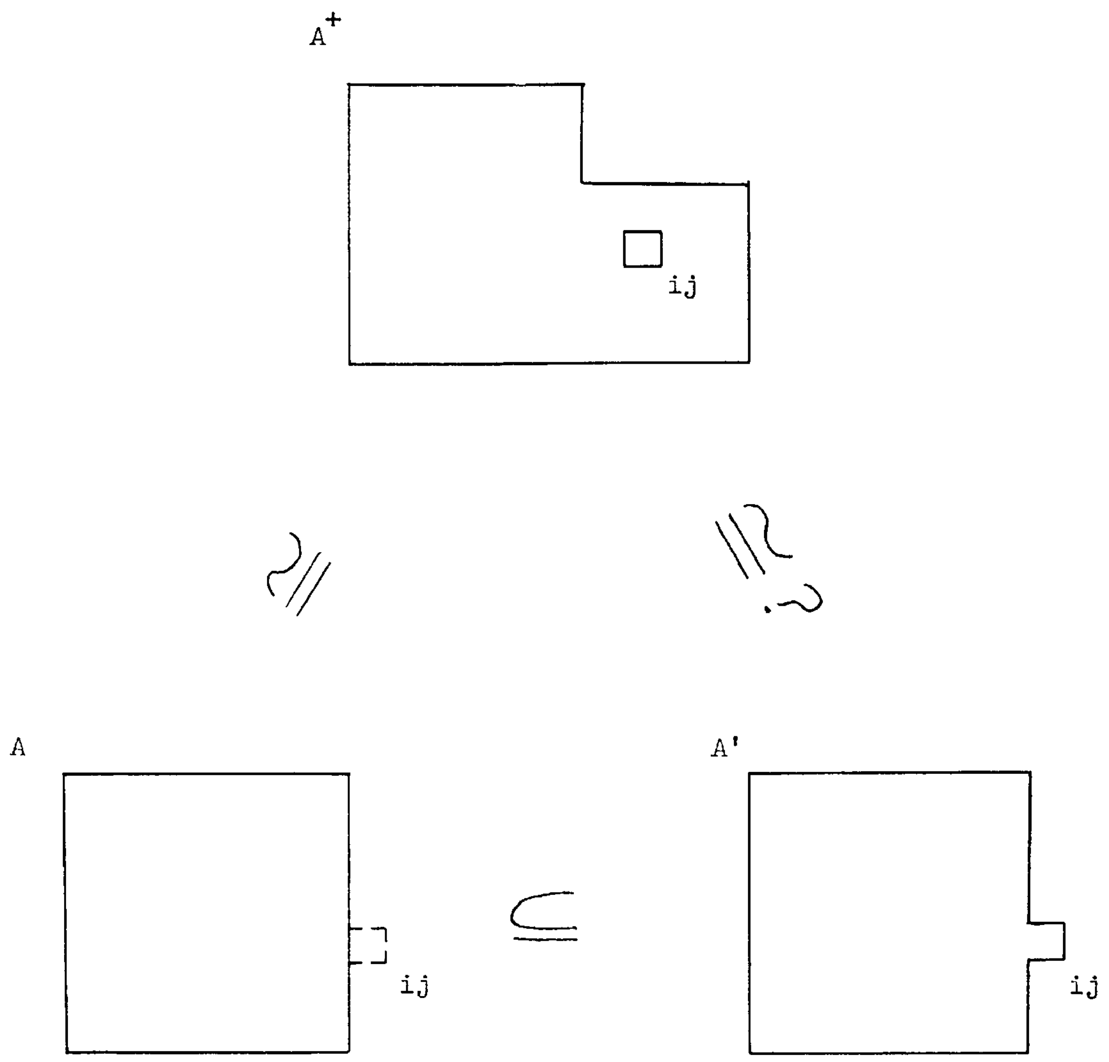


Figure 4.2. The environment of the inductive step.

The second case to be considered is the case in which position $\langle i, j \rangle$ is not defined in A; this is the case that is depicted in Figure 4.2. Here the j-th write into cell i occurs in the transition from R to R'; one might say that A'_{ij} is a "new" element of A'. Just as for the first case in the proof of the inductive step, the proof of $A'_{ij} = A^{+}_{ij}$ relies on the knowledge that R is a prefix of R', and that $A \cong A^{+}$. In addition, specific MCM properties, such as the enabling rules, are utilized.

The proof of the second case is complex. An adequate presentation of this proof requires the use of several auxiliary concepts. The number and intricacy of these concepts make it undesirable to introduce each at its first use during the proof. Therefore, a digression is made in the next three Sections to introduce these concepts all at once; following this digression the proof of the functionality theorem is resumed and completed.

Facts and Nomenclature about Augmented Arrays

Let A be the augmented array of a run R. Recall that if $\#A_{ij}$, i.e., if A_{ij} is defined, then

$$A_{ij} = \langle a, b, c \rangle$$

Here if $j = 0$ then a is the initial content of cell i, and $b = \phi$, and $c = \phi$. If $j > 0$ then a equals H_{ij}, and A_{bc} "writes" A_{ij}. It will be convenient to use the notation

$$_1A_{ij} = a$$
$$_2A_{ij} = b$$
$$_3A_{ij} = c$$

This notation, although unconventional, will prove much less cumbersome than, say, $(A_{ij})_1$, $(A_{ij})_2$, and $(A_{ij})_3$.

The quantity ${}_1A_{ij}$ is the j-th-written content of cell i. The zeroth-written content of cell i is the initial content of cell i. The quantity ${}_2A_{ij}$ is the name of the cell that is the j-th-writer of cell i. It is meaningless to speak of the 0-th-writer of cell i. The quantity ${}_3A_{ij}$ is the write ordinality of the j-th-writer of cell i. It is meaningless to speak of the write ordinality of the 0-th-writer of cell i.

Every time cell i performs a transaction, cell i writes into itself. Therefore, if

$${}_2A_{i,j+1} = i$$

i.e., if the (j + 1)-st-writer of cell i is cell i itself, then during the run R cell i is known to receive a go pulse at an instant when cell i has been written into j times. Then

$${}_3A_{i,j+1} = j$$

and position $\langle i, j \rangle$ of A is said to be an executed position of A, and A_{ij} is said to be an executed element of A.

If A_{ij} is an executed element of A, then there is a transaction associated in a natural way with the element A_{ij}, namely, the transaction corresponding in cell i's transaction table to the word ${}_1A_{ij}$. If and only if this transaction is such that cell i requires positive count for a cell x in order to be enabled to perform the transaction, then A_{ij} is said to be an x-requiring element of A. Notice that a statement that A_{ij} is x-requiring implies that A_{ij} is executed.

If A_{ij} is executed, then A_{ij} is i-requiring. Thus, if A_{ij} is x-requiring, then A_{ij} is i-requiring. Furthermore, if A_{ij} is x-requiring, and x is not equal to i, then the transaction associated with A_{ij} is either a get of x, a put of x, or a send of x.

An x-requiring element A_{ij} may be either x-reading, x-writing, or both, or neither. An x-requiring element A_{ij} is x-reading or x-writing if and only if the transaction associated with A_{ij} reads or writes, respectively, cell x. Notice that a statement that A_{ij} is x-reading or x-writing implies that A_{ij} is x-requiring, which in turn implies that A_{ij} is executed.

If A_{ij} is x-reading and x is not equal to i, then the transaction associated with A_{ij} is a get of x. If A_{ij} is x-writing and x is not equal to i, then the transaction associated with A_{ij} is a put of x. If A_{ij} is either i-reading or i-writing, then the transaction associated with A_{ij} might be any of the five types. If A_{ij} is both x-reading and x-writing, then x is equal to i, and the transaction associated with A_{ij} might be any of the five types. Finally, if the x-requiring element A_{ij} is neither x-reading nor x-writing, then the transaction associated with A_{ij} is a send of x.

If and only if $A_{ij} = \langle w, x, y \rangle$, then A_{xy} is i-writing, and A_{xy} is said to write A_{ij}. Similarly, if and only if during the run described by A, a cell x reads a cell i after exactly y and j writes have occurred into the cells x and i, respectively, then A_{xy} is said to read A_{ij}.

Boundaries

To isolate in an augmented array those elements having properties of interest, use will be made of boundaries, which are described by boundary vectors. As shown in Figure 4.3, a boundary is an imaginary line drawn between the elements of an array so that each row of the array is crossed exactly once. A boundary vector B describes a boundary in an array A by giving the number of elements to the left of the boundary in each row of A. Thus if $B_i = j$, then A_{ij} is the element just to the right of the B boundary in row i of A.

Associated with each augmented array is exactly one edge boundary, which is described by a boundary vector called the edge vector of the array. As shown in Figure 4.4, the edge vector E of an array A is defined for each cell i in the following way.

$$E_i = \text{maximum } j \text{ such that } \#A_{ij}$$

That is, the elements of E point to the right-most elements of A, and the E boundary falls just to the left of these elements. Thus, if A is the augmented array of some run, R, then for each cell i, ${}_1A_{iE_i}$ is the content of cell i at the conclusion of the computation denoted by R. An important property of the edge vector E of an augmented array A is that every executed position in A lies to the left of the E boundary.

The Count Matrix at the Conclusion of a Prefix Run

Let A be the augmented array of a run R. It has been shown how one can extract information from A on a small scale; that is, it has been shown how one can identify elements of A that are x-requiring,

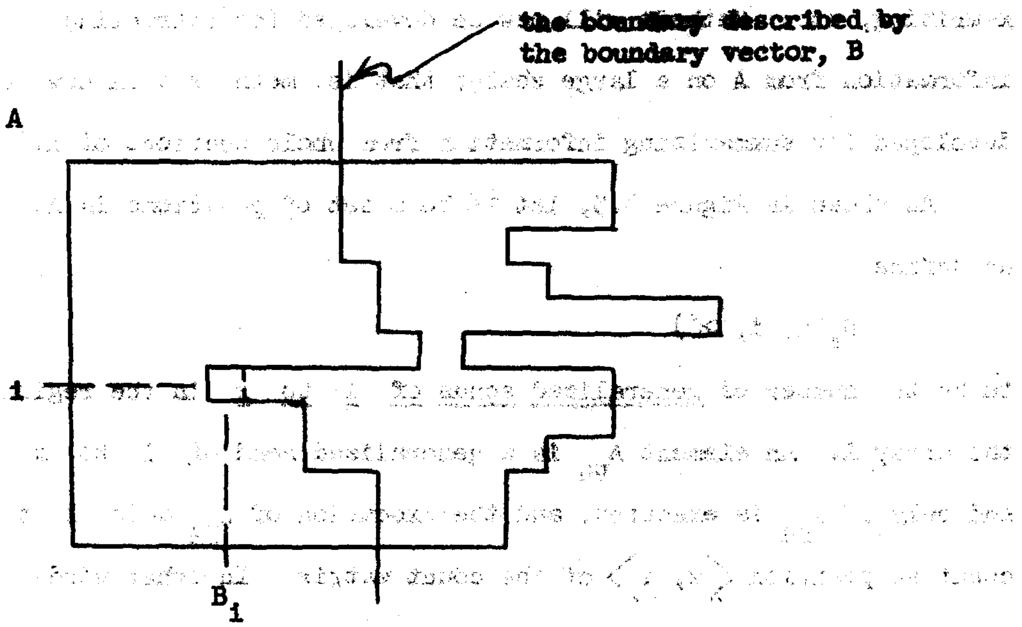


Figure 4.3. A boundary.

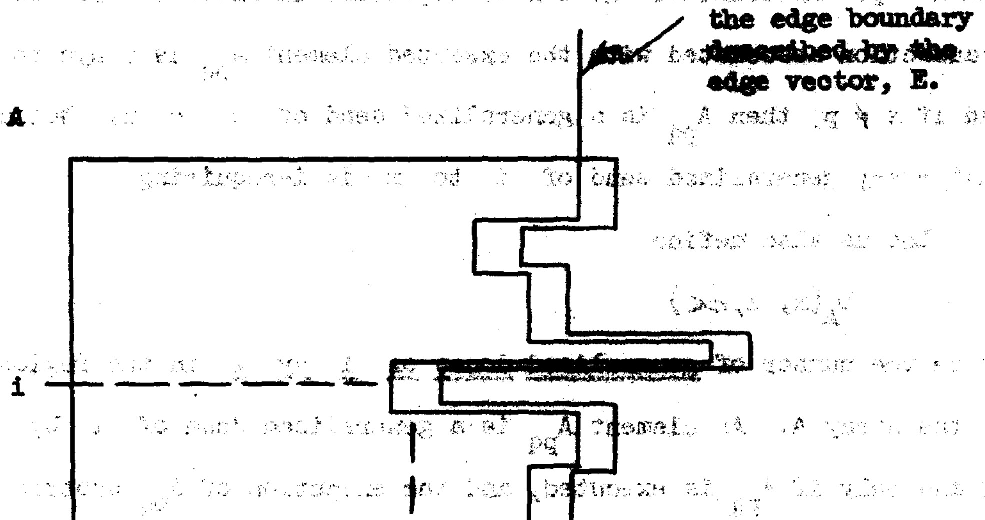


Figure 4.4. An edge boundary.

x-writing, etc. Methods will now be developed for extracting information from A on a large scale; that is, methods will now be developed for summarizing information from whole sections of A.

As shown in Figure 4.5, let α be a set of positions in A. Let us define

$$N_A(x, i, \alpha)$$

to be the number of <u>generalized</u> <u>sends</u> <u>of</u> <u>i</u> <u>to</u> <u>x</u> in the region α of the array A. An element A_{pq} is a generalized send of i to x if and only if A_{pq} is executed, and the execution of A_{pq} adds 1 to the count at position $\langle x, i\rangle$ of the count matrix. In other words, A_{pq} is a generalized send of i to x if and only if A_{pq} is executed and either (1) the transaction associated with A_{pq} is a send of i to x, or (2) the transaction associated with A_{pq} is a bye to x, and $x \neq p$, and $i = p$. Alternative (2) can be explained in another way: if the transaction associated with the executed element A_{pq} is a bye to x, and if $x \neq p$, then A_{pq} is a generalized send of p to x. Notice that every generalized send of i to x is i-requiring.

Let us also define

$$D_A(x, i, \alpha)$$

to be the number of <u>generalized</u> <u>dones</u> <u>of</u> <u>i</u> <u>by</u> <u>x</u> in the region α of the array A. An element A_{pq} is a generalized done of i by x if and only if A_{pq} is executed, and the execution of A_{pq} subtracts 1 from the count at position $\langle x, i\rangle$ of the count matrix; it is a property of the MCM design that A_{pq} can be a generalized done of i by x only if $x = p$. In other words, A_{pq} is a generalized done of i by x if and only if $x = p$, and A_{pq} is executed, and either (1) the

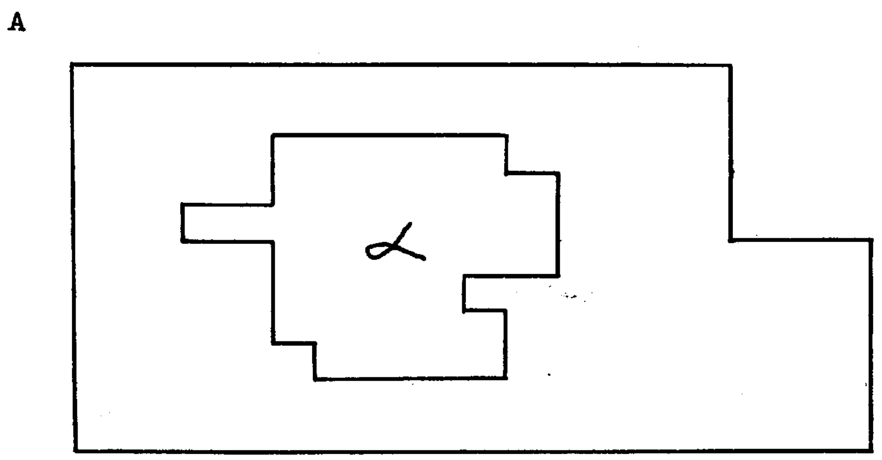


Figure 4.5. A set of positions in an augmented array.

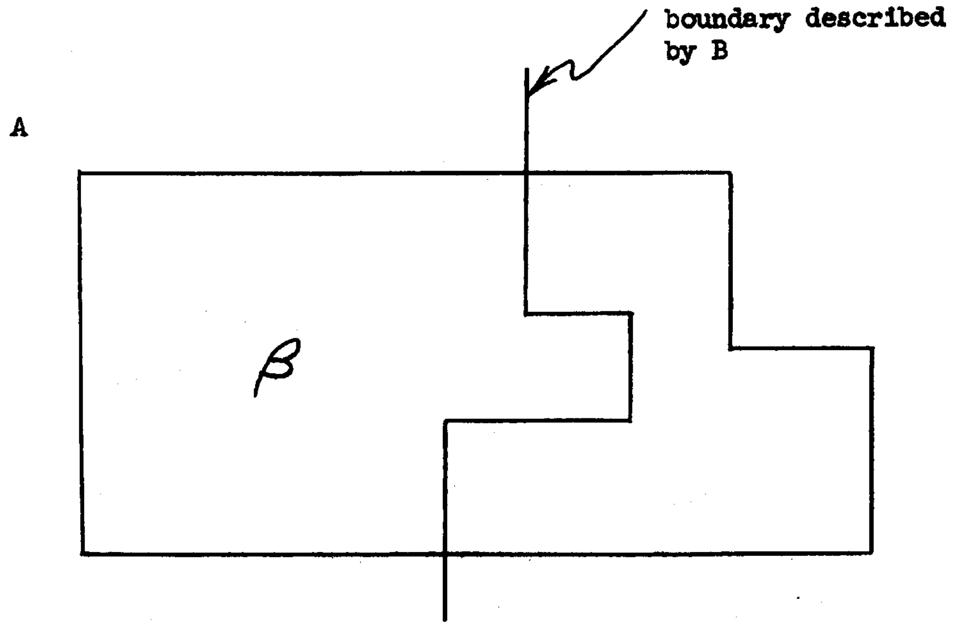


Figure 4.6. A boundary and the set of positions to its left.

transaction associated with A_{pq} is a done of i, or (2) the transaction associated with A_{pq} is a bye to e, and $e \neq x = p$, and $i = x = p$. Alternative (2) can be explained in another way: if the transaction associated with the executed element A_{pq} is a bye to e, and if $e \neq p$, then A_{pq} is both a generalized send of p to e, and a generalized done of p by p. This situation, in which the transaction associated with the executed element A_{pq} is a bye to e, and $e \neq p$, is the only situation in which an element A_{pq} is both a generalized send and a generalized done.

As shown in Figure 4.6, let B be a boundary vector describing a boundary in A, and let β be the set of positions lying to the left of the B boundary. Recall that A is the augmented array of a run R, and let $k(\cdot, \cdot)$ describe the state of the count matrix in the initial computation state of R. Let us define

$$K_R(x, i, \beta)$$

so that

$$K_R(x, i, \beta) = k(x, i) + N_A(x, i, \beta) - D_A(x, i, \beta) \qquad (4.9)$$

The significance of the quantity $K_R(x, i, \beta)$ is the following. Suppose there is some run P that is a prefix of R and that has an augmented array whose edge vector is B. Then $K_R(x, i, \beta)$ gives the count at position $\langle x, i \rangle$ of the count matrix at the conclusion of the run P.

An understanding of the above remark about $K_R(x, i, \beta)$ is vitally important for an adequate comprehension of the proof of the functionality theorem. Let us therefore review that import of (4.9). Consider an instant between two successive computation state transitions in the computation denoted by the run R. Let machine M's behavior up to this instant be described by a run P that is a prefix of R. If the edge vector of the augmented array of P is B, and if, as in Figure 4.6, the B boundary is drawn in the array A, then to every transaction performed during P there corresponds a unique executed element to the left of the B boundary, i.e., in the region β. Some of these elements in β might be generalized sends of i to x, some might be generalized dones of i by x, some might be both, and some might be neither. Consider the count at position $\langle x, i \rangle$ of the count matrix; equation (4.9) says that what this count is at the conclusion of P equals what this count was at the start of P, plus the number of generalized sends of i to x performed during P, minus the number of generalized dones of i by x performed during P.

Resuming the Proof

The digression introducing auxiliary concepts is now complete. Let us continue with the proof of the functionality theorem. Recall that the inductive step's second case, depicted in Figure 4.2, is being discussed. This case is the second case in the establishing of (4.8), i.e., the case in which position $\langle i, j \rangle$ is not defined in A. The quantities R, R', R^+, A, A', A^+, i, and j as used now are the same quantities that were used when the proof was begun. These quantities will remain bound in this way for the remainder of the proof. For the remainder of the proof, also, let us define x, y, z, and w so that A'_{xy} is the element of A' that writes A'_{ij}, and so that A^+_{zw} is the element of A^+ that writes A^+_{ij}. For convenience, the remainder of the proof is divided into three stages:

(1) a proof that $\langle x, y\rangle = \langle z, w\rangle$,

(2) a proof that if the transaction associated with A'_{xy} is a put, send, done, or bye, then $A'_{ij} = A^+_{ij}$, and

(3) a proof that if the transaction associated with A'_{xy} is a get, then $A'_{ij} = A^+_{ij}$.

Before the first stage of the proof is begun, several quantities will be introduced that will be useful in all three stages. The first quantity is the array A^o. The array A^o has the form of an augmented array, although there may not exist a run whose augmented array is A^o. The array A^o is defined everywhere that either A or A^+ is defined, and nowhere else. If $\#A^o_{pq}$ then A^o_{pq} equals either A_{pq}, or A^+_{pq}, or both; since $A \cong A^+$, then there is no ambiguity in saying that A^o_{pq} is equal to both A_{pq} and A^+_{pq}.* A convenient way of understanding the construction of A^o is to consider that an array such as A is a set of ordered pairs of the form

$$\langle\langle i, j\rangle, A_{ij}\rangle$$

and observe that A^o is the set-theoretic union** of A and A^+.

Let E, E', E^+, and E^o be the edge vectors of the arrays A, A', A^+, and A^o. Let R^- be the prefix run of R^+ whose performance immediately precedes the writing of A^+_{ij}. Let A^- be the augmented array of R^-, and

*The usefulness of working with the array A^o was pointed out by Prof. J. B. Dennis.

**The union of $\{a, b\}$ and $\{a, c\}$ is $\{a, b, c\}$.

let E^- be the edge vector of A^-. Figure 4.7 shows the relationship of the run R^- to the run R^+, and the relationship of the run R to the run R'.

Figure 4.8 shows the E and E^- boundaries drawn in A^o. During the computation state transition of R' that immediately follows the performance of R, A'_{ij} is written. Therefore $\#A_{i,j-1}$ and* $\neg \#A_{ij}$, and so the E boundary falls just to the left of $A^o_{i,j-1}$. Similar reasoning shows that the E^- boundary also falls just to the left of $A^o_{i,j-1}$.

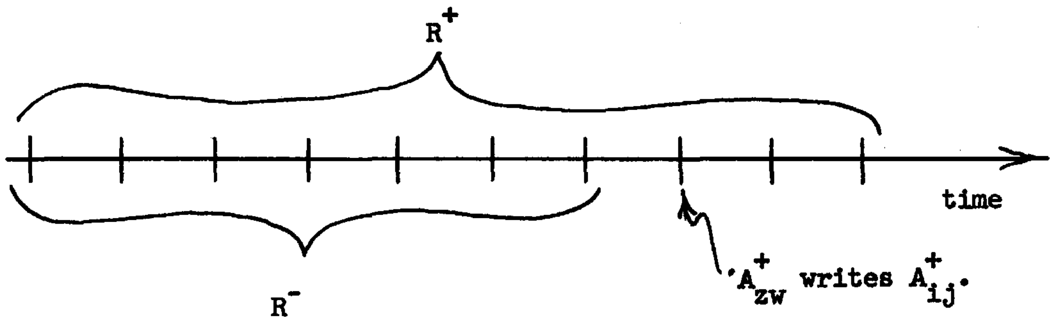
As shown in Figure 4.8, let α be the set of positions in A^o to the left of both the E and E^- boundaries. Let β be the set of positions to the right of the E boundary and to the left of the E^- boundary. Let γ be the set of positions to the left of the E boundary and to the right of the E^- boundary.

The First Stage

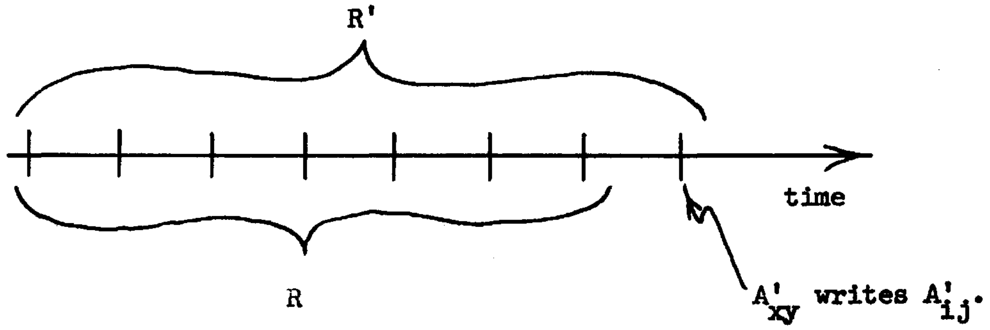
The object of the first stage of the proof of the second case of the inductive step is to show that A^o_{xy} and A^o_{zw} are one and the same element. For convenience, the reasoning is divided into (1) an enumeration of possibilities, and (2) five explicitly indicated tasks.

Enumeration of Possibilities. During the computation state transition of R' that immediately follows the performance of R, A'_{xy} is executed. Therefore $\#A_{xy}$ and $\neg \#A_{x,y+1}$, and so, as shown in Figure 4.9, A^o_{xy} lies just to the right of the E boundary. Similar reasoning shows that A^o_{zw} lies just to the right of the E^- boundary.

*The proposition $\neg A$ is true if and only if A is not true.



(a) Run R^- is a prefix of run R^+.



(b) Run R is a prefix of run R'.

Figure 4.7. Four runs. Hack marks denote computation state transitions, assumed here to be instantaneous.

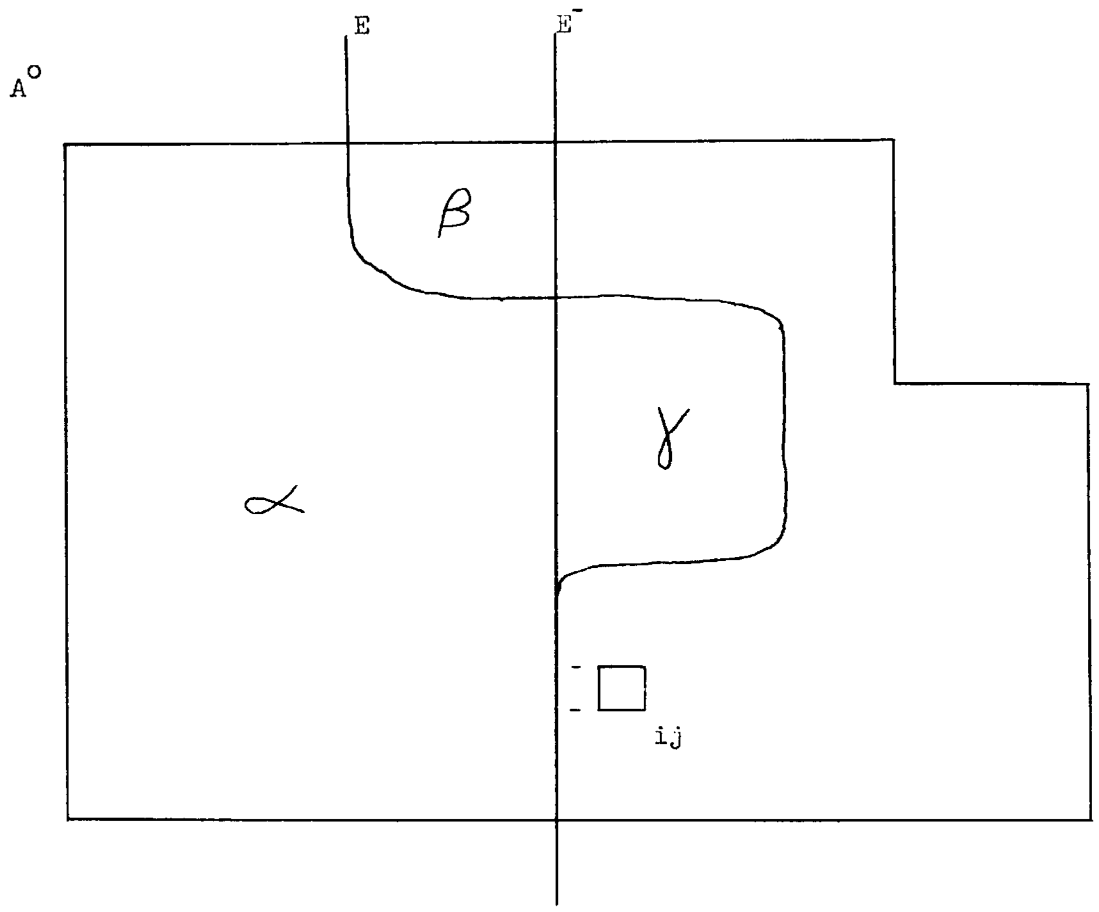


Figure 4.8. Principal boundaries used in the proof of the second case of the inductive step.

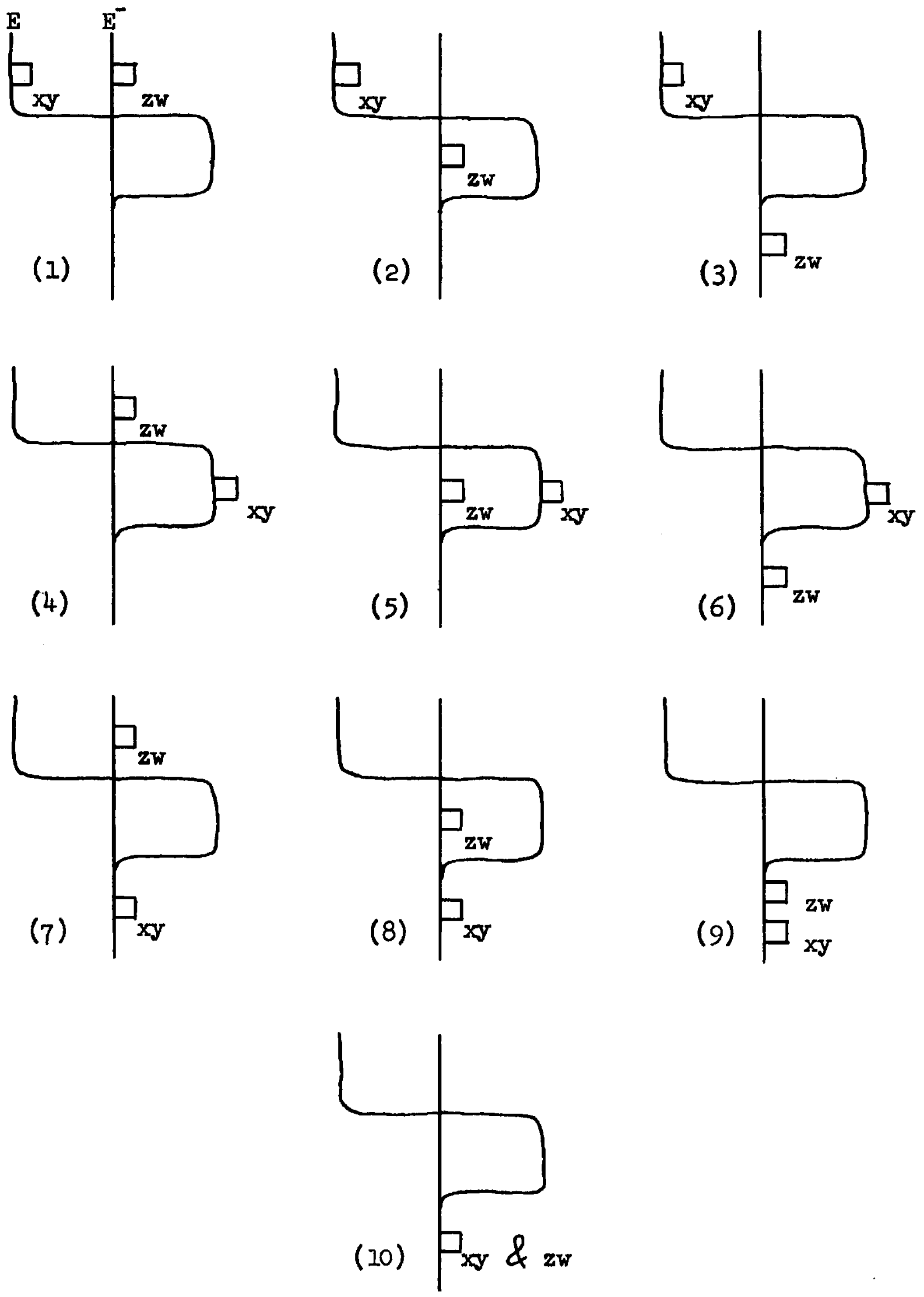


Figure 4.9. The ten possibilities in the proof of the first stage of the second case of the inductive step.

Figure 4.9 shows that element A^{o}_{xy} might lie either to the left of the E^{-} boundary, or just to the right of the E^{-} boundary, or at least one position away from and to the right of the E^{-} boundary. For each of these three possibilities, exactly the same three possibilities exist for A^{o}_{zw} with respect to the E boundary, making a total of nine possibilities. For one of these possibilities, namely, that in which A^{o}_{xy} and A^{o}_{zw} lie just to the right of the E^{-} and E boundaries, respectively, A^{o}_{xy} and A^{o}_{zw} might or might not be one and the same element. For the other eight possibilities, it is clear that A^{o}_{xy} and A^{o}_{zw} cannot be one and the same element. Thus there are ten possibilities in all, which are depicted in Figure 4.9. The job of the first stage of the second case of the inductive step is to disprove the first nine of these possibilities.

Task One. The first task is to disprove possibilities (2), (5), and (8) of Figure 4.9. Let us restrict our attention to these three possibilities, and recall that A^{+}_{zw} is executed. It is not known, however, that A_{zw} is executed, and an argument establishing this fact will now be presented. This argument will be given in much greater detail than will be usual in the rest of the proof, in order to allow the reader to become accustomed to the notation being used, and in order to help the reader's intuition develop along sound lines.

Since for possibilities (2), (5), and (8), A^{o}_{zw} lies to the left of the E boundary, therefore A_{zw} lies to the left of the E boundary. Since one element of A lies to the right of the E boundary, then at least one element of A must lie to the right of A_{zw}. Therefore, $\#A_{z,w+1}$.

Since A^+_{zw} is executed, then $\#A^+_{z,w+1}$ and ${}_2A^+_{z,w+1} = z$. By the inductive hypothesis we have $A \cong A^+$, and therefore $A_{z,w+1} = A^+_{z,w+1}$. Therefore ${}_2A_{z,w+1} = z$, and so A_{zw} is executed.

Digression. In order that the reader's intuition will not be led astray by the simplicity of the foregoing argument, a digression will now be made to alert the reader to some of the subtlties of the problem of proving the inductive step. Consider possibility (1) of Figure 4.9. It will be shown later that for possibility (1) A^-_{xy} is executed. It would seem that this fact could be established by relying on an apparent symmetry between possibilities (1) and (5). Suppose one tried to establish this fact, that A^-_{xy} is executed, by an argument similar to the argument which established for possibility (5) that A_{zw} is executed. The new argument would be constructed from the old one by interchanging x and y with z and w, respectively, by interchanging R^-, A^-, and E^- with R, A, and E, respectively, and by interchanging R^+, A^+, and E^+ with R', A', and E', respectively. The new argument would be valid up to the point at which it invokes the "inductive hypothesis" that $A^- \cong A'$. The proposition $A^- \cong A'$ is not implied by the inductive hypothesis, (4.6). In fact, since $A^- \subseteq A^+$, therefore $A^- \cong A'$ is implied by the fact that $A^+ \cong A'$, which is the inductive conclusion, (4.7). Thus one would have wandered into the time-honored pitfall of invoking that which one was trying to prove.

The notion given to us by untutored intuition of a symmetry between possibilities (1) and (5) has proved false. Indeed, there is an inherent asymmetry in the environment of the inductive step, as shown in Figure 4.10. Thus armed with a better understanding of the

$$\begin{tikzcd}
A^{-} \arrow[r, dash, "\subseteq"] \arrow[d, dash, "\cong"'] \arrow[dr, dash, dashed, "\cong ?"] & A^{+} \arrow[d, dash, dashed, "\cong ?"] \\
A \arrow[r, dash, "\subseteq"'] \arrow[ur, dash, "\cong"] & A'
\end{tikzcd}$$

Figure 4.10. Asymmetry in the environment of the inductive step. Solid lines represent known relationships; dashed lines represent relationships to be proved.

problem, let us resume the reasoning of the first stage. Recall that we have just established for possibilities (2), (5), and (8) of Figure 4.9 that A_{zw} is executed. The presentation will continue to be more detailed than will be usual.

<u>Resumption</u> <u>of</u> <u>Task</u> <u>One</u>. When A^+_{zw} writes A^+_{ij}, a certain transaction, call it θ, is performed. By the inductive hypothesis $A \cong A^+$, we have $A_{zw} = A^+_{zw}$. Therefore when A_{zw} is executed, the same transaction θ is performed. Since the execution of A^+_{zw} writes cell i, then the execution of A_{zw} writes cell i. Since $\neg \# A_{ij}$, then A_{zw} writes A_{iv} for some $v < j$. Thus ${}_2A_{iv} = z$ and ${}_3A_{iv} = w$. Since $\# A^+_{ij}$, then $\# A^+_{iv}$. By $A \cong A^+$ we have $A_{iv} = A^+_{iv}$. Therefore ${}_2A^+_{iv} = z$ and ${}_3A^+_{iv} = w$. Therefore A^+_{zw} writes A^+_{iv}. But by construction A^+_{zw} writes A^+_{ij}, and we know $v < j$. This contradiction has been obtained from the assumption that either possibility (2), or (5), or (8) of Figure 4.9 prevails. Therefore possibilities (2), (5), and (8) have been eliminated.

<u>Task</u> <u>Two</u>. The second task is to establish a result that does not, by itself, eliminate any of the possibilities of Figure 4.9, but that is rather just a useful intermediate result. This result is that there are no i-requiring elements of A^o whose positions belong to γ; recall that the region γ has been defined as indicated in Figure 4.8. The result will be shown by assuming the contrary and deriving a contradiction. Any i-requiring element in γ must be executed in run R. Let A^o_{pq} be an arbitrary i-requiring element such that A_{pq} is one of the first i-requiring elements in γ to be executed in R. That is, of perhaps

several i-requiring elements in γ that are executed simultaneously in R before any others, A_{pq} is an arbitrary one.

Let R^p be the prefix run of R whose performance immediately precedes the execution of A_{pq}; let A^p be the history array of R^p, and let E^p be the edge vector of A^p. Figure 4.11 shows the E^p boundary drawn in A^o. Notice that since R^p is a prefix of R, the E^p boundary lies everywhere on or to the left of the E boundary.

As shown in Figure 4.11, let ρ be the set of positions in A^o to the left of both the E^p and E^- boundaries. Let σ be the set of positions to the right of the E^p boundary and to the left of the E^- boundary. Let τ be the set of positions to the left of the E^p boundary and to the right of the E^- boundary. Since A^o_{pq} is i-requiring, then at the conclusion of R^p, cell p has read capability for cell i, and so according to (4.9),*

$$K(p, i, \rho \cup \tau) > 0 \qquad (4.10)$$

Digression. In (4.10), as throughout the remainder of the proof of the functionality theorem, the subscripts of K, N, and D are omitted. No ambiguity arises from the omission of the array subscript on N and D, because this subscript may always be taken to be A^o. No ambiguity arises from the omission of the run subscript on K, because the initial count matrix is the same for all runs being discussed.

Resumption of Task Two. Since A_{pq} is one of the first i-requiring elements in γ to be executed in R, and since any i-requiring element

*The set $A \cup B$ is the union of A and B.

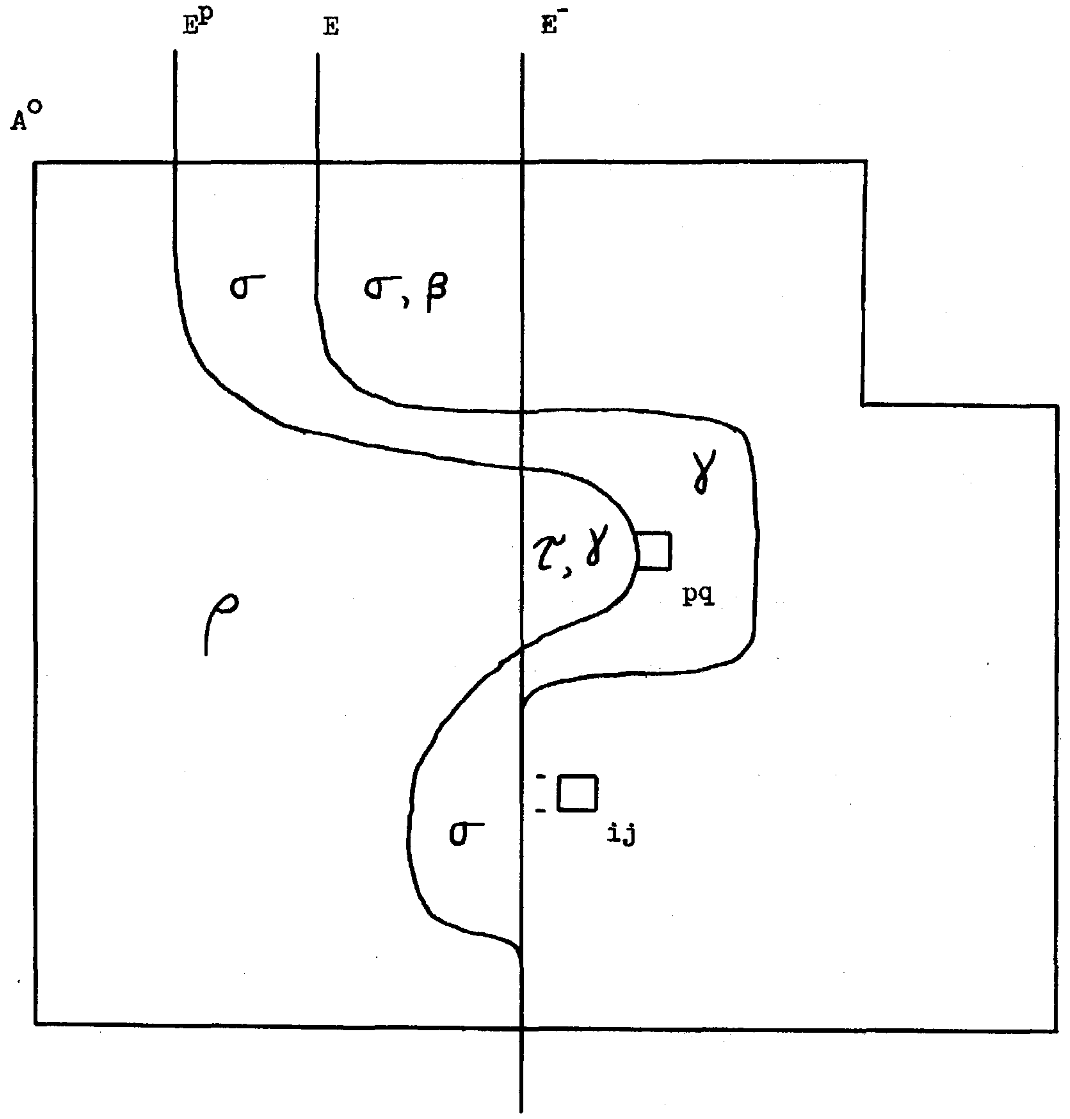


Figure 4.11. Boundaries used in the demonstration that there are no i-requiring elements in γ.

of A lying in τ is executed in R before A_{pq} is executed, therefore there are no i-requiring elements in τ. Since every generalized send of i to p is i-requiring, therefore

$$N(p, i, \tau) = 0$$

and so

$$N(p, i, \rho) = N(p, i, \rho \cup \tau)$$

Clearly

$$N(p, i, \rho \cup \sigma) \geqslant N(p, i, \rho)$$

and so

$$N(p, i, \rho \cup \sigma) \geqslant N(p, i, \rho \cup \tau) \tag{4.11}$$

Since generalized dones of i by p occur only in row p of A^o, and since in row p of A^o the E^p boundary lies on or to the right of the E^- boundary, then

$$D(p, i, \rho \cup \sigma) \leqslant D(p, i, \rho \cup \tau) \tag{4.12}$$

Combining (4.10), (4.11), and (4.12) with the definition of K, (4.9), we have

$$K(p, i, \rho \cup \sigma) > 0$$

This last proposition says that at the conclusion of R^-, cell p has read capability for cell i.

As indicated in Figure 4.7, the run R^- is the prefix of R^+ that just precedes the execution of A^+_{zw}. By construction, the element A^+_{zw} writes A^+_{ij}, and so at the conclusion of R^-, cell z has write capability, i.e., sole read capability, for cell i. Therefore, by the above result, z = p; thus A^o_{zw} lies in row p of A^o. Since A^o_{pq} was constructed to lie in γ, then in row p, the E boundary lies to the right of the E^- boundary. Since A^o_{zw} lies in row p and just to the right of the E^- boundary, then A^o_{zw} lies in γ. Therefore only possibilities (2), (5), and (8) of Figure 4.9 can prevail. But these possibilities have already been eliminated. This contradiction shows that there are no i-requiring elements in γ.

<u>Task Three</u>. The third task is to disprove possibilities (4), (6), (7), and (9) of Figure 4.9. Recall that the regions α and β have been defined as indicated in Figure 4.8. At the conclusion of R, cell x has write capability for cell i, and so

$$K(x,\ i,\ \alpha \cup \gamma\ ) > 0$$

Since there are no i-requiring elements in γ,

$$N(x,\ i, \alpha\ ) = N(x,\ i,\ \alpha \cup \gamma\ )$$

Clearly

$$N(x,\ i,\ \alpha \cup \beta\ ) \geqslant N(x,\ i, \alpha)$$

and so

$$N(x,\ i, \alpha \cup \beta\ ) \geqslant N(x,\ i, \alpha \cup \gamma)$$

For possibilities (4), (6), (7), and (9), in row x of A^o the E boundary lies on or to the right of the E^- boundary. Therefore

$$D(x,\ i, \alpha \cup \beta\ ) \leqslant D(x,\ i, \alpha \cup \gamma)$$

Thus

$$K(x,\ i, \alpha \cup \beta\ ) > 0$$

which says that at the conclusion of run R^-, cell x has read capability for cell i.

At the conclusion of R^-, cell z has sole read capability for cell i. Therefore x = z. But since $x \neq z$ in possibilities (4), (6), (7), and (9), therefore possibilities (4), (6), (7), and (9) have been eliminated. Only possibilities (1) and (3) of Figure 4.9 remain to be eliminated.

Task Four. The next task is to show that in possibilities (1) and (3) of Figure 4.9, A^{-}_{xy} is executed. Since A^{o}_{xy} lies to the left of the E^{-} boundary, we have $\#A^{-}_{x,y+1}$. Let A^{-}_{ab} be the element of A^{-} that writes $A^{-}_{x,y+1}$. The task is to show that $\langle a, b\rangle = \langle x, y\rangle$.

Since A^{-}_{ab} is executed, then A^{o}_{ab} lies to the left of the E^{-} boundary. As shown in Figure 4.12, A^{o}_{ab} might lie in either α or β. It will now be shown that A^{o}_{ab} does not lie in α by assuming the contrary and deriving a contradiction. By construction, A^{-}_{ab} is executed, and so $\#A^{-}_{a,b+1}$, and ${}_2A^{-}_{a,b+1} = a$. The assumption that A^{o}_{ab} lies in α implies $\#A_{a,b+1}$. Since $A \cong A^{-}$, therefore both ${}_2A^{-}_{a,b+1} = a$ and $A_{ab} = A^{-}_{ab}$. Thus, since A^{-}_{ab} is x-writing, then A_{ab} is x-writing. Since $\neg\#A_{x,y+1}$, therefore A_{ab} writes A_{xv} for some $v < y+1$. Therefore ${}_2A_{xv} = a$ and ${}_3A_{xv} = b$. We have $\#A^{-}_{xv}$, because $v \leq y$ and $\#A^{-}_{xy}$. Since $A \cong A^{-}$ then ${}_2A^{-}_{xv} = a$ and ${}_3A^{-}_{xv} = b$. Therefore A^{-}_{ab} writes A^{-}_{xv}. Since $v < y+1$, then A^{-}_{ab} does not write $A^{-}_{x,y+1}$. But by construction A^{-}_{ab} does write $A^{-}_{x,y+1}$. Therefore A^{o}_{ab} does not lie in α.

We now know that A^{o}_{ab} lies in β. By construction A^{-}_{ab} writes $A^{-}_{x,y+1}$. Therefore A^{o}_{ab} is x-requiring, and so there is at least one x-requiring element in β. Any x-requiring element in β must be executed in run R^{-}. Let A^{o}_{pq} be an arbitrary x-requiring element such that A^{-}_{pq} is one of the first x-requiring elements in β to be executed in R^{-}. It will now be shown that $p = x$.

Let R^{p} be the prefix run of R^{-} immediately preceding the execution of A^{-}_{pq}. Let A^{p} be the augmented array of R^{p}, and let E^{p} be the edge vector of A^{p}. Figure 4.13 shows the E^{p} boundary drawn in A^{o}. Since R^{p} is a prefix of R^{-}, the E^{p} boundary lies everywhere on or to the left of

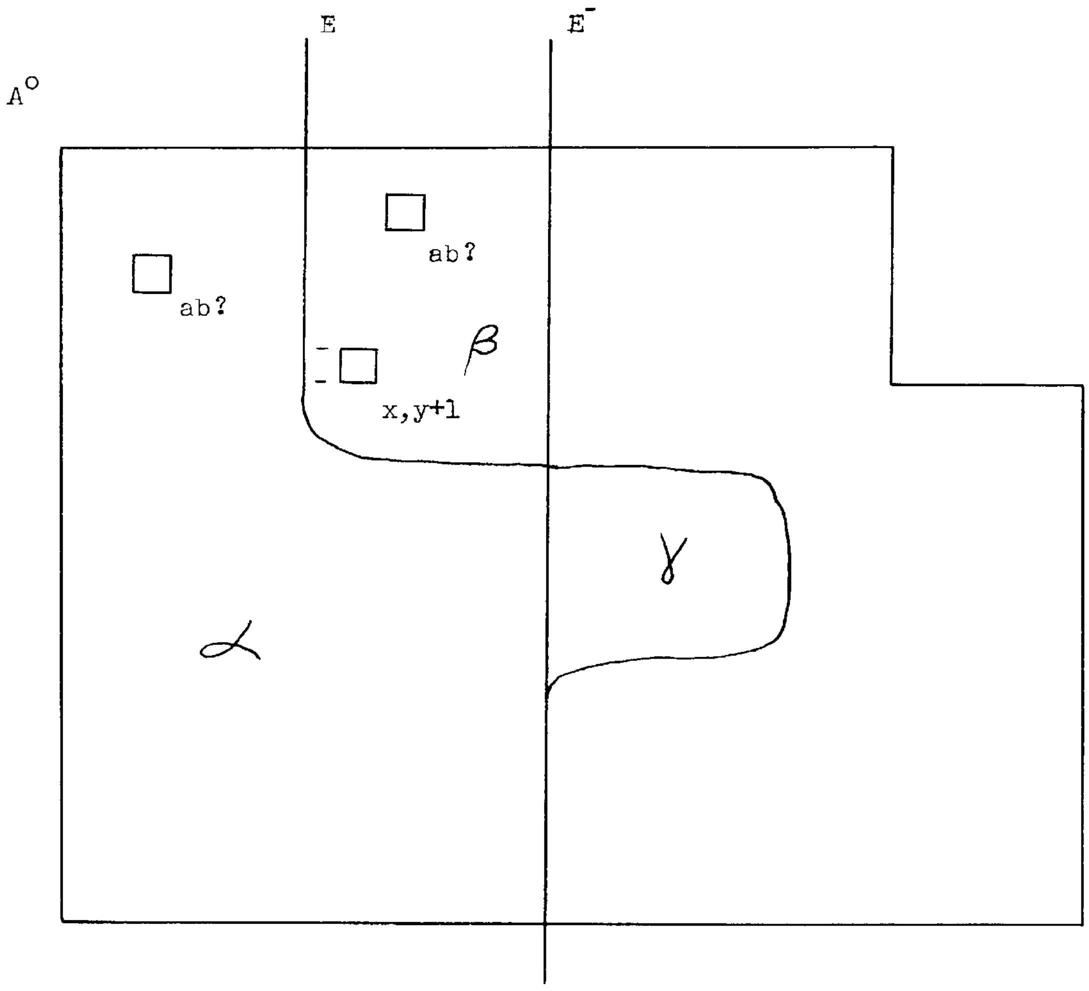


Figure 4.12. Possible locations of A^o_{ab}.

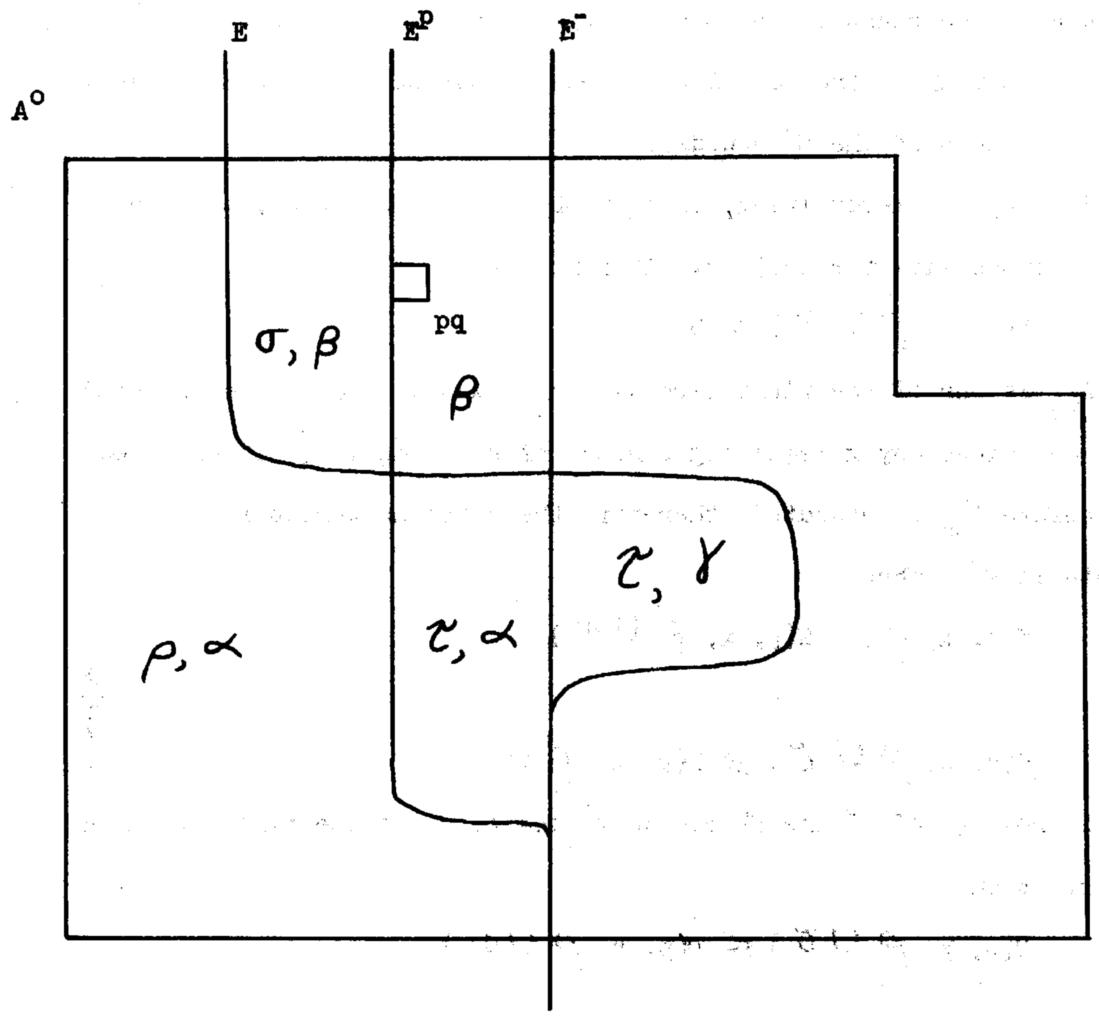


Figure 4.13. Boundaries used in the demonstration for possibilities (1) and (3) that A^-_{xy} is executed.

the E^- boundary. As shown in Figure 4.13, let ρ be the set of positions to the left of both the E and E^p boundaries. Let σ be the set of positions to the right of the E boundary and to the left of the E^p boundary. Let τ be the set of positions to the left of the E boundary and to the right of the E^p boundary.

Since A^-_{pq} is x-requiring, then at the conclusion of R^p, cell p has read capability for cell x. Therefore

$$K(p, x, \rho \cup \sigma) > 0$$

Since A^-_{pq} is one of the first x-requiring elements in β to be executed in R^-, and since any x-requiring element of A^- lying in σ is executed in R^- before A^-_{pq} is executed, therefore there are no x-requiring elements in σ. Thus

$$N(p, x, \rho) = N(p, x, \rho \cup \sigma)$$

and so

$$N(p, x, \rho \cup \tau) \geqslant N(p, x, \rho \cup \sigma)$$

Since in row p of A^o the E^p boundary lies on or to the right of the E boundary, then

$$D(p, x, \rho \cup \tau) \leqslant D(p, x, \rho \cup \sigma)$$

Thus

$$K(p, x, \rho \cup \tau) > 0$$

and so at the conclusion of R, cell p has read capability for cell x.

It is known that at the conclusion of R, cell x has sole read capability for itself, because during the computation state transition of R' that immediately follows the run R, cell x, in writing into cell i, also writes into itself. Therefore p = x.

The element A^{-}_{pq} was chosen arbitrarily out of the set of x-requiring elements in β that are executed first in R^{-}. Since $p = x$, therefore all x-requiring elements in β executed first in R^{-} lie in row x of A^{o}. Since only one element of β in row x can be executed first in R^{-}, therefore the set of x-requiring elements in β that are executed first in R^{-} contains only one element, namely A^{-}_{xq}. Suppose $q > y$. Then A^{-}_{ab} writes $A^{-}_{x,y+1}$ before A^{-}_{xq} is executed, and so A^{-}_{ab} is an x-requiring element in β that is executed before the first x-requiring element in β, namely A^{-}_{xq}, is executed. Therefore $q \leq y$. Since A^{-}_{xy} lies just to the right of the E boundary, then $q \geq y$, for otherwise A^{-}_{xq} would not lie in β. Therefore $q = y$, and A^{-}_{xy} is the first x-requiring element in β to be executed in R^{-}. Thus for possibilities (1) and (3) of Figure 4.9 it has been shown that A^{-}_{xy} is executed.

Task _Five_. The next task is to eliminate possibilities (1) and (3) of Figure 4.9. Since A_{xy} is i-writing, then by $A \cong A^{-}$, A^{-}_{xy} is i-writing. Since $\neg \# A^{-}_{ij}$, then A^{-}_{xy} writes A^{-}_{iv} for some $v < j$. Since $A \cong A^{-}$, then A_{iv} is written by A_{xy}. But A_{xy} is not executed. This contradiction eliminates possibilities (1) and (3) of Figure 4.9.

Possibilities (1) through (9) of Figure 4.9 have been eliminated. Therefore $\langle x, y \rangle = \langle z, w \rangle$, and so A'_{xy} writes A'_{ij} and A^{+}_{xy} writes A^{+}_{ij}. This completes the first stage of the proof of the second case of the inductive step.

The Second Stage

The object of the second stage of the proof of the second case of the inductive step is to show that if the transaction associated with

A'_{xy} is a put, send, done, or bye, then $A'_{ij} = A^{+}_{ij}$. Let θ be the transaction associated with A'_{xy}. Since $A \cong A^{+}$, then $A_{xy} = A^{+}_{xy}$. Since $A \subseteq A'$, then $A_{xy} = A'_{xy}$. Therefore $A'_{xy} = A^{+}_{xy}$, and the transaction associated with A^{+}_{xy} is also θ. Consider two alternatives: (a) $x \neq i$, and (b) $x = i$.

If $x \neq i$, then θ must be a put of i. Let v be the operand word of θ. Then the execution of A'_{xy} writes the word v into A'_{ij}. Therefore

$$A'_{ij} = \langle v, x, y \rangle$$

Likewise, the execution of A^{+}_{xy} writes the word v into A^{+}_{ij}. Therefore

$$A^{+}_{ij} = \langle v, x, y \rangle$$

Therefore $A'_{ij} = A^{+}_{ij}$ for alternative (a).

If $x = i$, then θ can be a put, send, done, or bye. Let w be the replacement word of θ. Then the execution of $A'_{i,j-1}$ writes the word w into A'_{ij}; recall that since $\neg\#A_{ij}$ then $j > 0$. Therefore

$$A'_{ij} = \langle w, i, j\text{-}1 \rangle$$

Likewise, the execution of $A^{+}_{i,j-1}$ writes the word w into A^{+}_{ij}. Therefore

$$A^{+}_{ij} = \langle w, i, j\text{-}1 \rangle$$

Therefore $A'_{ij} = A^{+}_{ij}$ for alternative (b). This completes the second stage of the proof of the second case of the inductive step.

The Third Stage

The object of the third and final stage of the proof of the second case of the inductive step is to show that if the transaction associated with A'_{xy} is a get, then $A'_{ij} = A^{+}_{ij}$. Since the transaction θ associated with A'_{xy} is a get, then $x = i$ and $y = j-1$. Thus we have for the third stage that $A'_{i,j-1}$ writes A'_{ij} and that $A^{+}_{i,j-1}$ writes A^{+}_{ij}. Since $A \cong A^{+}$ and $A \subseteq A'$, therefore the transaction θ is associated with both $A'_{i,j-1}$ and $A^{+}_{i,j-1}$. Let the operand name of θ be a. Let b and c be such that $A'_{i,j-1}$ reads A'_{ab} and $A^{+}_{i,j-1}$ reads A^{+}_{ac}. The major job of the third stage is to show that $b = c$. For convenience, the third stage is divided into (1) an enumeration of possibilities, and (2) three explicitly indicated tasks.

Enumeration of Possibilities. Figure 4.14 depicts three possibilities: (1) $b < c$, (2) $b > c$, and (3) $b = c$. Since R is the prefix of R' that immediately precedes the reading of A'_{ab} by $A'_{i,j-1}$, then A^{o}_{ab} lies just to the right of the E boundary. Similar reasoning shows that A^{o}_{ac} lies just to the right of the E^{-} boundary.

Possibility (1) will be disproved first. It will turn out that a proof which eliminates possibility (2) can be constructed from the proof which eliminates possibility (1) by an interchange of notation. This symmetry between possibilities (1) and (2) is due, at least in part, to the fact that whenever $A \cong A^{+}$ would seem to be needed in disproving either possibility (1) or possibility (2), $A \cong A^{-}$ may be invoked instead. As indicated in Figure 4.10, we know $A \cong A^{-}$ because $A \cong A^{+}$ and $A^{-} \subseteq A^{+}$. Thus the difficulty will not be encountered here that was encountered when symmetry was sought between possibilities (1) and (5) in the first stage of the proof of the second case of the inductive step.

Task One. The first task is to disprove possibility (1) of Figure 4.14. For possibility (1), let r and s be such that A^{-}_{rs}

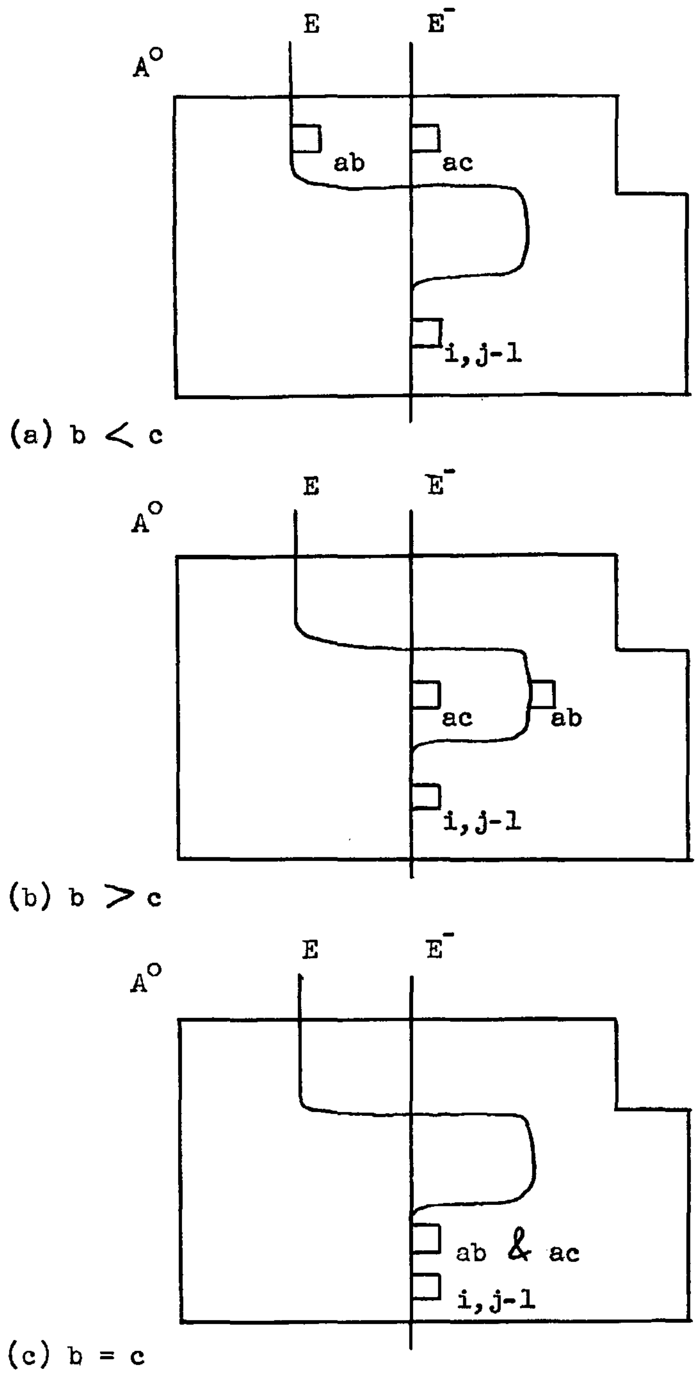


Figure 4.14. The three possibilities in the proof of the third stage of the second case of the inductive step.

writes A^-_{ac}. Since A^-_{rs} is executed, A^o_{rs} lies to the left of the E^- boundary. As shown in Figure 4.15, A^-_{rs} might lie in either α or β.

It will now be shown that A^o_{rs} does not lie in α. This fact will be shown by assuming the contrary and deriving a contradiction. By construction A^-_{rs} is executed, and so $\#A^-_{r,s+1}$, and ${}_2A^-_{r,s+1} = r$. The assumption that A^o_{rs} lies in α implies $\#A_{r,s+1}$. Since $A \cong A^-$, therefore ${}_2A_{r,s+1} = r$. Therefore A_{rs} is executed. Since A^-_{rs} is a-writing, then by $A \cong A^-$, A_{rs} is a-writing. Since $\neg\#A_{a,b+1}$, then A_{rs} writes A_{av} for some $v < b+1$. Therefore ${}_2A_{av} = r$ and ${}_3A_{av} = s$. Since $v \leq b < c$ and $\#A^-_{ac}$, therefore $\#A^-_{av}$ and by $A \cong A^-$, ${}_2A^-_{av} = r$ and ${}_3A^-_{av} = s$. Therefore A^-_{rs} writes A^-_{av}. Since $v \leq b < c$, then A^-_{rs} does not write A^-_{ac}. But by construction A^-_{rs} does write A^-_{ac}. Therefore A^o_{rs} does not lie in α.

As shown in Figure 4.16, A^o_{rs} lies in β. Let R^r be the prefix of R^- immediately preceding the execution of A^-_{rs}. Let A^r be the augmented array of R^r, and let E^r be the edge vector of A^r. Figure 4.16 shows the E^r boundary drawn in A^o. Since R^r is a prefix of R^-, the E^r boundary lies everywhere on or to the left of the E^- boundary.

As shown in Figure 4.17, let λ be the set of positions in A^o to the left of both the E and E^r boundaries. Let μ be the set of positions to the right of the E boundary and to the left of the E^r boundary. Let ν be the set of positions to the left of the E boundary and to the right of the E^r boundary.

It will now be shown that there are no a-requiring elements in ν. This fact will be shown by assuming the contrary and deriving a contradiction. Any a-requiring element in ν must be executed in R.

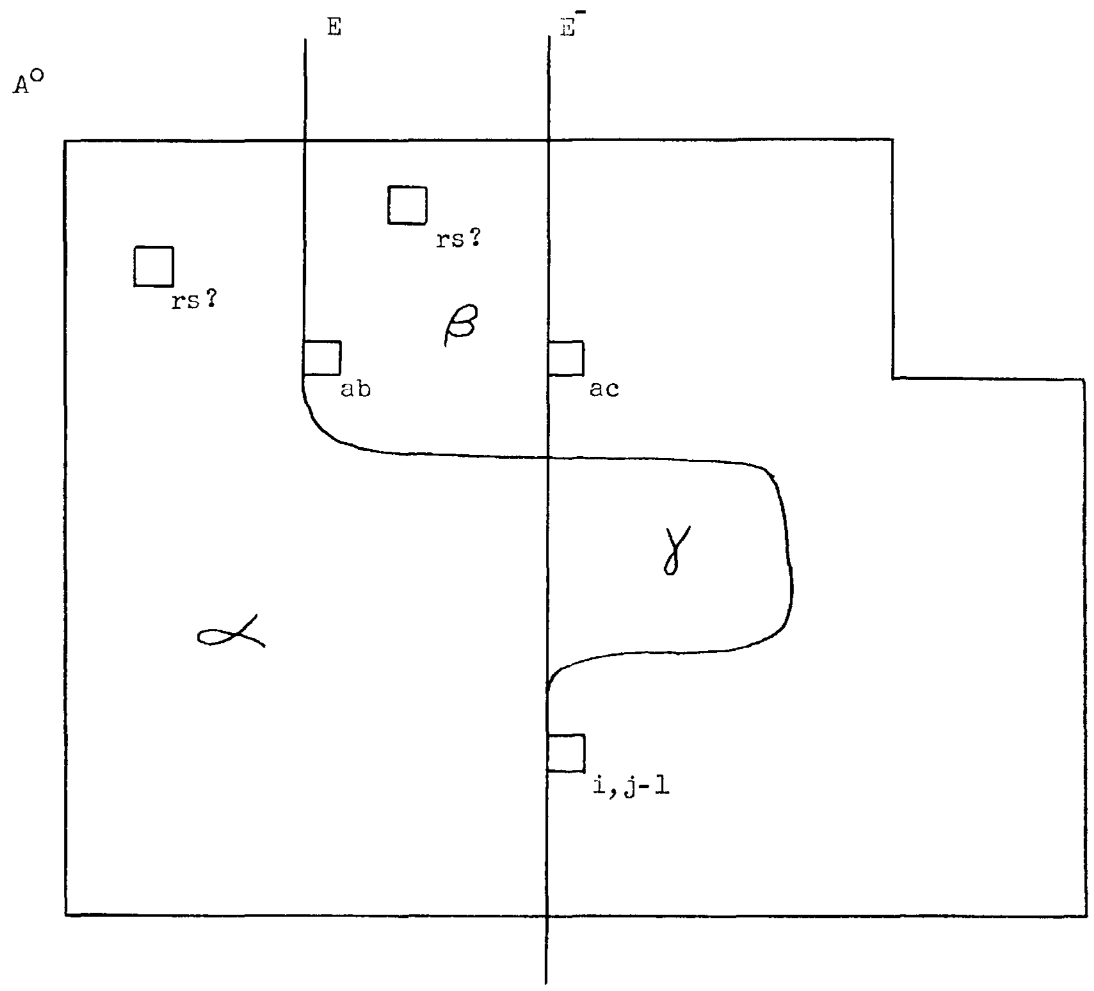


Figure 4.15. Possible locations of A^o_{rs}.

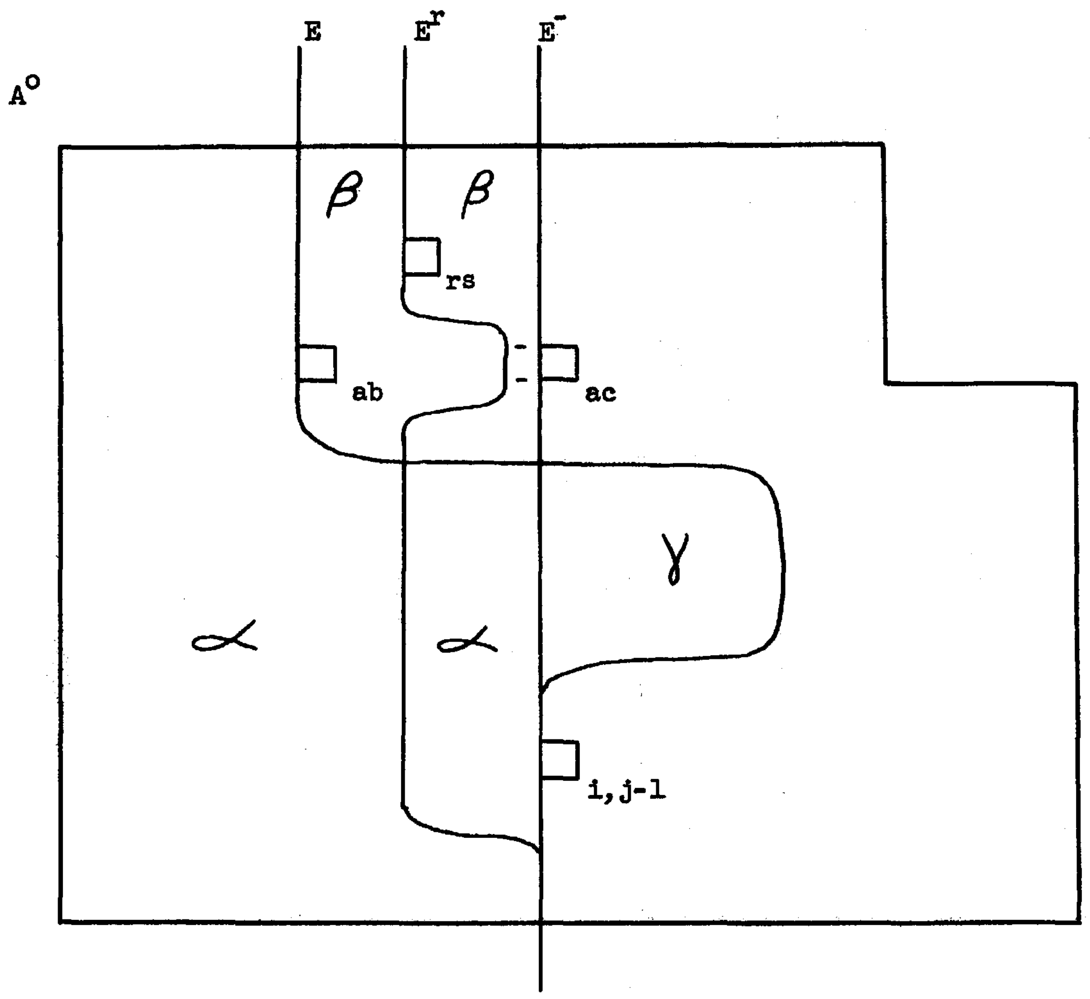


Figure 4.16. Principal boundaries used in the proof of the third stage of the second case of the inductive step.

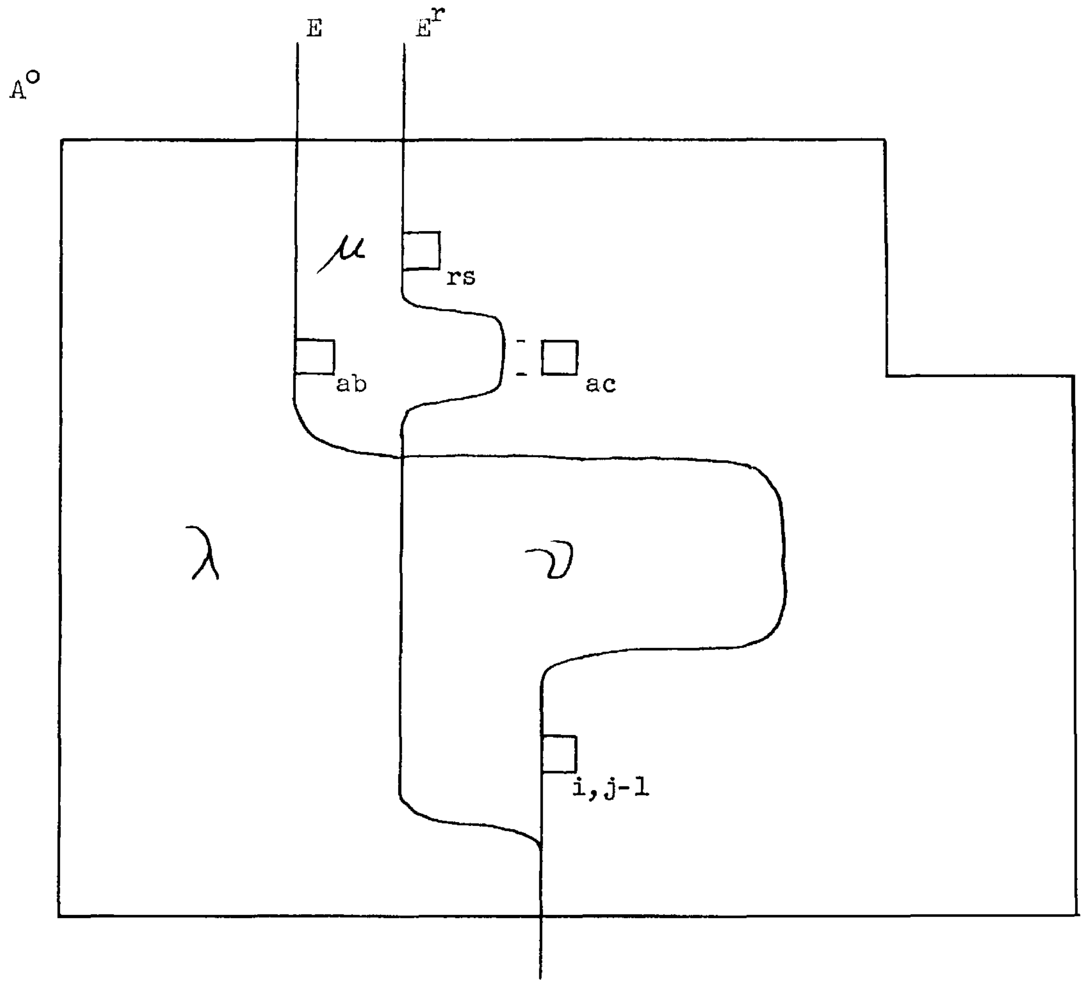


Figure 4.17. The regions λ, μ, and ν.

Let A^{o}_{pq} be an arbitrary a-requiring element such that A_{pq} is one of the first a-requiring elements in ν to be executed in R. Let R^{p} be the prefix of R immediately preceding the execution of A_{pq}. Let A^{p} be the augmented array of R^{p}, and let E^{p} be the edge vector of A^{p}. Figure 4.18 shows the E^{p} boundary drawn in A^{o}. Since R^{p} is a prefix of R, the E^{p} boundary lies everywhere on or to the left of the E boundary.
Figure 4.19 shows the relationship among the runs R^{r}, R^{-}, and R^{+}, and the relationship among the runs R^{p}, R, and R'.

As shown in Figure 4.18, let ρ be the set of positions to the left of both the E^{p} and E^{r} boundaries. Let σ be the set of positions to the right of the E^{p} boundary and to the left of the E^{r} boundary. Let τ be the set of positions to the left of the E^{p} boundary and to the right of the E^{r} boundary.

Since A_{pq} is a-requiring, then at the conclusion of R^{p}, cell p has read capability for cell a. Therefore

$$K(p, a, \rho \cup \tau) > 0$$

Since A_{pq} is one of the first a-requiring elements in ν to be executed in R, and since any a-requiring element of A lying in τ is executed in R before A_{pq} is executed, therefore there are no a-requiring elements in τ. Hence

$$N(p, a, \rho) = N(p, a, \rho \cup \tau)$$

and

$$N(p, a, \rho \cup \sigma) \geqslant N(p, a, \rho \cup \tau)$$

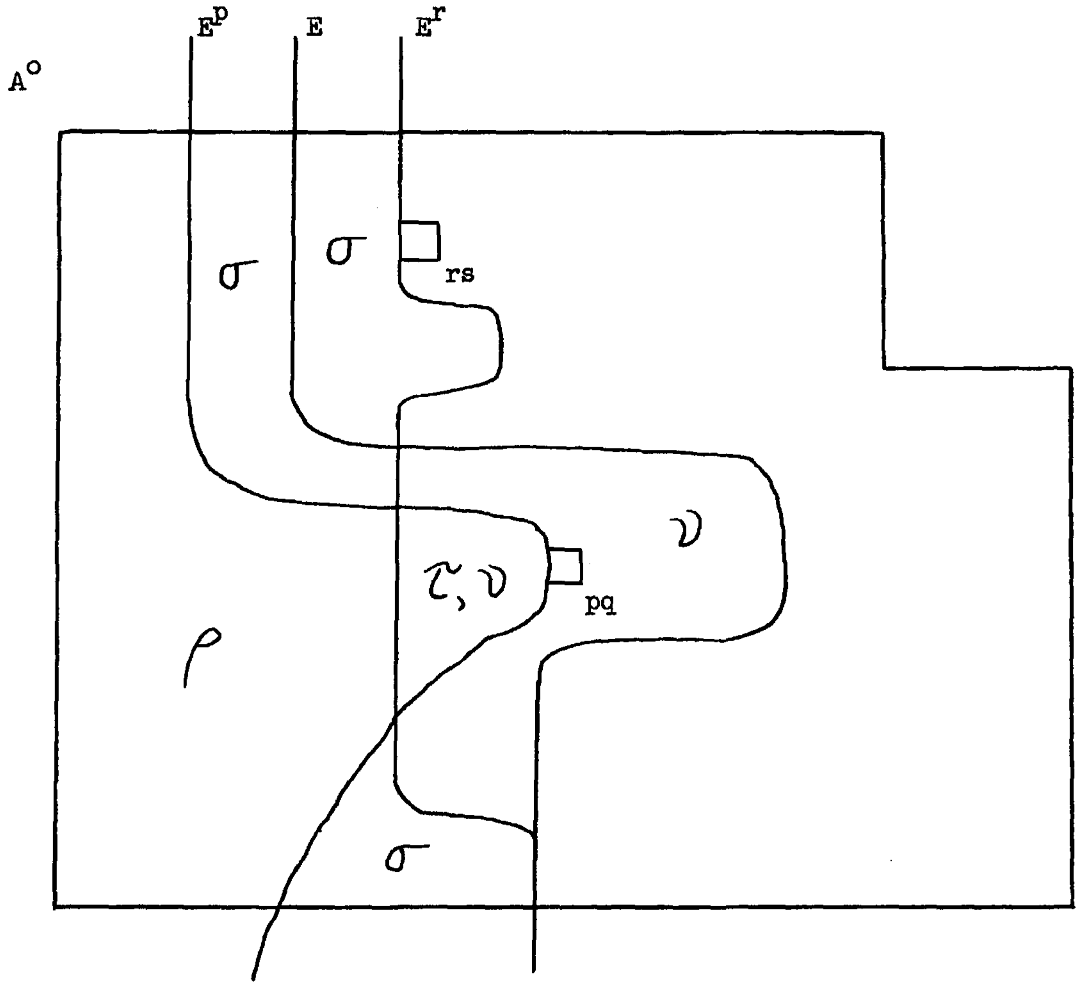
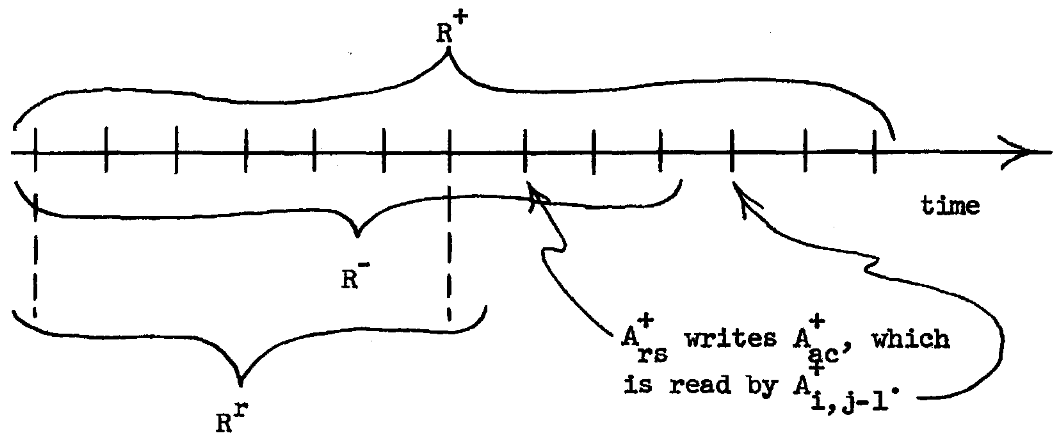


Figure 4.18. Boundaries used in the demonstration that there are no a-requiring elements in ν.


R^+
time
R^-
R^r
A^+_{rs} writes A^+_{ac}, which is read by $A^+_{i,j-1}$.

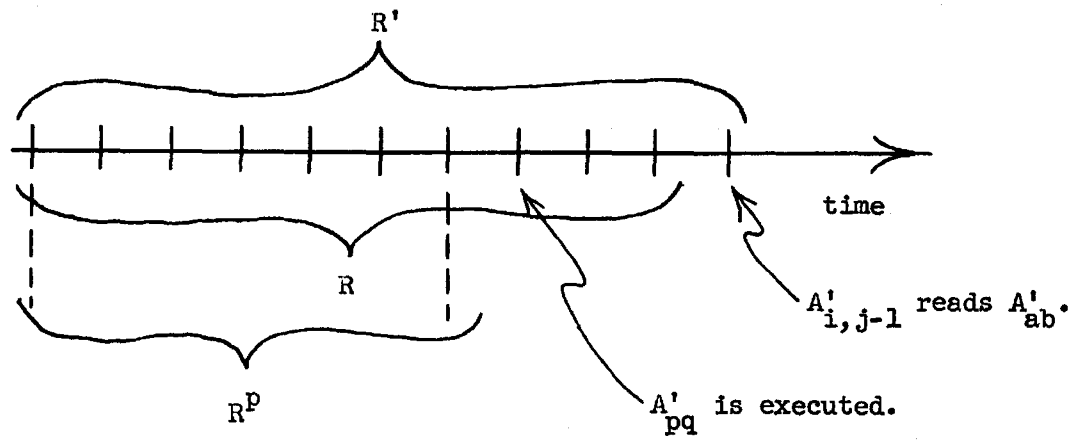


Figure 4.19. Six runs. Hack marks denote computation state transitions, assumed here to be instantaneous.

Since in row p of A^o the E^p boundary lies on or to the right of the E^r boundary, then

$$D(p, a, \rho \cup \sigma) \leq D(p, a, \rho \cup \tau)$$

Thus

$$K(p, a, \rho \cup \sigma) > 0$$

and so at the conclusion of R^r, cell p has read capability for cell a.

Since R^r is the prefix of R^- that immediately precedes the writing of A^-_{ac} by A^-_{rs}, then at the conclusion of R^r, cell r has sole read capability for cell a. Therefore p = r. Then A^o_{rq} belongs to ν, and A^o_{rs} belongs to β. But, as may be seen in Figures 4.16 and 4.17, it is impossible for an element in ν and an element in β to belong to the same row. This contradiction shows that there are no a-requiring elements in ν.

In the first task, which is to eliminate possibility (1) of Figure 4.14, the following two results have been established: A^o_{rs} lies in β, and there are no a-requiring elements in ν. With the aid of Figures 4.16 and 4.17, a contradiction will now be obtained that eliminates possibility (1).

At the conclusion of R, cell i has read capability for cell a. Thus

$$K(i, a, \lambda \cup \nu) > 0$$

Since there are no a-requiring elements in ν, then

$$N(i, a, \lambda) = N(i, a, \lambda \cup \nu)$$

and

$$N(i, a, \lambda \cup \mu) \geq N(i, a, \lambda \cup \nu)$$

Since in run R^-, A^-_{rs} must write A^-_{ac} before $A^-_{i,j-1}$ reads A^-_{ac}, therefore, as shown in Figure 4.16, in row i of A^o the E^r boundary lies on or to the left of the E^- boundary. In row i of A^o, the E^- boundary coincides with the E boundary. Therefore in row i the E^r boundary lies on or to the left of the E boundary, and so

$$D(i, a, \lambda \cup \mu) \leq D(i, a, \lambda \cup \nu)$$

and

$$K(i, a, \lambda \cup \mu) > 0$$

Thus at the conclusion of R^r, cell i has read capability for cell a.

At the conclusion of R^r, cell r has sole read capability for cell a. Therefore, $r = i$, and so in row r the E boundary coincides with the E^- boundary. Therefore no element in row r can lie in β. But A^o_{rs} lies in β. This contradiction eliminates possibility (1) of Figure 4.14.

<u>Task</u> <u>Two</u>. The next task is to disprove possibility (2) of Figure 4.14. As mentioned before, an argument eliminating possibility (2) can be constructed from the argument given above to eliminate possibility (1). The new argument is constructed by interchanging b with c, by interchanging R, A, and E with R^-, A^-, and E^-, respectively, and by interchanging R', A', and E' with R^+, A^+, and E^+, respectively.

<u>Task</u> <u>Three</u>. The third and final task is to show that $A'_{ij} = A^+_{ij}$. It has been established that only possibility (3) of Figure 4.14 prevails, namely, that A^o_{ab} and A^o_{ac} are one and the same element. It is known that the transaction, θ, associated with $A'_{i,j-1}$ is a get of a, and that the same transaction, θ, is associated with $A^+_{i,j-1}$. Thus we have that when $A'_{i,j-1}$ writes A'_{ij}, $A'_{i,j-1}$ reads A'_{ab}, and we have that

when $A^{+}_{i,j-1}$ writes A^{+}_{ij}, $A^{+}_{i,j-1}$ reads A^{+}_{ab}. Clearly $\#A^{+}_{ab}$. Since A'_{ab} must be written before it is read, then $\#A_{ab}$. Since $A \cong A^{+}$, then $A_{ab} = A^{+}_{ab}$. Since $A \subseteq A'$, then $A_{ab} = A'_{ab}$. Thus $A'_{ab} = A^{+}_{ab}$.

Let the replacement function of θ be $f(\cdot)$. We have

$$A'_{ij} = \left\langle f({}_1A'_{ab}),\ i,\ j-1 \right\rangle$$
$$A^{+}_{ij} = \left\langle f({}_1A^{+}_{ab}),\ i,\ j-1 \right\rangle$$

Therefore $A'_{ij} = A^{+}_{ij}$. This completes the third stage of the proof of the second case of the inductive step.

For the second case in the proof of (4.8), it has been shown that $A'_{ij} = A^{+}_{ij}$ for any of the five types of transactions that might be associated with A'_{xy}. The inductive step is thus complete, and the functionality theorem has been proved.

Q. E. D.

Chapter V

The Output Assuredness of an MCM

Introduction

This Chapter presents a proof of the fact that noncompletion lurking bug effects do not occur in an MCM, or in other words, that every MCM is <u>output-assured</u>. An MCM is output-assured if and only if for each initial computation state the output produced by a computation begun from the initial computation state is never arbitrarily cut short, provided the user does not prematurely abort the computation.

In pursuit of the proof of output assuredness, a set-theoretic entity called a <u>trace</u> is introduced as a formal description of an MCM's potential or future behavior subsequent to a start-up from an initial computation state. Then the notion of a <u>limit vector</u> is introduced to describe certain properties of a trace. Using traces and limit vectors, a theorem, called the <u>assuredness theorem</u>, is stated and proved; the truth of the assuredness theorem implies that every MCM is output-assured.

Specialization to a Single Arbitrary MCM

As in Chapter IV, the remarks to be made in this Chapter concern one specific, but arbitrary, MCM. This MCM, which will be designated by the letter M, is described by four quantities: (1) a set N of cell names, (2) a set Q of output cell names that is a subset of N, (3) a transaction table J(i) for each i belonging to N, and (4) a set I of initial computation states. It is assumed that this machine M is well-defined, in the sense described near the end of Chapter II. Since M is arbitrary, the remarks to be made about M, including the assuredness theorem, are true for any well-defined MCM.

Types of Computations

A computation is an instance of the behavior of the machine M. A computation having a finite number of state transitions is either terminating or aborted, depending on whether or not there are no enabled clerk cells at the conclusion of the computation.

There is no such thing as a computation with an infinite number of state transitions; either a computation terminates, or else there comes a time when the user of the machine M, having seen enough, creates an aborted computation by stopping M's activity.

An aborted computation should not be thought of as a "black sheep" indicating the failure of M to perform as hoped for. In some applications of systems for multiprocessing, it is specifically desired that a system's activity not terminate, i.e., that the system not reach a state in which no further actions can take place.

The set-theoretic entity called a run has been formulated as a formal description of a computation; recall that the length of a run is the number of elements in the run's transition sequence. Then the above remarks indicate that every run of the machine M is a finite run, i.e., a run of finite length, and that every run of M is either terminating or aborted.

The Traces of an Initial Computation State

Suppose a user starts up the machine M from some initial computation state S, and let r(S) be the set of possible runs that can proceed from S. While the user does not, in general, know which run in r(S) he will get for his trouble, he does have some choice in the matter -- he may choose one from among a set of progressively longer runs presented sequentially to him by M. Specifically, each transition from one computation state to the next marks the presentation of a new run to the user. If such a transition yields a computation state in which at least one cell is an enabled clerk cell, then the user has the option either of choosing the run obtained by stopping M at the present time, or of waiting until after the next transition in hopes that the run he would obtain by stopping M at that time would be more to his liking.

It is known that each run presented to a user is a possible run, and that the strategy of M's scheduler is reasonable; except for these facts, we have no knowledge at all of the manner in which M selects the runs it presents to a user. It is possible, therefore, to explain the presentation of any sequence of runs by saying that when M is started up from an initial computation state, M selects a set of reasonably scheduled possible runs and thereafter presents these runs to the user sequentially until the user chooses one by stopping M's operation. This model of M's activity will be used in subsequent developments; no generality is lost in doing so, because this model can be used to explain any activity of M that we are allowing ourselves to observe.

When a user starts up M from some initial computation state S, then the set of runs that M immediately selects for presentation to the user is called a <u>trace</u> of S. In general, there are many traces associated with S. On each occasion when M is started up from the computation state S, M arbitrarily selects one of the traces of S to be the set of runs that M will present sequentially to the user during the ensuing computation.

Every trace contains a run of length zero. If a trace contains a run of length m, then it contains exactly one run of length m, and contains all of the prefixes of that run. Every member of a trace is a finite run.

When M is started up from an initial computation state, M's operation might terminate, i.e., M might reach a computation state in which there are no enabled clerk cells. This situation prevails if and only if at the time of M's start-up, M selects a trace containing a terminating run. This trace is necessarily finite, i.e. necessarily contains a finite number of (finite) runs, because a terminating run cannot be a prefix of some longer run.

On the other hand, when M is started up from an initial computation state, M's operation might never terminate, i.e., M might never reach a computation state in which there are no enabled clerk cells. This situation prevails if and only if at the time of M's start-up M selects a trace that does not contain a terminating run. Since the strategy of M's scheduler is reasonable, then this trace cannot be finite, and must therefore contain an infinite number of (finite) aborted runs. Thus the situations which one might be tempted to formalize using runs of infinite length are here being formalized using infinite traces, each containing only finite runs.

The Limit Vector of a Trace

The limit vector L of a trace T is similar in structure to a boundary vector. That is, to each cell of M there corresponds an element of L, and vice-versa. If j is the maximum number of writes occurring into cell i in the runs belonging to T, then

$$L_i = j$$

If there is no upper bound on the number of writes occurring into cell i in the runs belonging to T, then

$$L_i = \phi$$

If M has chosen to behave according to a trace T whose limit vector is L, then the fact that $L_j = i$ says that belonging to T there is a run in which j writes occur into cell i. Since this run is of finite length, then it must be presented to the user in finite time. Therefore, if the user waits long enough, j writes will occur into cell i. No matter how long the user waits, however, no more than j writes will occur into cell i. On the other hand, if $L_i = \phi$, then the user can observe as many writes into cell i as he cares to wait for.

Statement of the Assuredness Theorem

Let the function $t(\cdot)$ be defined so that t(S) is the set of traces according to which M might choose to behave when M is started up from the initial computation state S. Also, let the function $v(\cdot)$ be defined so that v(T) is the limit vector of the trace T. The assuredness theorem is stated as follows: for each initial computation state S, all traces belonging to t(S) have equal limit vectors. In symbols, the assuredness theorem is

$$(S)(T)(U)(T \in t(S) \wedge U \in t(S) \rightarrow v(T) = v(U)) \qquad (5.1)$$

What is the relationship between the assuredness theorem and the notion of output assuredness? Output assuredness holds just when for each output cell x, one or the other of the following two statements is true: (1) the maximum number of words that can be written into cell x is a function only of the initial computation state, or (2) the fact that the number of such words has no upper bound is a function only of the initial computation state. The assuredness theorem says that one of these two statements is true, not just for each output cell x, but for each cell x. That is, the assuredness theorem asserts <u>complete</u> assuredness, not just output assuredness; clearly the assuredness theorem implies output assuredness. It may be noted that complete assuredness is a worthwhile design goal in its own right, because complete assuredness materially facilitates debugging.

The remainder of this Chapter is devoted to a proof of the assuredness theorem, (5.1).

<u>The Formulation</u>

The assuredness theorem will be proved by deriving a contradiction from the assumption that for the initial computation state S, there exists in t(S) two traces, T and U having limit vectors L^t and L^u respectively, such that for some cell i,

$$L_i^t \neq L_i^u \qquad (5.2)$$

It will be further assumed that[*]

$$L_i^u \neq \emptyset \wedge (L_i^t = \emptyset \vee L_i^t > L_i^u) \qquad (5.3)$$

Notice that for the only other possibility implied by (5.2), a contradiction may be derived by interchanging the roles of T and U in the following argument.

The argument leading to a contradiction consists of two parts. First, two particular runs, R^a belonging to T and R^b belonging to U, will be constructed. Second, a contradiction concerning these two runs will be derived.

The Construction

The concepts of augmented array and edge vector, introduced in Chapter IV, will be used here. The following notation will prove convenient: if α is a lower case Roman letter, then the augmented array of the run R^α is A^α, and the edge vector of A^α is E^α.

Let σ be the set of the names of the cells that are written into at most a finite number of times in the runs of U. That is,[**]

$$\sigma = \left\{ n \in N : L_n^u \neq \emptyset \right\}$$

where N is the machine M's set of cell names. By assumption (5.3), $i \in \sigma$.

[*]The proposition A $\vee$ B is true if and only if either A or B or both are true.

[**]The set $\{a \in A : B\}$, where B is usually a function of a, is the set of just those elements of A for which B is true.

Let the predicate $\pi(\cdot)$ be defined so that $\pi(R^e)$ is true if and only if R^e belongs to T and R^e exceeds one of the finite limits of L^u. Specifically, $\pi(R^e)$ is true if and only if*

$$(R^e \in T) \wedge (\exists c)(c \in \sigma \wedge E_c^e > L_c^u) \qquad (5.4)$$

Assumption (5.3) implies that there is at least one run for which $\pi(\cdot)$ is true.

For a run R^e of length 0, $E_c^e = 0$ for each cell c. Since for each cell c, $L_c^u \geqslant 0$, therefore $\pi(\cdot)$ is false for the run belonging to T and having length 0. As was mentioned above, assumption (5.3) implies that there belongs to T at least one run for which $\pi(\cdot)$ is true. Therefore there belongs to T a run R^a that is the longest run for which $\pi(\cdot)$ is false. That is, R^a is constructed to be the longest run that belongs to T and that does not exceed any of the finite limits of L^u.

In proving the functionality theorem in Chapter IV, the proposition (4.3) was proved, which says that any two possible runs which have the same initial computation state have similar augmented arrays. Since all runs belonging to either T or U are possible, and since all these runs have the same initial computation state S, therefore any two of these runs have similar augmented arrays. Thus it is meaningful to let A^o be the union of the augmented arrays of all the runs belonging to T and U.

*The proposition $(\exists x)A$, where A is usually a function of x, is true if and only if there exists at least one x such that A is true.

Figure 5.1 shows the L^t, L^u, and E^a boundaries drawn in A^o. Notice that although E^a is the edge vector of the run R^a, the vectors L^t and L^u are not necessarily the edge vectors of any runs.

Having constructed the run R^a, the next task is to construct a run R^b that belongs to U and for which $A^a \subseteq A^b$. Since $A^a \cong A^b$ for any R^b belonging to U, then R^b need only be constructed so that for each cell n, $E_n^a \leq E_n^b$. Figure 5.2 shows the position of the E^b boundary for the desired R^b. It will now be shown that the execution of the algorithm depicted in flow-chart form in Figure 5.3 always yields a suitable R^b after a finite number of steps.

First it will be shown, by assuming the contrary and deriving a contradiction, that during the execution of the algorithm of Figure 5.3, the "error" exit is never followed. At an instant when the "error" exit is followed, U is finite, R^b is the longest run in U, and $E_n^a > E_n^b$ for some cell n. Since every run in U is a prefix of R^b, then $E_n^b = L_n^u$, and since U is finite, then $n \in \sigma$. Since by construction of R^a, $\pi(R^a)$ is false, then the negation of (5.4) for $R^e = R^a$ is true. In other words, it is true that

$$(R^a \notin T) \vee (c)(c \in \sigma \rightarrow E_c^a \leq L_c^u) \qquad (5.5)$$

Recalling that by construction $R^a \in T$, and putting c = n, where n is that cell mentioned above for which $E_n^a > E_n^b$,

$$n \in \sigma \rightarrow E_n^a \leq L_n^u$$

From above, $n \in \sigma$, and so $E_n^a \leq L_n^u$. But from above also, $E_n^b = L_n^u$ and $E_n^a > E_n^b$, and so $E_n^a > L_n^u$. This contradiction shows that during the execution of the algorithm of Figure 5.3 the "error" exit is never followed.

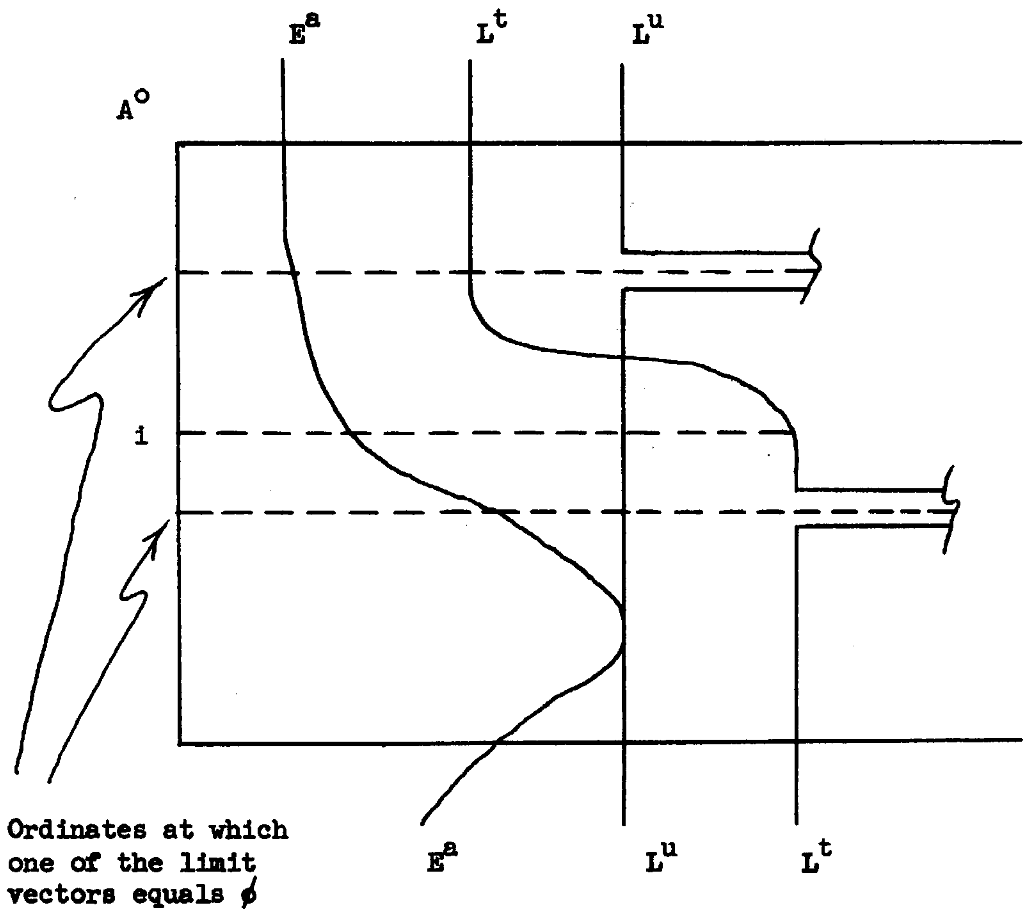


Figure 5.1. The boundaries E^a, L^t, and L^u.

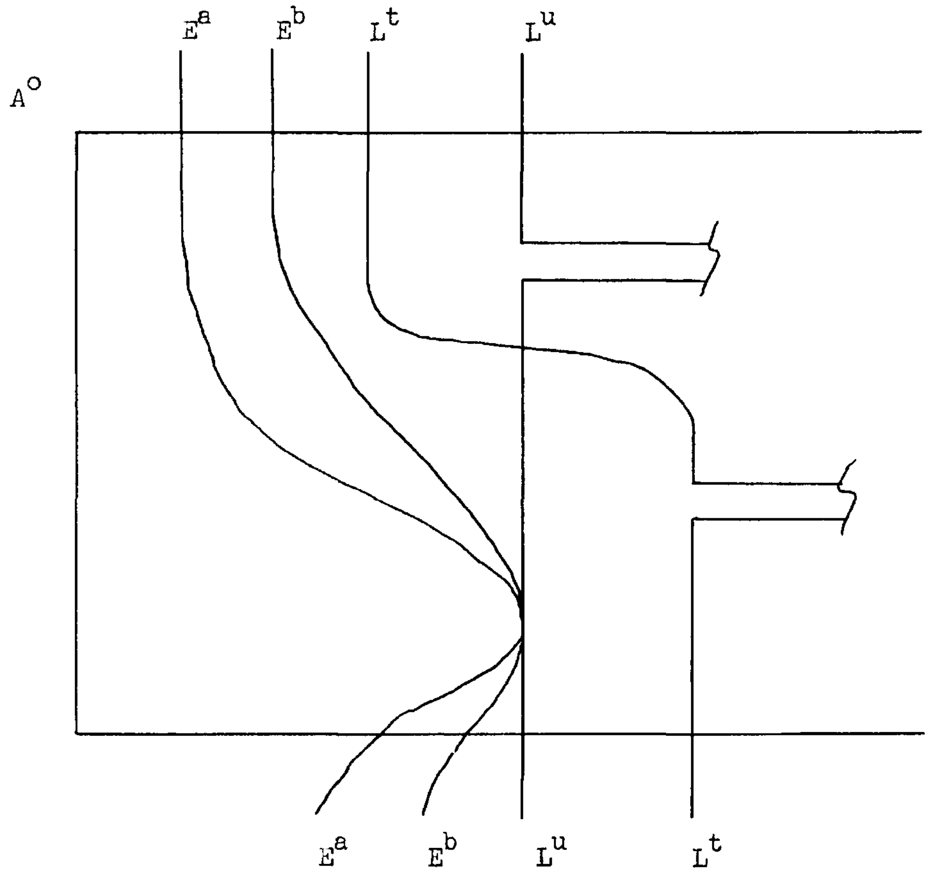


Figure 5.2. Position of the desired E^b boundary.

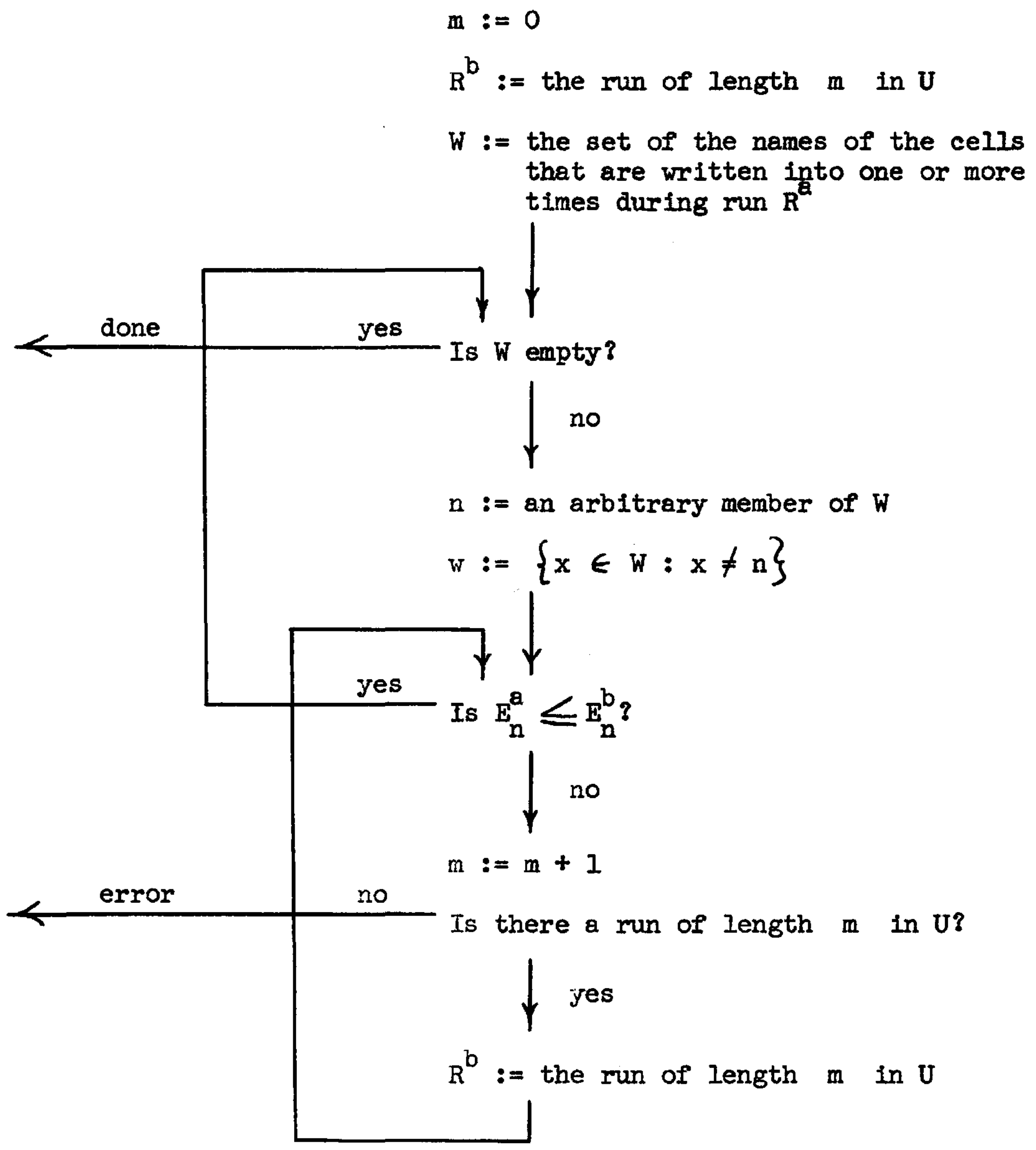

m := 0
R^b := the run of length m in U
W := the set of the names of the cells that are written into one or more times during run R^a
done
yes
Is W empty?
no
n := an arbitrary member of W
w := $\{x \in W : x \neq n\}$
yes
Is $E_n^a \leqslant E_n^b$?
no
m := m + 1
error
no
Is there a run of length m in U?
yes
R^b := the run of length m in U

Figure 5.3. Flow-chart of an algorithm for finding R^b.

Whenever execution of the test "Is $E_n^a \leqslant E_n^b$?" gives a "no" answer, then a "yes" answer to this test is obtained after a finite number of iterations around the lower loop. In order to show that this statement is true, it must first be established that whenever $E_n^a > E_n^b$ for some n, then there exists a run R^f that belongs to U and for which $E_n^a \leqslant E_n^f$. If $L_n^u = \phi$, then R^f clearly exists, because then there is no upper bound on the number of writes into cell n in the runs of U. If $L_n^u \neq \phi$, then by definition of σ, $n \in \sigma$. Using $R^a \in T$ in (5.5), and as before putting $c = n$ and using $n \in \sigma$, then $E_n^a \leq L_n^u$. Then R^f exists for the case in which $L_n^u \neq \phi$, because there exists some run R^f that belongs to U and for which $E_n^f = L_n^u$.

Since the lower loop is entered when $E_n^a > E_n^b$ for some n, and since there exists a run R^f that belongs to U and for which $E_n^a \leqslant E_n^f$, therefore R^b is a prefix of this R^f and the length of R^b upon entry into the lower loop is less than the length of R^f. Since iterations around the lower loop increase the length of R^b, then the "yes" exit is sure to be taken from the lower loop when $R^b = R^f$, or sooner. Since R^f is of finite length, then the "yes" exit is sure to be taken from the lower loop after a finite number of iterations.

The "done" exit is taken from the upper loop after a finite number of iterations, because the lower loop always takes a finite number of iterations, and because the finite length of R^a assures that W is initially finite. Finally, upon exit at "done", R^b is a run with the desired properties, because for each n such that $E_n^a > 0$, the execution of the algorithm has uncovered a run $R^{(n)}$, having edge vector $E^{(n)}$, whose length is less than or equal to that of R^b and for which $E_n^a \leqslant E_n^{(n)}$. Since for each such n, $R^{(n)}$ is a prefix of R^b, then for each such n, $E_n^{(n)} \leqslant E_n^b$, and so for each such n, $E_n^a \leqslant E_n^b$. On the other hand, if n is such that $E_n^a = 0$, then clearly $E_n^a \leqslant E_n^b$. Thus an R^b has been constructed that belongs to U and for which $A^a \subseteq A^b$.

<u>The Contradiction</u>

The run R^a is the longest run in T that does not exceed any of the finite limits set by L^u. Let R^c be the run in T of length one greater than the length of R^a. Then there exists some cell k into which exactly m writes occur in R^c, where $m = L_k^u + 1$. Thus $\#A_{km}^c$, i.e., A_{km}^c is defined, but $\neg\#A_{km}^a$. Notice that cell k is not necessarily the same as the cell i mentioned in the assumption to be contradicted, (5.2). Cell k is one of the first "trespassers" of L^u, whereas cell i is an arbitrary "trespasser" of L^u.

Let $\langle x, y\rangle$ be defined so that A_{xy}^c writes A_{km}^c. As shown in Figure 5.4, A_{xy}^o lies just to the right of the E^a boundary, and so $y = E_x^a$.

By construction, it is known that for the run R^c belonging to T, A_{xy}^c is executed. It will now be shown for each run R^d belonging to U, $\neg\#A_{x,y+1}^d$, or in other words that $L_x^u \leqslant y$. This fact will be shown by assuming the contrary and deriving a contradiction. Let R^d be a run belonging to U such that $\#A_{x,y+1}^d$. Since A_{xy}^c is executed, then by $A^c \cong A^d$ we have that A_{xy}^d is executed. Since A_{xy}^c is k-writing, then by $A^c \cong A^d$ we have that A_{xy}^d writes A_{kv}^d for some $v \leqslant L_k^u$. Since by construction $m = L_k^u + 1$, then $v < m$. Since $\#A_{km}^c$, then $\#A_{kv}^c$, and by

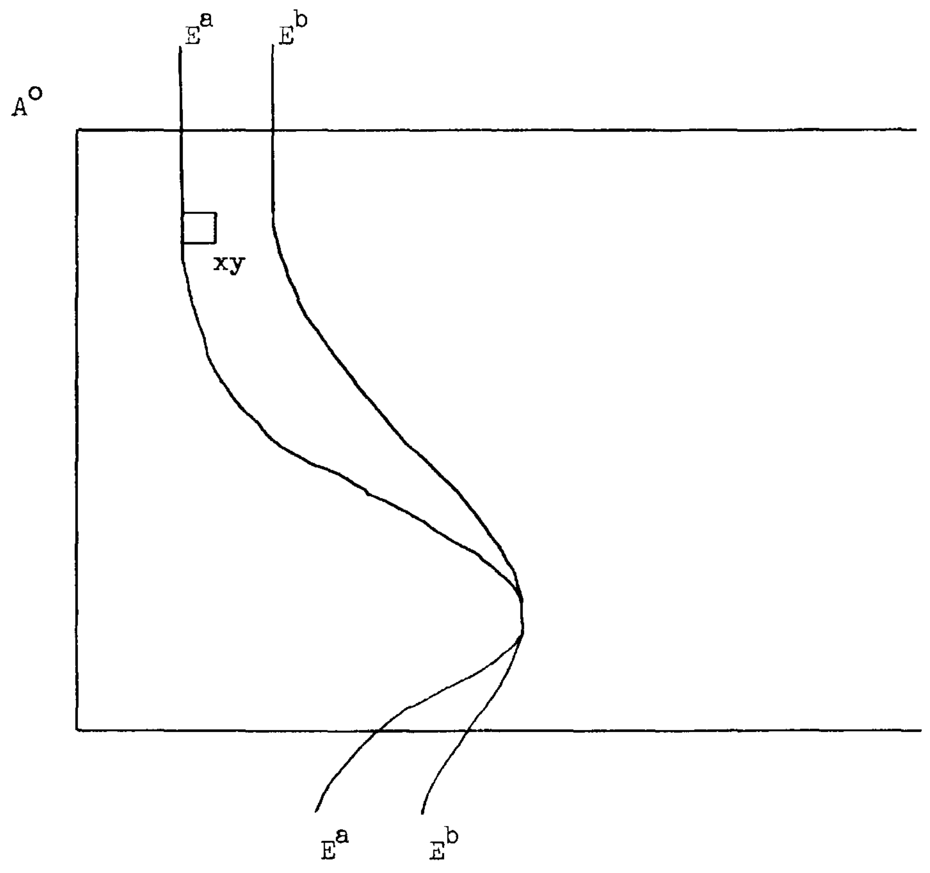


Figure 5.4. The element A^o_{xy}.

$A^c \cong A^d$ we have that A^c_{kv} is written by A^c_{xy}. But A^c_{xy} writes A^c_{km}, and $v < m$. This contradiction shows that $L^u_x \leq y$.

It will now be shown that $L^u_x = y$. Since $y = E^a_x$, and, by construction of R^b, $E^a_x \leq E^b_x$, therefore $y \leq E^b_x$. But since $R^b \in U$, then $E^b_x \leq L^u_x$, and so $y \leq L^u_x$. From the preceding paragraph, $L^u_x \leq y$, and so $L^u_x = y$.

The situation concerning the runs R^a and R^b is shown in Figure 5.5, where for clarity the E^a and E^b boundaries are shaped differently than in preceding Figures. In accordance with the construction of A^c_{xy} and A^c_{km}, Figure 5.5 shows A^o_{xy} lying just to the right of the E^a boundary, and A^o_{km} lying to the right of, and exactly one position away from the E^a boundary. In accordance with the construction of R^b, Figure 5.5 shows the E^b boundary lying everywhere on or to the right of the E^a boundary. Since $L^u_x = y = E^a_x$, therefore in row x of A^o the E^b boundary coincides with the E^a boundary. Thus $E^a_x = E^b_x = y = L^u_x$.

Figure 5.5 shows that in row k of A^o the E^b boundary also coincides with the E^a boundary. This fact is true because $m = L^u_k + 1$. That is, $L^u_k = m-1$, and so $E^b_k \leq m-1$. Since $E^a_k = m-1$, then $E^b_k \leq E^a_k$. Since the E^b boundary lies everywhere on or to the right of the E^a boundary, therefore $E^b_k \geq E^a_k$. Thus $E^a_k = E^b_k = m-1 = L^u_k$.

It will now be shown that cell x is an enabled clerk cell at the conclusion of R^b. As shown in Figure 5.5, let α be the set of positions in A^o to the left of both the E^a and E^b boundaries. Let β be the set of positions in A^o to the right of the E^a boundary and to the left of the E^b boundary.

Let θ be the transaction associated with A^b_{xy}. By $A^a \cong A^b$, θ is also the transaction associated with A^a_{xy}. Since x might equal k,

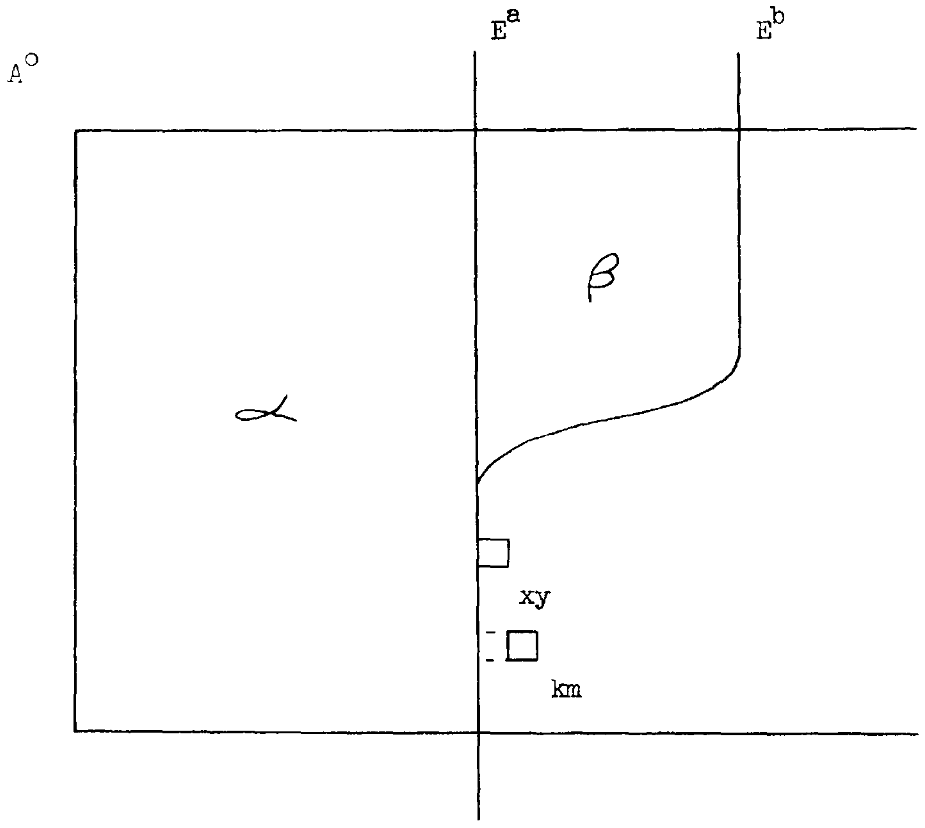


Figure 5.5. The E^a and E^b boundaries redrawn.

then θ might be any of the five transaction types. At the conclusion of R^a, cell x is an enabled clerk cell that is enabled to perform θ. In order to show that cell x is also an enabled clerk cell at the conclusion of R^b, it must be shown that at the conclusion of R^b cell x is enabled to perform θ.

To show that at the conclusion of R^b, cell x is enabled to perform θ, two points will be demonstrated: (1) if cell x has read capability for an arbitrary cell r at the conclusion of R^a, then cell x has read capability for cell r at the conclusion of R^b, and (2) if cell x has write capability for any arbitrary cell w at the conclusion of R^a, then cell x has write capability for cell w at the conclusion of R^b.

Digression. Just as in Chapter IV, the subscripts on K, N, and D will be omitted here, for the same reasons that justified their omission in Chapter IV. Specifically, the subscript of N and D may be taken here to be A^o, and the subscript of K does not matter here since all runs being discussed have the same initial count matrix.

Resumption. The truth of point (1) mentioned above, concerning read capability for a cell r, is easily demonstrated. Since cell x has read capability for cell r at the conclusion of R^a, then

$$K(x, r, \alpha) > 0$$

Clearly

$$N(x, r, \alpha \cup \beta) \geq N(x, r, \alpha)$$

Since the E^a and E^b boundaries coincide in row x of A^o, then

$$D(x, r, \alpha \cup \beta) = D(x, r, \alpha)$$

Therefore

$$K(x, r, \alpha \cup \beta) > 0$$

and so cell x has read capability for cell r at the conclusion of R^b.

The truth of point (2) mentioned above, concerning write capability for a cell w, is demonstrated by an argument similar to that used several times in Chapter IV. First it will be shown that there are no w-requiring elements in β, by assuming the contrary and deriving a contradiction. Let A^o_{pq} be an arbitrary element of A^o such that A^b_{pq} is one of the first w-requiring elements in β to be executed in R^b. Let R^p be the prefix run of R^b that just precedes the execution of A^b_{pq} in R^b. Figure 5.6 shows the E^p boundary drawn in A^o.

Let ρ be the set of positions in A^o to the left of both the E^a and E^p boundaries. Let σ be the set of positions in A^o to the right of the E^a boundary and to the left of the E^p boundary. Let τ be the set of positions in A^o to the left of the E^a boundary and to the right of the E^p boundary.

Since A^b_{pq} is w-requiring, then

$$K(p, w, \rho \cup \sigma) > 0$$

Since A^b_{pq} is one of the first w-requiring elements in β to be executed in R^b, then there are no w-requiring elements in σ. Thus

$$N(p, w, \rho) = N(p, w, \rho \cup \sigma)$$

and so

$$N(p, w, \rho \cup \tau) \geqslant N(p, w, \rho \cup \sigma)$$

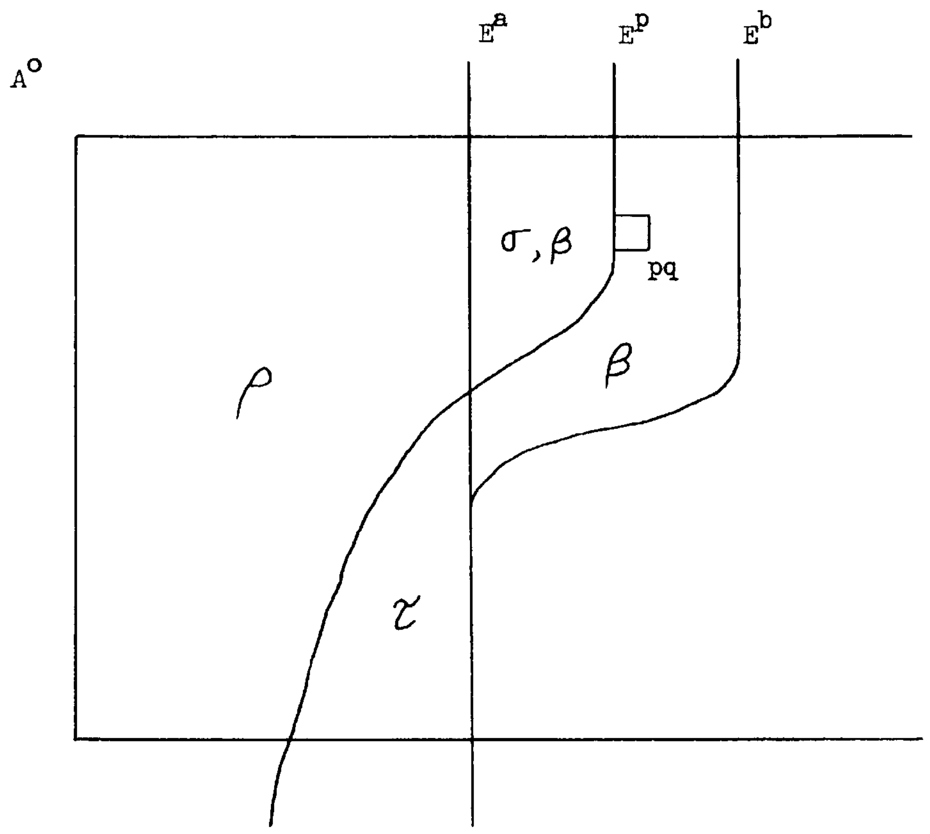


Figure 5.6. The E^p boundary.

Since in row p of A^o the E^a boundary lies on or to the left of the E^p boundary, then

$$D(p, w, \rho \cup \tau) \leqslant D(p, w, \rho \cup \sigma)$$

Therefore

$$K(p, w, \rho \cup \tau) > 0$$

and cell p has read capability for cell w at the conclusion of R^a.

It is known that at the conclusion of R^a, cell x has write capability, i.e., sole read capability, for cell w. Therefore p = x, and so A^b_{xq} lies in β. But in row x of A^o, the E^a boundary coincides with the E^b boundary, and so no element in row x of A^o can lie in β. This contradiction shows that there are no w-requiring elements in β.

Let z be an arbitrary cell name, such that $z \neq x$. With the aid of Figure 5.5, it will now be shown that cell z does not have read capability for cell w at the conclusion of R^b. At the conclusion of run R^a, cell x has write capability for cell w, and so cell z does not have read capability for cell w at the conclusion of R^a. Therefore

$$K(z, w, \alpha) \leqslant 0$$

Since there are no w-requiring elements in β,

$$N(z, w, \alpha \cup \beta) = N(z, w, \alpha)$$

Since in row z of A^o, the E^b boundary lies everywhere on or to the right of the E^a boundary, then

$$D(z, w, \alpha \cup \beta) \geqslant D(z, w, \alpha)$$

Thus

$$K(z, w, \alpha \cup \beta) \leqslant 0$$

and cell z, an arbitrary cell not the same as cell x, does not have read capability for cell w at the conclusion of R^b.

The above result shows that no cell other than perhaps cell x has read capability for cell w at the conclusion of R^b. Thus if cell x has read capability for cell w at the conclusion of R^b, then cell x has write capability for cell w at the conclusion of R^b. It is known that at the conclusion of R^a, cell x has write capability for cell w, and it has already been shown that if cell x had read capability for some cell at the conclusion of R^a, then cell x has read capability for the cell at the conclusion of R^b. Therefore, cell x has read capability for cell w at the conclusion of R^b, and so cell x has write capability for cell w at the conclusion of R^b. The demonstration of point (2) is now complete.

It has been shown that cell x has the same capabilities at the conclusion of R^b as it does at the conclusion of R^a. Since $E_x^a = E_x^b$, and $A^a \cong A^b$, then cell x has the same content at the conclusion of R^a as it does at the conclusion of R^b. Since cell x is an enabled clerk cell at the conclusion of R^a, therefore cell x is an enabled clerk cell at the conclusion of R^b.

At the conclusion of R^b, $E_x^b = y = L_x^u$. Therefore position $\langle x, y+1 \rangle$ is never written in any run belonging to U, and therefore position $\langle x, y \rangle$ is never executed in any run belonging to U. Thus, after the run R^b is presented to the user, the cell x, even though it is an enabled clerk cell, will never receive a go pulse no matter how long the user waits. This result means that the strategy of M's scheduler is not reasonable, and contradicts the fact that the strategy of M's scheduler is reasonable.

This contradiction proves that assumption (5.2) is false. Since the only other possibility that (5.2) implies can be contradicted by interchanging the roles of T and U in the preceding argument, then (5.2) has been contradicted. Thus $L_i^t = L_i^u$ for each cell i, and the assuredness theorem has been proved.

Q. E. D.

Chapter VI

Conclusions and Suggestions for Future Research

Introduction

Previous Chapters have described some results of research on the subject of multiprocessing. The present Chapter discusses the possible effects that this body of results might have in the field of computer science and engineering. In discussing the effects of the present research, the Chapter pursues two related topics: effects the research might have on applications, and avenues along which the research might be continued.

Asynchronous Reproducibility

The Thesis has reported on a concept for designing a computing facility to meet certain performance criteria. An attempt has been made to capture the essence of this design concept by formulating a class of abstract machines called machines for coordinated multiprocessing, or MCM's. The performance criteria that an MCM has been shown to satisfy are the criteria of output functionality and output assuredness.

In showing that an MCM is both output-functional and output-assured, it has actually been shown that an MCM has the properties of complete functionality and complete assuredness. Let us sum up these latter properties using the term <u>asynchronous reproducibility</u>: to say that the behavior of an MCM is asynchronously reproducible is to say that the MCM has both the property of complete functionality and the property of complete assuredness.

The fact that an MCM's behavior is asynchronously reproducible means that if one starts up an MCM on two different occasions from the same arbitrary initial computation state, then for each cell, one will observe the same sequence of words written into the cell during both computations, provided one does not prematurely abort either or both of the computations. The adjective "asynchronous" means "without regard to the relative timing among them". Thus asynchronous reproducibility means that the sequence of words written into each cell is reproducible, but that the relative timing of the production of such sequences for several cells is not necessarily reproducible.

The Conditional Nature of Asynchronous Reproducibility

Asynchronous reproducibility does not mean that it is actually possible to start up an MCM from the same, arbitrary initial computation state on two different occasions. Asynchronous reproducibility means only that if an MCM is started up from the same, arbitrary initial computation state on two different occasions, then the same sequences of words will be written into its cells on both occasions, provided the computations are not prematurely aborted.

Consider a facility in which a clerk is able to obtain from a clock device the date and time of day. It is possible for the behavior of such a facility to be asynchronously reproducible, even though any given instance of its behavior cannot actually be reproduced. The sequence of values read from the clock device constitutes an input stream to the facility. In deciding whether the facility has asynchronously reproducible behavior, we must ask ourselves, "If the clock input stream were reproduced, along with all other initial conditions, would the facility's cell histories be reproduced?" If and only if for all initial conditions, including clock input streams, the answer to this question is "Yes", then the facility has asynchronously reproducible behavior.

Repeatable Input Streams

To prevent the occurrence of lurking bugs of the two types mentioned in Chapter I, it is sufficient to construct a facility with asynchronously reproducible behavior. In practice, however, a user of such a facility will not benefit from the facility's asynchronously reproducible behavior unless he is able to repeat at will the initial conditions of a computation. Thus, when a facility is constructed to have asynchronously reproducible behavior, it is a practical necessity that the facility also be constructed to allow a user to provide, say, a previously prepared tape as a substitute for any input stream not normally under the user's control. Such an input stream might be, for example, the sequence of values read from a clock, or the sequence of responses obtained from requests for access to a "common file" shared by several users.

From the point of view of the designer of a facility, the requirement that the facility have repeatable input streams means the following: a clerk must not be allowed to read any quantity that in the midst of a computation might be affected by external influences and for which the designer is not willing to provide a mechanism allowing a user to repeat at will the quantity's sequence of values. For example, based on this requirement, a designer might choose to not allow a clerk to sense the state of memory allocation, and might choose to not allow a clerk to read a timer register that tells when the clerk's current processing unit will start playing the role of another clerk. For still another example, suppose the designer of a facility does choose to allow a clerk to read the time of day, and therefore also chooses to allow a user to switch the facility between reading the time of day and reading, say, a previously prepared tape. Consider the designer's dilemma now in deciding whether to allow this switching between the clock and the tape to be performed by a clerk as well as by a user. The requirement of repeatable inputs immediately resolves the issue: to allow such switching by a clerk would be self-defeating, because, as a result of a program bug, and without the user's knowledge, a clerk might start reading the time of day instead of the prepared tape.

Clearly the requirement of repeatable inputs, like the requirement of asynchronous reproducibility, is a strong design criterion in the sense that it serves to eliminate from consideration many possible system designs.

State Input Streams

In Chapter III it was mentioned that the state input stream of an input or output device is usually not repeatable at the will of the programmer. To allow a prepared tape to be substituted for a state input stream, however, is likely to lead to difficulties. For example, a clerk may sense from a tape-simulated state input stream that a device is "ready" when in fact the device is not. The device might then be sent an improper command, which could cause the reading of primary input stream values different from the values intended by the user. The problem of designing an input or output device so that its behavior can be repeated in synchronism with a computation is a good subject for future research.

The Hang-Up Phenomenon

An interesting phenomenon that can occur in an MCM is the hang-up. In the computation state S, a clerk cell x is hung up if and only if the following two statements are true: (1) clerk cell x will never become enabled in any computation that proceeds from the computation state S, and (2) the truth of statement (1) can be established by examining only the computation state S, and the types and operand names of the transactions in the MCM's transaction tables. Statement (2) means, in particular, that the existence of a hang-up depends neither on the potential occurrence of an endless loop, nor on any other fact that can be discovered only through what would be in effect a simulation of each computation that can proceed from the computation state S.

An example of a computation in which clerk cell x is hung up is one in which clerk cell x requires read capability for a cell i in order to be enabled, and in which no cell at all has read capability for cell i.

It is also possible for two clerk cells to be mutually hung up. For example, consider a computation state in which a clerk cell x requires read capability for a cell i in order to be enabled, and in which a clerk cell p requires read capability for a cell k in order to be enabled. Then clerk cells x and p are both hung up in any such computation state in which clerk cell x has read capability for cell k, and clerk cell p has read capability for cell i.

Clearly one can have more than two clerk cells mutually hung up on each other, and it is not difficult to construct examples of complex hung-up modes in which an arbitrary number of clerk cells are hung up for a variety of reasons.

The assuredness theorem says that the effects of an MCM hang-up are reproducible in the following sense. If during a computation from an initial computation state S, a user observes that clerk cell x is hung up after exactly y writes have occurred into clerk cell x, then during any subsequent computation from S, the user will observe exactly y writes into cell x and no more, provided he waits long enough.

It is possible that not only the effect but also the mode of an MCM hang-up is reproducible; this possibility is a worthwhile subject for future research.

It appears that a hang-up results from an inconsistency in the specification of the computational activity, as for example if one tried to program the simultaneous execution of

$$a := g(b)$$

by one clerk, and

$$b := f(a)$$

by another clerk. A contemporary facility might respond to this ambiguity by making an arbitrary choice as to which statement to execute first. A facility that behaves like an MCM, however, would always respond to this ambiguity by hanging up.

The hang-up mode in which no cell at all has read capability for some cell can be avoided by using the concept of ownership discussed in the next Section. The cataloging of other hang-up modes and the finding of methods to avoid these modes are topics for future research.

Ownership

It is possible for the cells of an MCM to perform enough dones of some cell i so that no cell has read capability for cell i. After such a total relinquishment of cell i, no cell can ever again obtain read capability for cell i, because read capability for cell i can only be obtained as a result of the action of a cell that already has read capability for cell i.

The concept of ownership is a method of preventing total relinquishment in a facility that behaves like an MCM. It is hoped that the following explanation of this concept will be of assistance in the search for ways to prevent hang-up modes other than total relinquishment.

When the ownership concept is employed in a facility, then for each part n, such as a segment, clerk, input device, or output device, there is in every computation state either exactly one part, or exactly one unused name, that is the <u>owner</u> of the part n.

Clerk x is the owner of, say, segment n if and only if a special <u>ownership</u> <u>bit</u> at position $\langle x, n \rangle$ of the control matrix is on. The fact that clerk x is the owner of segment n means the following: an enormously large number, say

$$10^{(10^{10})} \tag{7.1}$$

is considered to be added to the integer at position $\langle x, n \rangle$ of the control matrix, and the result is considered to be the number at position $\langle x, n \rangle$ of the control matrix for the purpose of determining capabilities. Thus the owner of a part always has read capability for the part.

Ownership of the segment n may be conveyed from clerk x to clerk e by clerk x's execution of the procedure step

    convey 'n', 'e';

Execution of this procedure step turns on the ownership bit at position $\langle e, n \rangle$ of the control matrix, and turns off the ownership bit at position $\langle x, n \rangle$ of the control matrix.

The use of ownership in a facility that behaves like an MCM does not imply a modification to the MCM design as presented in Chapter II. The combination of ownership bit and integer in a control matrix element is just a special way of encoding an extremely large count. Executions of send's and done's affect only the integer parts of control matrix elements, because in practice it is impossible for enough send's and done's to be performed to affect ownership. The transfer of ownership brought about by an execution of a convey procedure step corresponds to the performance in an MCM of the (7.1) number of send transactions, followed by the performance of the (7.1) number of done transactions.

Questions of Necessity

Questions that should prove very stimulating to future research are questions of the form, "Is property A of the MCM design necessary for satisfying performance criterion B?" For example, the question raised in Chapter IV concerning whether complete functionality is necessary for output functionality is a question of this type. The key to the proper formulation of such questions of necessity is to recognize that their proper formulation requires a framework broader than that of the MCM design itself. That is, in order to state such questions, one must have in mind a class of machines for multiprocessing that includes the class of MCM's. Then with respect to this class of machines, property A is necessary for criterion B if and only if the sub-class of machines satisfying criterion B is included in the sub-class of machines having property A.

Let us consider what a reasonable class, μ, of machines for multiprocessing might be. The class μ ought to include the class of MCM's. Also, μ ought to include a class of machines that are models of contemporary facilities for multiprocessing. One such class of machines is the class of machines that are like MCM's, but that have no count matrices, and that have schedulers which transmit go pulses according to arbitrary schemes.

The class μ consisting of just MCM's and contemporary multiprocessing models is not very interesting. For example, recall from Chapter I that contemporary facilities for multiprocessing are not, in general, output-functional. It was shown in Chapter IV, however, that MCM's are output-functional. Therefore, with respect to the class μ as developed so far, the count matrix and enabling rules found in the MCM design are, in a trivial sense, necessary for output functionality.

To make a non-trivial assertion of the necessity of some MCM feature that is not found in any contemporary facility, one must include in μ more machines than just MCM's and models of contemporary facilities. What might be the structure of these additional machines? The computation state of such a machine might be given by a vector of state variables. Some of these variables would be similar to words in MCM cells, in that their histories would be of interest in the criterion of asynchronous reproducibility. The rest of the variables in the computation state vector would be similar to elements in an MCM's count matrix, in that their histories would not be of interest in the criterion of asynchronous reproducibility. Each active element of such a machine might be memoryless, but might have "tentacles" extending to some subset of the computation state variables. Upon receipt of a go pulse, an active element would read the variables at the ends of its tentacles, and then give new values to some or all of these variables, according to its "wired in" properties.* Go pulses

*This scheme was suggested by R. Gammill.

would be transmitted to active elements by a scheduler on the basis of both current computation state, and other influences.

If the class μ is augmented by including in it, say, the class of "tentacle" machines described above, then a search for the answers to questions of necessity with respect to μ would amount to a study of the properties and performance of various specializations of the "tentacle" structure. A further exploration of such questions of necessity is left for future research.

Toward a Science of Computer Design

As was mentioned previously in the Chapter, the criteria of asynchronous reproducibility and input repeatability place strong requirements on the design of a computing facility; that is, these criteria serve in the design process to eliminate from consideration a great many possible designs.

Computer design today is an art. Today an engineer designing a computer cannot help but be worried that the apparently arbitrary decisions he makes may prove wrong or inconsistent when his computer is used in some application. Designers have too much choice; criteria must be formulated which serve to determine as many aspects of a computer's design as possible. The development of such criteria helps to turn computer design from an art into a science.

A science of computer design must be based on fundamental principles of at least two kinds: (1) performance criteria, and (2) postulates characterizing the technological and economic environment. An example of a postulate might be, "The cost of high-speed, random access memory will never be low enough to eliminate the need for a storage hierarchy." Once a satisfactory set of criteria and postulates has been formulated, then one may deduce, rather than design, the structure of a computer. Of course, several sets of criteria and postulates might exist, each appropriate for a different class of applications.

The present research has contributed toward the reduction of computer design from an art into a science by displaying the performance criterion of asynchronous reproducibility, and by showing that it is possible to satisfy this criterion with a reasonably economical computer design. The present research, however, is merely a beginning; the seeking of criteria and postulates for a science of computer design is a most fruitful subject for future research efforts.

Appendix A

The Non-Redundancy of Bye Transactions

In Chapter II's Section on send, done, and bye transactions, it was mentioned that bye transactions are not redundant in the MCM design. This assertion will now be explained in more specific terms.

Consider the class of events which we shall call <u>events of type A</u>. An event of type A is said to have occurred just when both of the following statements are true: (1) at one instant some cell x has write capability for itself, and (2) at a later instant some other cell has write capability for cell x.

If bye transactions are included in the MCM design, then an MCM can be constructed in which an event of type A can occur. For example, suppose that at some time t_1 every element in column x of an MCM's count matrix is zero, except that the element at position $\langle x, x\rangle$ equals one. During the interval between time t_1 and some later time t_2 let cell x perform a bye to e for some e not equal to x, and let this performance be the only performance of a transaction by any cell between the times t_1 and t_2. At time t_2, every element in column x of the count matrix is zero, except that the element at position $\langle e, x\rangle$ equals one. The circumstances of this example constitute an event of type A.

If bye transactions are excluded from the MCM design, then no MCM can be constructed in which an event of type A can occur. This fact will be proved by assuming it is false and deriving a contradiction. That is, it is now assumed that there exists an MCM which does not perform bye transactions, but in which an event of type A can occur. In order to describe the event of type A which can occur in the assumed MCM, let t_1, t_2, and x be chosen so that at time t_1 the cell x has write capability for itself, and so that at the later time t_2 some other cell has write capability for cell x.

As shown in Figure A.1, cell x performs at least one done of x in the interval between t_1 and t_2. Let α denote the computation state transition during which the _last_ such done is performed. By the enabling rule for dones, cell x has write capability for itself just before the transition α. Therefore, no sends of x are performed during α, and so just after α, no cell, other than perhaps cell x itself, has read capability for cell x.

Since at time t_2, cell x does not have read capability for itself, then by construction of α, we know that just after α cell x does not have read capability for itself. Therefore, just after α no cell at all has read capability for cell x. Thus, after α has occurred no cell can ever again have read capability for cell x, because every transaction that adds to an element in column x of the count matrix requires read capability for cell x in order to be enabled.

The above result contradicts the fact that some cell has write capability for cell x at the time t_2. This contradiction means that if bye transactions are excluded from the MCM design, then no MCM can be constructed in which an event of type A can occur. As was shown previously, the inclusion of bye transactions in the MCM design does allow events of type A to occur, and so bye transactions are not redundant.

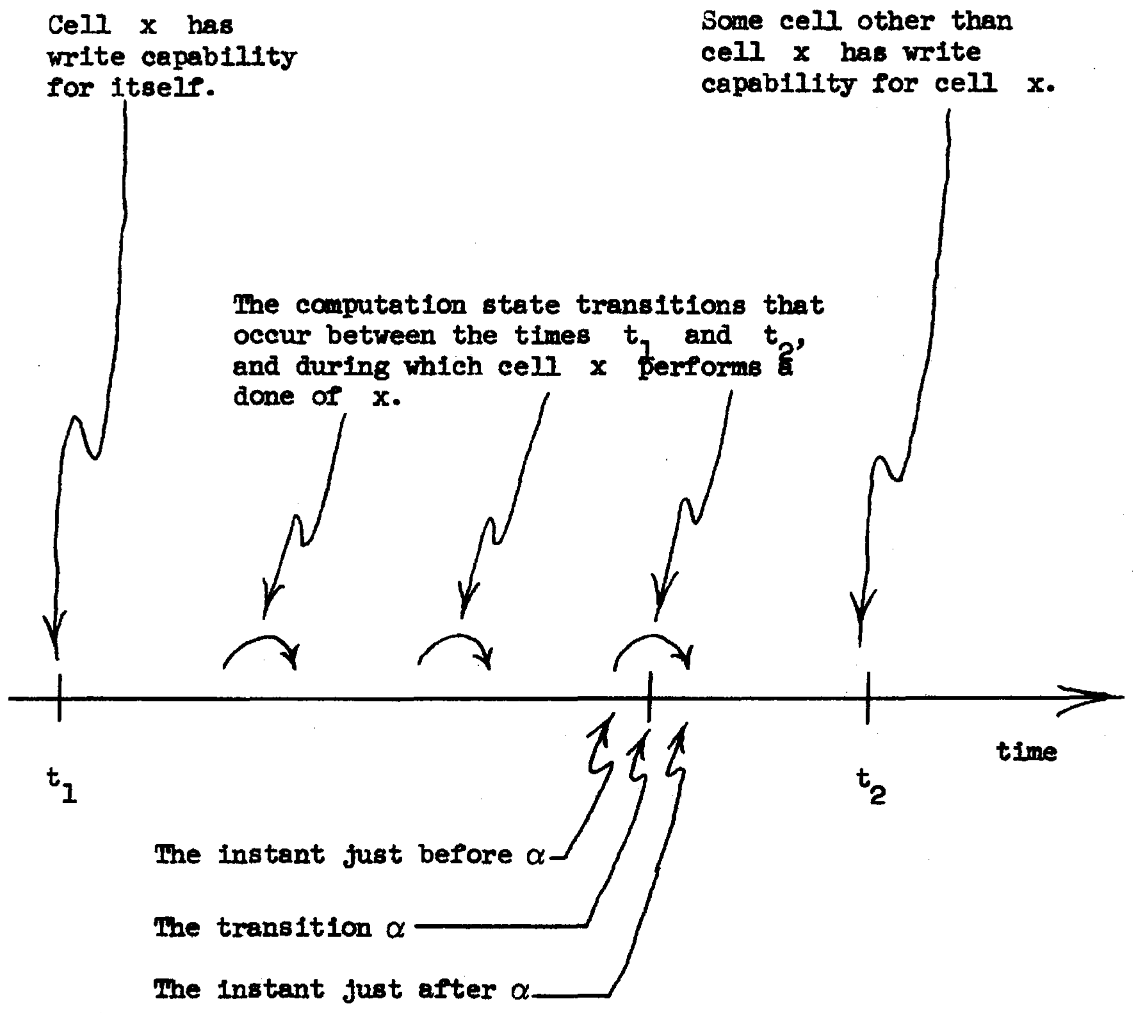

Cell x has write capability for itself.
Some cell other than cell x has write capability for cell x.
The computation state transitions that occur between the times t_1 and t_2, and during which cell x performs a done of x.
t_1
t_2
time
The instant just before α
The transition α
The instant just after α

Figure A.1. Events of interest in the demonstration of the non-redundancy of bye transactions.

Appendix B

Well-Defined MCM's

The notion of a well-defined MCM was introduced near the end of Chapter II. Presented here is both a more precise definition of a well-defined MCM, and a condition sufficient for an MCM to be well-defined.

An MCM is well-defined if and only if during each computation performed by the MCM, every computation state held by the MCM is proper for the MCM. An MCM holds a computation state that is proper for it if and only if both of the following statements are true for each cell x: (1) to cell x's content there corresponds a transaction in cell x's transaction table, and (2) if cell x is an enabled clerk cell that would upon receipt of a go pulse, perform

get of i replace f(·)

then cell i's content belongs to the domain* of f(·).

A condition will now be presented that is sufficient for an MCM to be well-defined. For each cell x, consider the set, W(x), of words cell x might hold. The set W(x) contains: (1) each word that cell x holds in any initial computation state, (2) each word that is the replacement word of any transaction listed in cell x's transaction

*The domain of a function is the set of arguments for which the function is defined.

table, (3) each word in the range[*] of the replacement function of any get listed in cell x's transaction table, and (4) each word that is the operand word of a put having operand name x and listed in any transaction table except cell x's transaction table. Thus W(x) is sure to contain every word that cell x will ever hold, but W(x) may contain some words that cell x will never hold.

An MCM is well-defined if, but not necessarily only if, both of the following statements are true for each cell x: (1) cell x's transaction table lists a transaction to correspond to each distinct word in W(x), and (2) W(x) is a subset of the domain of the replacement function of each get having operand name x and listed in any transaction table.

[*]The range of a function is the set of the function's possible output values.

Appendix C

Summary of Notation

The set-theoretic and logical notation introduced in footnotes, principally in Chapter IV, is summarized here. The number in parentheses following the explanation of a symbol gives the page on which the symbol was first used.

The subsets of the set $\{a, b\}$ are the sets: $\{\}$ (the empty set), $\{a\}$, $\{b\}$, and $\{a, b\}$. (p. 43)

The proposition $A \subseteq B$ is true if and only if A is included in B, i.e., if and only if A is a subset of B. For example, $\{a\} \subseteq \{a, b\}$, $\{a, b\} \subseteq \{a, b\}$, and $\{\} \subseteq \{a, b\}$, where $\{\}$ is the empty set. (p. 143)

The proposition $A \in B$ is true if and only if A belongs to B. For example, $a \in \{a, b\}$. (p. 139)

The proposition $A \notin B$ is true if and only if A does not belong to B. (p. 149)

The union of $\{a, b\}$ and $\{a, c\}$ is $\{a, b, c\}$. (p. 162)

The set $A \cup B$ is the union of A and B. (p. 171)

The ordered n-tuple $\langle a_1, a_2, \ldots, a_n\rangle$ is equal to the ordered m-tuple $\langle b_1, b_2, \ldots, b_m\rangle$ if and only if $n = m$, and $a_1 = b_1$, and $a_2 = b_2$, and ..., and $a_n = b_n$. Thus, for example, $\{a, b\} = \{b, a\}$ always but $\langle a, b\rangle = \langle b, a\rangle$ only if $a = b$. (p. 56)

The set $\mathfrak{D}F$ is the domain of the function F, i.e., the set of arguments for which F is defined. A sequence may be thought of as a function that takes an integer i into the i-th element of the sequence. Thus, for example, the domain of the sequence $\langle T_1, T_2, \ldots, T_m \rangle$ is the set $\{1, 2, \ldots, m\}$. (p. 139)

The proposition $\neg A$ is true if and only if A is not true. (p. 163)

The proposition $A \wedge B$ is true if and only if both A and B are true. (p. 144)

The proposition $A \vee B$ is true if and only if either A or B or both are true. (p. 199)

The proposition $A \rightarrow B$ is true if and only if A implies B, i.e., if and only if either A is false, or both A and B are true. (p. 144)

The proposition $(x)A$, where A is usually a function of x, is true if and only if A is true for every x. (p. 144)

The proposition $(\exists x)A$, where A is usually a function of x, is true if and only if there exists at least one x such that A is true. (p. 200)

The set $\{a \in A : B\}$, where B is usually a function of a, is the set of just those elements of A for which B is true. (p. 199)

References

1. Anderson, J. P. Program structures for parallel processing. Comm. ACM 8 (Dec. 1965), 786-788.

2. Bottenbruch, H. Structure and use of ALGOL 60. J. ACM 9 (Apr. 1962), 161-221.

3. Conway, M. E. A multiprocessor system design. AFIPS Conf. Proc. 24 (Nov. 1963), 139-146. Spartan Books, Baltimore.

4. Corbató, F. J., and Vyssotsky, V. A. Introduction and overview of the Multics system. AFIPS Conf. Proc. 27 (Nov. 1965), 185-196.

5. Corbató, F. J., Merwin-Daggett, M., and Daley, R. C. An experimental time-sharing system. AFIPS Conf. Proc. 21 (May 1962), 335-344. National Press, Palo Alto, Calif.

6. Crisman, P. A. (Ed.) The Compatible Time-Sharing System A Programmer's Guide. M.I.T. Press, Cambridge, Mass., 2d ed., 1965.

7. Daley, R. C., and Neumann, P. G. A general-purpose file system for secondary storage. AFIPS Conf. Proc. 27 (Nov. 1965), 213-229. Spartan Books, Baltimore.

8. Davis, M. Computability and Unsolvability. McGraw-Hill, New York, 1958.

9. Dennis, J. B. Segmentation and the design of multiprogrammed computer systems. J. ACM 12 (Oct. 1965), 589-602.

10. Dennis, J. B., and Van Horn, E. C. Programming semantics for multiprogrammed computations. Comm. ACM 9 (Mar. 1966), 143-155.

11. Dijkstra, E. W. Cooperating Sequential Processes. Mathematical Department, Technological University, Eindhoven, Netherlands, Sept. 1965.

12. Dijkstra, E. W. Solution of a problem in concurrent programming control. Comm. ACM 8 (Sept. 1965), 569.

13. D825 AOSP. Burroughs Corp., 1963.

14. Fano, R. M. The MAC system: the computer utility approach. IEEE Spectrum 2 (Jan. 1965), 56-64.

15. Glaser, E. L., Couleur, J. F., and Oliver, G. A. System design of a computer for time-sharing applications. AFIPS Conf. Proc. 27 (Nov. 1965), 197-202. Spartan Books, Baltimore.

16. Holland, J. A universal computer capable of executing an arbitrary number of sub-programs simultaneously. Proc. Eastern Joint Computer Conf., Dec. 1959, 108-113. Assoc. for Comp. Mach., New York.

17. IBM 7090 Data Processing System Reference Manual. I.B.M. Corp., 1962.

18. IBM 7090/7094 Programming Systems FORTRAN II Assembly Program (FAP). I.B.M. Corp., 1963.

19. IBM 7090/7094 Programming Systems FORTRAN II Programming. I.B.M. Corp., 1963.

20. Knuth, D. E. Additional comments on a problem in concurrent programming control. Comm. ACM 9 (May 1966), 321-322.

21. Maurer, W. D. A theory of computer instructions. J. ACM 13 (Apr. 1966), 226-235.

22. McCarthy, J., et al. LISP 1.5 Programmer's Manual. M.I.T. Press, Cambridge, Mass., 1962.

23. Programmed Data Processor-1 Manual. Digital Equipment Corp., Maynard, Mass., 1962.

24. Rabin, M. O., and Scott, D. Finite automata and their decision problems. IBM J. Res. Dev. 3 (Apr. 1959), 114-125.

25. Suppes, P. Axiomatic Set Theory. Van Nostrand, Princeton, 1960.

26. Vyssotsky, V. A., Corbató, F. J., and Graham, R. M. Structure of the Multics supervisor. AFIPS Conf. Proc. 27 (Nov. 1965), 203-212. Spartan Books, Baltimore.

Biographical Note

Earl Cornelius Van Horn, Jr. was born in Cincinnati, Ohio on January 24, 1939. He attended the Cincinnati public schools, graduating from Walnut Hills High School in June, 1957. He received the S.B. and S.M. degrees in Electrical Science and Engineering from the Massachusetts Institute of Technology in June, 1961, and February, 1963, respectively. In June, 1965, he was married to the former Sandra Ann Wallaesa of Cherry Hill, New Jersey.

In 1962 Mr. Van Horn was a Teaching Assistant for the M.I.T. Electrical Engineering Department. In 1963 he was a Research Assistant for the M.I.T. Electronic Systems Laboratory, the facilities of which he used in his Master's thesis research concerning a computer-controlled psychological experiment. Since 1963 he has been a Research Assistant for M.I.T.'s Project MAC, which has supported his current research in the field of computer system organization.

While in high school and college he pursued interests in theatrical lighting and audio technology. During summer vacations he worked for the General Electric Company on instrumentation for jet engine testing, for the Bendix Corporation in the testing of digital systems, for the Aerospace Corporation in research on digital adaptive control systems, and for Information International, Inc. on the preliminary design of a digital computer. In recent years he has had the opportunity to serve as consultant for Computer Control Company, Inc., Honeywell, Inc., and Abt Associates, Inc.

Mr. Van Horn is a member of Tau Beta Pi, Eta Kappa Nu, Sigma Xi, the Association for Computing Machinery, and the Institute of Electrical and Electronic Engineers. He is co-author of two publications:

A computer-controlled experiment in human prediction. Fourth National Symposium on Human Factors in Electronics (May 1963). Institute of Electrical and Electronic Engineers, New York. (with L. Stark)

Programming semantics for multiprogrammed computations. Comm. ACM 9 (Mar. 1966), 143-155. (with J. B. Dennis)