THE UNIVERSITY OF MICHIGAN

Technical Report 14

A SYSTEM FOR THE SOLUTION OF SIMPLE STOCHASTIC NETWORKS

Keki B. Irani
Victor L. Wallace

CONCOMP: Research in Conversational Use of Computers
F. H. Westervelt, Project Director
ORA Project 07449

supported by:

ADVANCED RESEARCH PROJECTS AGENCY
DEPARTMENT OF DEFENSE
WASHINGTON, D. C.

CONTRACT NO. DA-49-083 OSA-3050
ARPA ORDER NO. 716

administered through:

OFFICE OF RESEARCH ADMINISTRATION ANN ARBOR

September 1969

Abstract

This report details the data and program structures for a conversational programming system which translates commands describing a Markovian queueing network into a matrix of transition intensities, and which provides equilibrium distributions and related solutions of the network according to requested specifications.

Table of Contents

List of Figures

List of Figures (Continued)

List of Tables

Preface

This report is in the form of a proposal heavily dosed with tutorial matter. It was originally written in the latter part of 1967, before work had begun on the program system now known as the Queue Analyzer System (QAS). A working version of QAS [9] was completed in early 1969, adhering in all major respects to the principles laid forth in this report, except that the solution technique of Sections 6 and 7 has not been used directly.

While the system proposed here is described in terms of a specific hardware configuration, this work is not limited in use to any one such configuration. It should be noted that for a programming system to be truly efficient, it is always necessary to tailor its structures and formats in some way to the hardware used. This has been done here. However, the techniques and structures are readily adapted to other configurations.

Because of the unique nature of the system, it is instructive to publish this report in essentially its original form, with only minor attempts to bring it to correspondence with the final system. Thus, it is more a treatment of programming philosophy, specialized theory, and technique, rather than a complete documentation of an existing program. Nevertheless, all essential information required to understand the operation of QAS is here contained, as well as some information pointing to future directions.

1. INTRODUCTION

A graphical, problem-oriented system is being developed, by means of which solutions to simple stochastic networks can be obtained rapidly enough to facilitate conversational use. The problem descriptions will be constructed in network diagram form on a remote DEC 339 graphic console. Simultaneously, information concerning the construction will be sent via dataphone to a time-shared IBM 360/67, which will prepare the solutions. It is the purpose of this report to generally specify the central 360/67 programs and data structures which accomplish this latter feat. These programs will be referred to here as the Queue Analyzer System (QAS).

The networks drawn will be restricted by the pictorial language syntax to systems which can be modeled by a continuous-time, finite Markov chain, and will be solved by numerical solution of the Kolmogorov equilibrium equations. The advantage, in terms of speed and precision, of this technique over simulation procedures is documented in [2].

Because the data required by the analysis program is in a very different form from that describing a network (which is in terms of blocks and connections), a translator from the latter to the former is required. This translator, called the network compiler, is the central, and most difficult, operation carried out by this system.

In addition, there are four other major operations which the system must perform. It must <u>generate</u> the network description, which is the input to the compiler, from information contained in commands supplied by the console. It must <u>analyze</u> the resulting structure for the equilibrium state probabilities. It must <u>calculate</u> from these probabilities the specific results requested by the console user. Finally, it must accept new <u>definitions</u> of symbols used in network generation.

All of these operations are to be performed on the 360/67. The manipulation of pictures (network diagrams), the preparation and updating of display files, and the recognition of commands and interrupts from the user are all solely the responsibility of the SELMA system in the remote PDP9 which is part of the 339 console. The SELMA system [11] must also keep the QAS system informed, via the dataphone link, of any operations which concern it, and must issue commands to it to initiate major operations. The QAS system will carry out its processes under the control of the remote computer, treated as an input file (probably named *SOURCE* in MTS [4] terminology). It will send its responses to the remote computer as an output file (*SINK*). To accomplish this, QAS will have its own version of the current "displayed network" stored in the central computer memory.

In order to reduce some of the QAS system-programming problems to manageable proportions, certain limitations of capability have been accepted. These limitations are chiefly ones which limit the meanings which can be assigned to network symbols, and the manner in which the symbols can be related. Our minimal objective in this system has been to provide a system which can at least treat networks consisting wholly of queues, exponential servers, infinite sources, infinite sinks, random branches, merges, and priority branches. While many other symbols can also be treated in this system, we can by no means treat all meaningful symbols. Nevertheless, those which can be treated define a significant class of models having considerable variety and power.

2. THE BASIC COMPILATION AND CALCULATION PHILOSOPHY

A continuous-time, finite Markov chain is a stochastic process Z_t which takes on values in a finite set of points called states. If the set of states is mapped by a one-one function h onto a set of integers $\{0, 1, \ldots, N_s\}$, and if the equilibrium probability of the state mapped to the integer k is represented by π_k, then the vector $\pi = \langle \pi_0, \pi_1, \ldots, \pi_{N_s} \rangle$ of these probabilities is a solution of the equation

$$\pi U = 0 \tag{2.1}$$

where U is a matrix of transition intensities descriptive of the Markov chain. An efficient procedure [1] for solving equation (2.1) for large N_s has been in use for several years, and derives much of its efficiency from the form in which U is stored. An improved storage structure for this purpose called the matrix outline structure [3] has been devised, and proposed for this project. An adaptation of the earlier procedure to this new structure will be required, and will be called RQA-2. Its purpose is simply to calculate π when U is given.

In contradistinction to equation (2.1), the description of the Markovian system provided by the console is in the form of a network. We may say, by way of definition, that a network N consists of a set $\mathcal{E}$ of elements and a set $\mathcal{C}$ of connections,

$$N = \langle \mathcal{E}, \mathcal{C} \rangle \tag{2.2}$$

where the terms element and connection are yet to be defined but have the intuitive meaning implied by "blocks having a distinctive function" and "lines joining the blocks in particular ways." Our objective is the construction of the matrix outline structure for the matrix U from knowledge of the structure that describes the network N. It is this process which we term <u>compilation</u> of the network.

It should be noted that the state of the Markov chain will be related to the joint conditions of all of the elements in the network, so that a part of the compilation will be to form the state <u>mapping</u> from a set of multi-dimensional state vectors to the set of consecutive integers $\{0, 1, 2, \ldots, N_S\}$. It is only after this mapping is specified that the matrix U defines a Markov chain unambiguously. (The process of deriving a suitable mapping is not a trivial one, since the boundaries of the set of possible states will generally be irregular, and the mapping must be one-one on <u>consecutive</u> integers if computer memory is to be used efficiently.)

The calculation process with which we are concerned involves eight distinct phases.

(1) Generation of the Network

(2) Compilation of the Network

(3) Definition of New Elements

(4) Calculation of Equilibrium Probabilities

(5) Specification of Results

(6) Calculation of Results

(7) Documentation

(8) System Control and Supervision

The user at the console creates a network through a long series of actions, such as creation of elements or connections, evaluating parameters, changing connections, etc. The sum total of all these commands defines the network. The SELMA system (or possibly a teletypewriter) conveys this list of commands to QAS, which must generate the appropriate descriptive structure, step by step. This process is the process of network generation (phase 1, above). In definition of new elements, a series of commands is received which represents the meaning of a new symbol. This meaning must be assimilated during a definition phase (phase 3), and preserved in a form which can be used to identify future occurrences of the new symbol. Similarly, when de sired results are being specified (phase 5), the specifications are received as a stream of commands whose meaning must be collected in a specific form. This form is used by the calculator program (phase 6) to prepare a matrix which can produce the desired result when applied to the vector of equilibrium state probabilities. The documentation phase (phase 7) is one in which names are associated with structures to be saved, and structures in the system can be substituted by a named file.

Each of the phases is a distinctly different operation, requiring its own set of programs and data structures. Each is likely to proceed for a significant interval of time, and to continue at the discretion of the SELMA-generated commands which are, in turn, responsive to the user's changes of pace or purpose. Any phase can be terminated before completion and resumed later. It is the purpose of the "System Control and Supervisor" programs (phase 8) to recognize changes of phase from the information received from SELMA, and make appropriate changes in the current files contained in (virtual) memory. (Since the entire system is large, and the user's sessions are likely to be long, it would be uneconomical to keep all files resident throughout the session.) The segmented nature of the 360/67 addressing scheme seems particularly well suited to this kind of overlaying of programs.

In the succeeding sections of this report, the structures of the special files required will be described, along with the programs in the various phases which use them. In the remainder of this section, some further comments about the Supervisor and general file organization will be made. The Supervisor program is tentatively diagrammed in Figure 2.1.

The Supervisor will be assumed to reside in virtual memory throughout the session. One of its chief component parts is a program which regularly reads the commands and data arriving from SELMA. These are buffered (queued), and treated in the order in which they

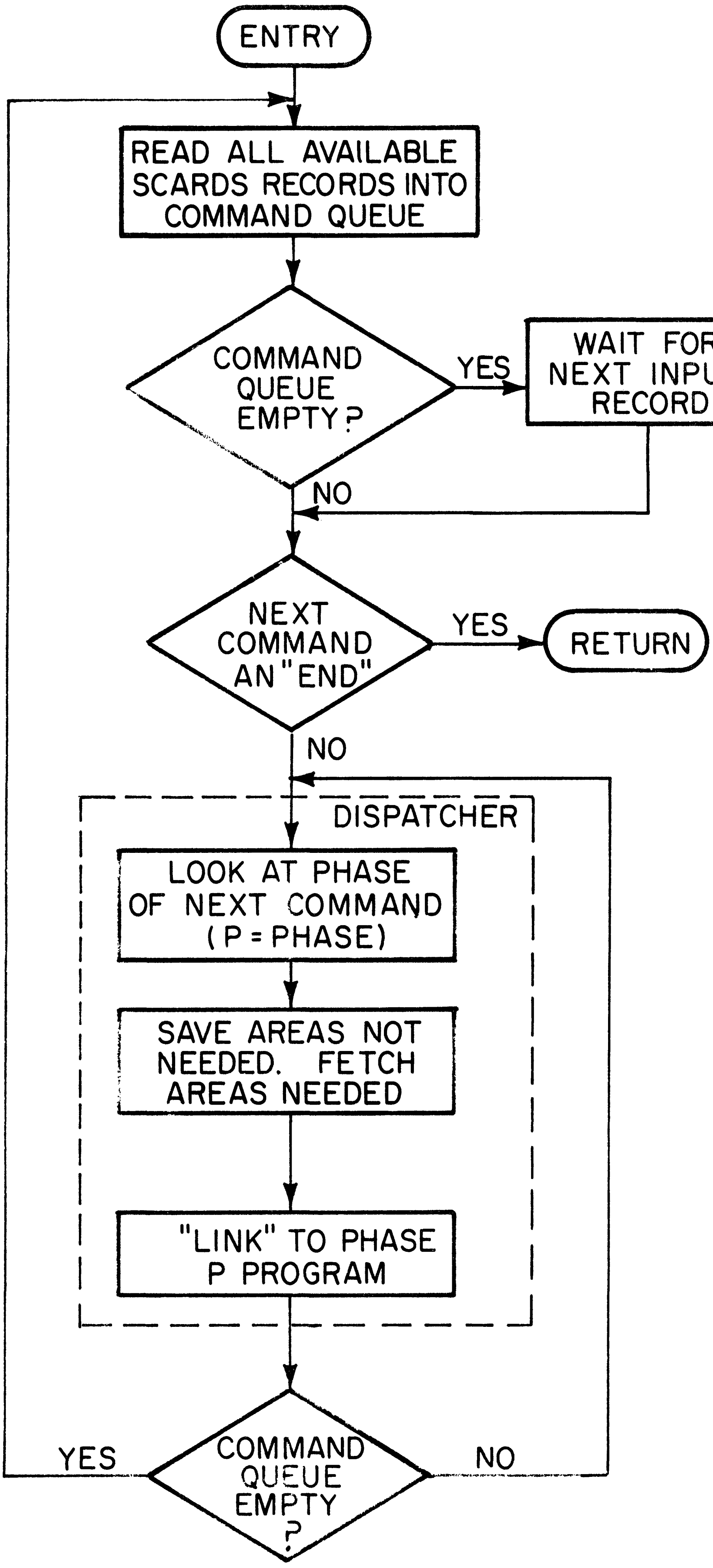

ENTRY
READ ALL AVAILABLE SCARDS RECORDS INTO COMMAND QUEUE
COMMAND QUEUE EMPTY?
YES
WAIT FOR NEXT INPUT RECORD
NO
NEXT COMMAND AN "END"
YES
RETURN
NO
DISPATCHER
LOOK AT PHASE OF NEXT COMMAND (P = PHASE)
SAVE AREAS NOT NEEDED. FETCH AREAS NEEDED
"LINK" TO PHASE P PROGRAM
YES
COMMAND QUEUE EMPTY ?
NO

Figure 2.1 Supervisor flow diagram.

arrived. Thus, QAS operates asynchronously with SELMA, lagging behind it more or less depending upon 360 scheduling, rate of SELMA command generation, and its own speed. For this reason, barring blunders of either system, care must be exercised that no command received from SELMA and queued by the Supervisor should be able to produce a "fatal" type error in QAS. QAS must not grind to a halt when SELMA commits a syntax error, for example, because by the time the error flag is conveyed to SELMA, SELMA may have gone far beyond the point of error and be unable to retrace. The result would be a dilemma resolvable only by starting again from a clean slate--an intolerable burden for the user. On the other hand, after certain commands SELMA might wait for an affirmative response before issuing more. One such case would be after the command for a network compilation. No other operations should proceed until it is known the network was compilable and did not require changes.

The Supervisor has responsibility for a table of area names, and for calling the routines for each particular phase via a LINK-type call to MTS. Every area should have its own table of special pointers to items within itself, the structure of the table being known to every program which uses it. In that way, the Supervisor does not need to keep tables other than the area tables.

[illegible]

$$P_i = \{p_{ij} : j \in J_i\}. \tag{3.2}$$

Then the element e_i can be algebraically described in the form

$$e_i = \langle \tau_i, \rho_i, P_i \rangle, \text{ for all } i \in I \tag{3.3}$$

For certain useful elements, although not for the four basic primitives, the number of parameters and the number of ports in a particular element is a function of one or more of the parameters. Since the nature of ρ_i and P_i is consequently dependent upon the value of this parameter (or parameters), this parameter (or parameters) must be supplied before equation (3.3) has meaning. Such parameter(s) will be called generation parameters because of their special role in the network generation phase. They will always be placed first in the parameter list.

Quite similarly to elements, connections are defined by their type, their parameter values, and the ports they connect. Types of connections in our basic list are "simple," "overflow branch" (sometimes called "priority branch"), "join," and "random branch." The remarks concerning parameter values of elements apply equally well to those of connections. The ports connected by connections are the ports of the elements, and each port may be involved in only one connection. Thus, if the connection set $\mathcal{C}$ is represented by the indexed set

$$\mathcal{C} = \{c_k : k \in K\}, \tag{3.4}$$

then a connection c_k is represented as

$$c_k = \langle \tau_k, \rho_k, P_k \rangle, \qquad (3.5)$$

where τ_k and ρ_k are the type and the parameter list, respectively, and P_k is a subset of the ports of the elements

$$P_k \subseteq \bigcup_{i \in I} P_i \qquad (3.6)$$

Every port is in exactly one connection:

$$P_{k_1} \quad P_{k_2} = \phi \quad k_1, k_2 \in K \qquad (3.7)$$

$$\bigcup_{k \in K} P_k = \bigcup_{i \in I} P_i. \qquad (3.8)$$

(In other words $\{P_k : k \in K\}$ and $\{P_i : i \in I\}$ are each partitionings of the set of all ports.)

The random branch is an example of a connection having a generation parameter. This parameter is the number of "branches" among which tasks can be switched. The number of ports connected by this connection is one more than the number of branches, as is the number of parameters.

The elements and connections as described here collectively describe a network. However, it is not necessary that these elements correspond precisely with the symbols used at the console. This notation represents a language for describing networks which is distinct from that used by SFLMA, and it is the PDP-9's responsibility to translate between these languages when issuing generation commands to QAS. Nevertheless, if we associate particular graphs with each element-type in QAS, then a network N can be drawn like that in Figure 3.1.

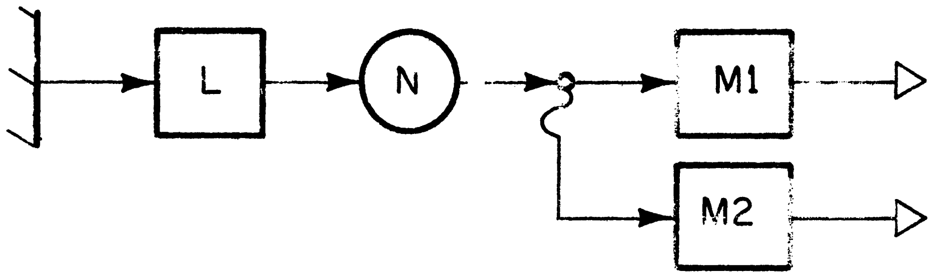


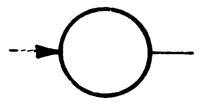
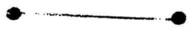
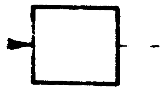
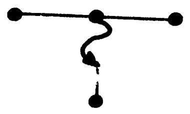
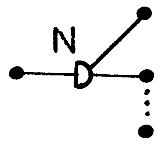
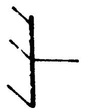
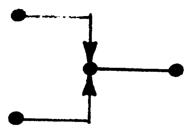
Figure 3.1

A Diagram

Such a diagram is useful in visualizing networks, and may be so used in this report. A summary of properties of the standard elements and connection types is given in Table 3.1, along with a set of graphics which may be used for them.

3.2 Syntax of Networks and Diagrams

Equations 2.1 and 3.1 through 3.8 define the syntax of a network as required by QAS. QAS must enforce rules upon the specification of networks to it which will guarantee that the network satisfies this form. In addition, SELMA may apply further rules for the form of its diagrams. These are primarily to protect the user from

Elements			Connectors		
Symbol	Type	Number of Ports	Symbol	Type	Number of Ports
	Queue	2		Simple	2
	Server (Exponential holding)	2		Overflow Branch	3
	Exit	1		Random Branch	(N+1)*
	Source	1		Merge	3

*N an arbitrary integer greater than 1.

Table 3.1 Connection Characteristics and Symbols for Standard Objects

constructions which may be compilable, but not meaningful in semantic terms.

The chief syntactic rule of QAS is that ports of elements may only be joined through connections. A second constraint is the one, mentioned in the description of connections, that requires ports to be associated with exactly one element and exactly one connection. A third constraint is that parameters of an object (element or connection) must be numerically valued (not variables) at the time of compilation, and generation parameters must be numerically valued at the time of the first mention of the object (i. e., at the time of creation). Finally, elements and connections are restricted to instances of types whose meaning has been previously defined using the notation to be described in sections 4 and 5 of this report. These restrictions merely reflect the requirements necessary to fit the data structures used by QAS.

The first constraint, part of the second constraint (the requirement that exactly one element be associated with a port), and the last constraint will all be automatically enforced because, using the command set to be described, SELMA will be unable to violate them. However, it is possible to attempt multiple connections at a port, or to leave a port unconnected, or to supply the wrong number of generation parameters, or to fail to provide values for other parameters. Either SELMA will be expected to guarantee that the proper constraints are met, so that QAS may treat their violation as fatal, or QAS will be empowered to test for them at the time their violation can cause difficulty, and demand immediate correction from SELMA. The latter course means that there must be a procedure for the QAS supervisor (the only program residing in memory at all times) to interrupt SELMA demanding a specific type of response without losing commands from the input stream. In any case, there must be provision in QAS to test these properties at appropriate times, and to return diagnostics.

SELMA will require that ports be characterized as either input or output, and that each connection type may connect inputs and outputs only in certain ways. QAS does not have this requirement, and will not test for it. Indeed, it will not even know which ports are inputs and which are outputs.

3.3 The Network Data Structure

The mathematical notation used in equations 2.2, 3.1, and 3.2 in defining networks can also be regarded as a shorthand notation for a data structure. To illustrate, let

$$\begin{aligned}
N &= \langle \mathcal{E}, \mathcal{C} \rangle \\
\mathcal{E} &= \{e_1, e_2, \ldots\} \\
\mathcal{C} &= \{c_3, \ldots\} \\
e_1 &= \langle \tau_1, \rho_1, P_1 \rangle \\
e_2 &= \langle \tau_2, \rho_2, P_3 \rangle \\
c_3 &= \langle \tau_3, \rho_3, P_3 \rangle
\end{aligned}$$

$$\rho_1 = \langle n_{\rho_1}, \ldots \rangle$$

$$\rho_2 = \langle n_{\rho_2}, \ldots \rangle$$

$$P_1 = \{p_4, p_5, \ldots\}$$

$$P_3 = \{p_5, \ldots\}.$$

Figure 3.2 describes, schematically, a data structure for this network. The meaning of the various diagram symbols is called out in Figure 3.3. We have represented sets by a one-word head of a ring, the ring linking its elements. The values of n-tuples have been represented by an n-word contiguous block, while elements belonging to a predeterminable number of sets have been represented by a like number of contiguous, one-word ring elements followed contiguously by the value of the element.

Because every set in this structure contains elements having identical form and because the programs operating on these structures will always know what kind of element is being operated upon, the use of contiguous blocks in this manner is feasible, and will result in an efficient and compact structure. If objects belonging to a variable number of sets were present, a form of association structure such as that used by Newman [5] could be easily incorporated using "nubs" and nested rings in addition to the above. So far, that has been unnecessary. Although the structure of the network is to be manipulated in the course of executing these programs, the sizes of contiguous blocks will not change during the execution. However, blocks will be created and

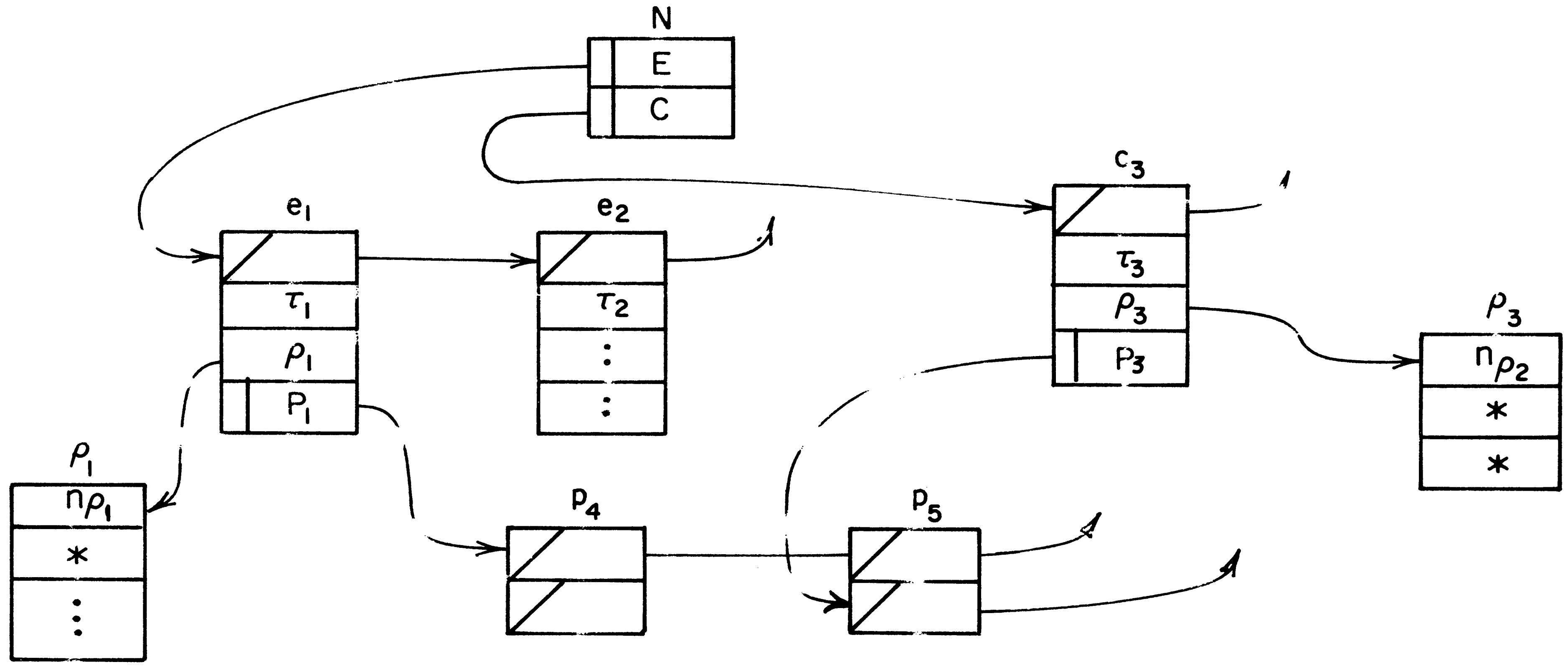


Figure 3. 2

A fragment of a completed network structure.

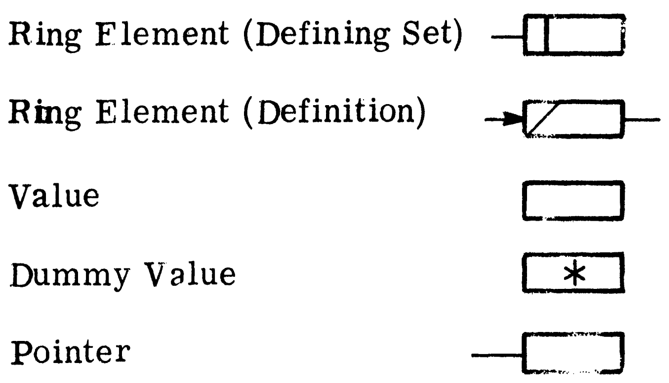


Figure 3.3

Network structure diagram notation.

destroyed, and this structural technique depends heavily upon the availability of an efficient free-core manager. The manager proposed in a separate memo [6] (Appendix A) appears to satisfy the requirements.

3.4 The "Generate" Program

The generate phase program consists of a collection of routines corresponding to the generation-commands receivable from SELMA. Upon entry from the Supervisor, the generate program examines the command and issues a call to the appropriate generation routine. The network description is received by QAS in the form of a stream of these commands. The functions of the "generation routines" carrying out the commands are summarized in Table 3.2. More generation routines may be added as use of this system becomes more sophisticated.

These routines operate entirely on an area called the network area of memory which contains the network structure, the type structure (to be described in section 4), and a symbol table. The symbol table provides the means for converting element names, connection names, and other names used by SELMA to corresponding addresses in the network area. The table grows as commands are received defining new objects. Objects also can be removed from the symbol table. The simplest form for it is obtained if SELMA assigns consecutive integers as names for objects, whereby the table consists solely

Generation Routine Name	Input Arguments	Operations Performed	Error Conditions
Create Element	Element Name, element type, generation parameter list.	Create element block and associated structure, and joins to network block. Places name into symbol table.	Failure to specify sufficient generation parameters
Connect	Connection type, generation parameter list, element name, port number, etc.	Creates connection block and associated structure, joins it to designated ports.	Failure to specify sufficient generation parameters. Attempt to connect to already connected port. No such port.
Assign Parameter Values	Element or connection name, parameter list.	Merges new parameters with parameter list block (ρ).	None
Disconnect	Element name, port number.	Destroys connection block and associated structure for connection joined to designated port. Replaces ports on unconnected port set R.	None
Destroy Element	Element name	Undoes "create element"	Failure to previously disconnect all ports of element.
Alter generate parameter.	Element name, parameter number.	Changes generation parameter, regenerates associated structure of element preserving existing values and joinings.	None

Table 3.2 Generation Routines—Summary

of a vector of pointers. The location of this vector within the network area should be permanent. The symbol table should also keep a pointer to the network block (N) as, say, its first symbol. Thus, the symbol table will keep the locations of all master objects as well as those which must be referred to by SELMA. This will permit master blocks to be moved at will for such things as documentation functions which use the network area, by merely changing a pointer in the network area's symbol table.

To facilitate discovery of unconnected ports, which would prevent execution of a compile command, a special master object R is in the symbol table representing the set of unconnected ports. When elements are created, their ports are automatically joined to this set. "Connect" commands unjoin ports from this set, and join them to a newly created connection. The symbol table and unconnected port set block are illustrated in Figure 3.4.

The "create element" and "alter generation parameter" routines will be described in detail in section 4, which treats the information needed to define types. The other generation routines in Table 3.2 should be self-explanatory.

3.5 Service Programs Required

Four sets of service programs will be needed by the generation routines. They should be FORTRAN-callable so that the generation subroutines can be written in FORTRAN. They will also be found

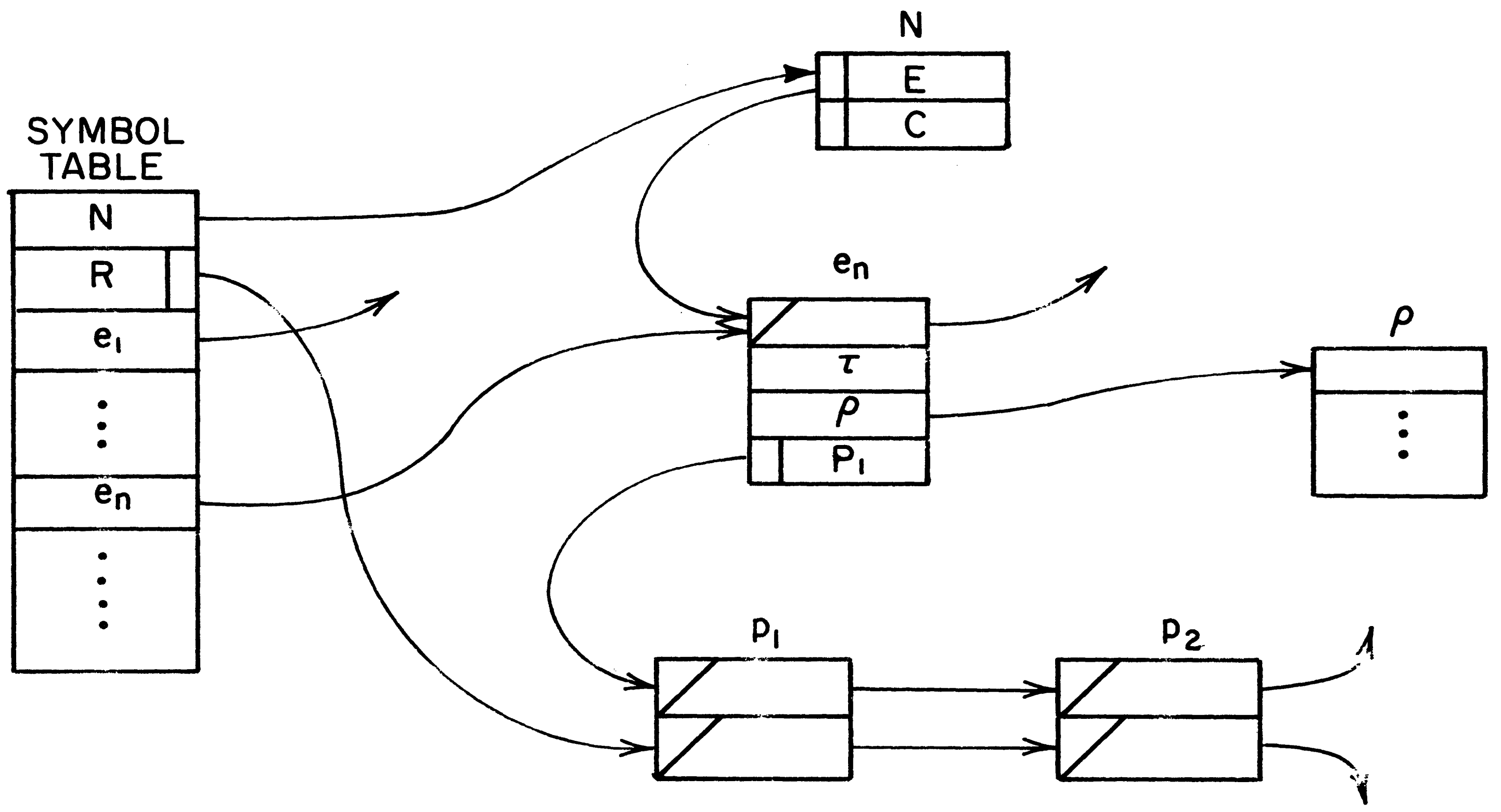


Figure 3. 4

Network structure fragment showing symbol table

and unconnected port set R.

useful to other phase programs, such as the compiler. Their functions and calls are summarized in Table 3.3.

The list of programs shown is probably not complete, and may be added to once programming is begun. In particular, the ring operators represent a subset of the commands used in Roberts' CORAL system [7], and some of the other CORAL commands may be also desirable.

Name	Input Arguments	Operations Performed	Value
Pointer Operators			
Store indirect	Pointer to location, contents value	Stores contents in location pointed to.	None
Read indirect	Pointer to location	Reads contents of location pointed to.	Contents of location
Storage Management			
Get block	Size of block	Gets block of storage from free core	Pointer to block
Free block	Pointer to block, size of block	Puts block back into free core	None
Ring Operators			
Load head	Pointer to location	Makes location the head of an empty ring	None
Insert Member	Pointer to member Pointer to ring element or head	Inserts member behind ring member or head	None
Concatenate	Pointer to ring element or head, pointer to head	Inserts all members of second argument in behind first argument. Destroys head of second.	None
Extract Member	Pointer to member	Removes member from ring	None

Table 3. 3 Service Programs for Generation—Summary

Name	Input arguments	Operations performed	Value
Find head	Pointer to member	Circles around ring to head	Pointer to head
Next member	Pointer to member	Finds next member of ring	Pointer to next member
Previous member	Pointer to member	Finds previous member of ring	Pointer to previous member
Is it head?	Pointer to member	Determines if ring member is head	"True" or "false"
<u>Symbol Table</u>			
Find pointer	Symbol of object	Finds the pointer to the object	Pointer to object
Find symbol	Pointer to object	Searches symbol table for object pointer, returns symbol	Symbol of object
<u>Area Management</u>			
Set current area	Area number	Informs all pointer routines of area in use	None

Table 3.3 Service Programs for Generation—Summary (continued)

4. PRIMITIVE ELEMENTS AND THEIR MEANING

The elements in the network structure have been described in terms of their types. Examples cited were the types "queue," "server," "exit," and "source." Each of these are called primitive types because they cannot be described in terms of any other element types. However, a description of their meaning must be given if the compiler is to have sufficient information on which to operate. In this section, a means for formal description of the meaning of a type will be given, and a data structure for conveying that meaning to the translator will be provided.

4.1 The State Set

Every element has a set of states associated with it. These states represent the various possible identifiable conditions which the element can assume. In QAS, the state set of any element will be a finite set of non-negative integer n-tuples, where n is characteristic of the element. In the case of a queue whose "maximum length" parameter is N, this set is the one-dimensional set of integers, $\{0, 1, \ldots, N\}$, representing "the number of waiting tasks." In the case of the server it is the one-dimensional set of integers $\{0, 1, 2\}$ representing the "idle," "busy," and "holding" conditions, respectively. Finally, in the case of both the exit and source it is the set $\{0\}$, a condition representing "readiness to receive (or send) a task." Although all four of the basic element types have one-dimensional state sets, higher dimensions are possible for yet unnamed elements.

The things which most characterize elements are the stimuli which can produce changes in state, and the responsive changes. These are characterized by the so-called _events_ of the element-type. Some of the stimuli are self-generated, as, for example, the "service completions" of a server. Others are external stimuli, as the occurrence of "inputs of tasks" at a port of an element. The stimuli and responses of the former will be described by _autoevents_, while those of the latter will be described by _exoevents_.

4.2 Autoevents

By definition, an autoevent ξ_ℓ of an element e_i having a state set S_i is a triple

$$\xi_\ell = < b_\ell, g_\ell, \mu_\ell > \qquad (4.1)$$

where:

b_ℓ is a subset of S_i

g_ℓ is a constant n-tuple, such that for each $x \in b_\ell$, $x + g_\ell$ is in S_i

μ_ℓ is a positive real number

ℓ is an index in some index set L_i.

Here b_ℓ is called the _autocondition set_ of the event ξ_ℓ, and represents a set of states for which the event is possible; the constant μ_ℓ is called the _rate_ of the event, and represents the probability intensity of its occurrence; the constant n-tuple g_ℓ is called the <u>increment</u> of the event, and represents the change in state which results from the occurrence of the event.

To illustrate, consider the "service completion" event ξ_{σ_1} for a server having mean service rate (parameter) γ. The event can occur only in the busy state, state 1, and when in that state it occurs with probability intensity γ. The event causes a change from state 1 to the holding state, state 2, so that the change in state is unity. Thus,

$$\xi_{\sigma_1} = < \{1\}, 1, \gamma > . \qquad (4.2)$$

It is possible for an element to have more than one autoevent. Consequently, for every element there is a set of autoevents, called the <u>autoevent set</u>

$$\Xi_i = \{\xi_\ell : \ell \in L_i\} \qquad (4.3)$$

of the element e_i where the index sets L_i are disjoint over all i in I. Frequently, when the format of equation 4.1 is insufficiently general to describe a particular stimulus and its response, the desired description can be obtained by splitting the description up into several autoevents.

4.3 <u>Exoevents</u>

By definition an exoevent ζ_m at a port p of an element e_i having a state set S_i is also a triple

$$\zeta_m = < b_m, g_m, \pi_m > \qquad (4.4)$$

where:

b_m is a subset of S_i

g_m is a constant n-tuple such that for each $x \in b_m$, $x + g_m$ is in S_i

π_m is a probability

m is an index in some index set.

Here b_m is the endocondition set of the event, and represents a set of states for which the event is possible (e.g., if the port p is an input port, b is the set of states for which an input of a task can be accepted); the probability of the event π_m is the probability, given that the stimulus (e.g., an input of a task) has occurred and that the element is in a state of b_m, that the change in state produced is equal to g_m, the increment of the event.

An example of an exoevent is the "input of a task" event (call it ζ_{σ_2}) at the input port p of a queue whose maximum length is N. An input can be allowed only when the state is less than N, and it results, with certainty, in an increase of the state by unity. Thus

$$\zeta_{\sigma_2} = < \{0, 1, \ldots, N\text{-}1\}, 1, 1 >, \qquad (4.5)$$

for $N \geq 1$.

For every port p of an element e_i, there is an exoevent set $Z_i(p)$, representing all the possible responses to the external stimulus at the port p. The function Z_i will be called the <u>exoevent function</u> of the element e_i. The <u>exoevent set</u> is a set of exoevents whose probabilities add up to zero or one for each state of the element. Let

$$Z_i(p) = \{\zeta_m, \; m \in M_i(p)\}, \; p \in P_i, \; i \in I, \tag{4.6}$$

and

$$\zeta_m = < b_m, g_m, \pi_m > , \tag{4.7}$$

where the b_m, g_m, π_m have the forms appropriate to exoevents of an element, and where the $M_i(p)$ are disjoint for all p and i. Then

$$\sum_{\substack{m: \\ x \in b_m}} \pi_m = 0 \text{ or } 1 \tag{4.8}$$

for each $x \in S_i$ where S_i is the state set of the element. Those states for which this sum is zero are states for which the external stimulus cannot occur, such as, for example, the states where an input task cannot be accepted.

4.4 <u>Type Definition</u>

The meaning of the "type" τ_i of a particular element e_i is consequently conveyed by the triple $< S_i, \Xi_i, Z_i >$, so that we can write

$$\tau_i = < S_i, \Xi_i, Z_i > \tag{4.9}$$

for all $i \in I$. The above rules for determining the state set, autoevent set, and exoevent function have been applied for instances of each of the four basic element-types, and these properties have been summarized in Table 4.1. In each of the first two cases, the port p_1 is the input port, and port p_2 is the output port. It is interesting to note that the exit and source elements are identical in these terms. Consequently, they will both be treated as instances of a single element-type, the source-exit. Other element-types are, of course, possible provided their stimuli and responses can be completely described by events of the form described.

The information conveyed in Table 4.1, and extensions of it if more element-types are defined, must be made available to the compiler. In particular, there must be a data structure which describes the type τ_i of an element e_i in literal form, with all dependencies upon parameters evaluated and readily available. In addition, the compiler in the course of its work will be creating more elements which are not instances of these basic types but are equally described by triples $< S_i, \Xi_i, Z_i >$. The structure for providing this information to the compiler is added, at the appropriate times, to the network structure by use of the τ-line in the element blocks. This structure is illustrated schematically in Figure 4.1 for an element e_1 such that

$$P_1 = \{p_1, p_2\} \tag{4.10}$$

$$\Xi_1 = \{\xi_1, \xi_2\} \tag{4.11}$$

$$Z_1(p_1) = \{\zeta_3, \zeta_4\} \tag{4.12}$$

$$Z_1(p_2) = \{\zeta_5\}. \tag{4.13}$$

Element Type	Parameter Value	State Set	Autoevent	Exoevent Function: Port	Exoevent Function: Exoevent	
Queue	N	$\{0, 1, \ldots, N\}$	None	p_1	$< \{0, 1, \ldots, N-1\}, 1, 1 >$,	for $N > 0$
					None	for $N = 0$
				p_2	$< \{1, 2, \ldots, N\}, -1, 1 >$,	for $N > 0$
					None	for $N = 0$
Server	γ	$\{0, 1, 2\}$	$<\{1\}, 1, \gamma>$	p_1	$< \{0\}, 1, 1 >$	
				p_2	$< \{2\}, -2, 1 >$	
Exit	None	$\{0\}$	None	p_1	$< \{0\}, 0, 1 >$	
Source	None	$\{0\}$	None	p_1	$< \{0\}, 0, 1 >$	

Table 4.1 Summary of Basic Elements' State Sets and Events

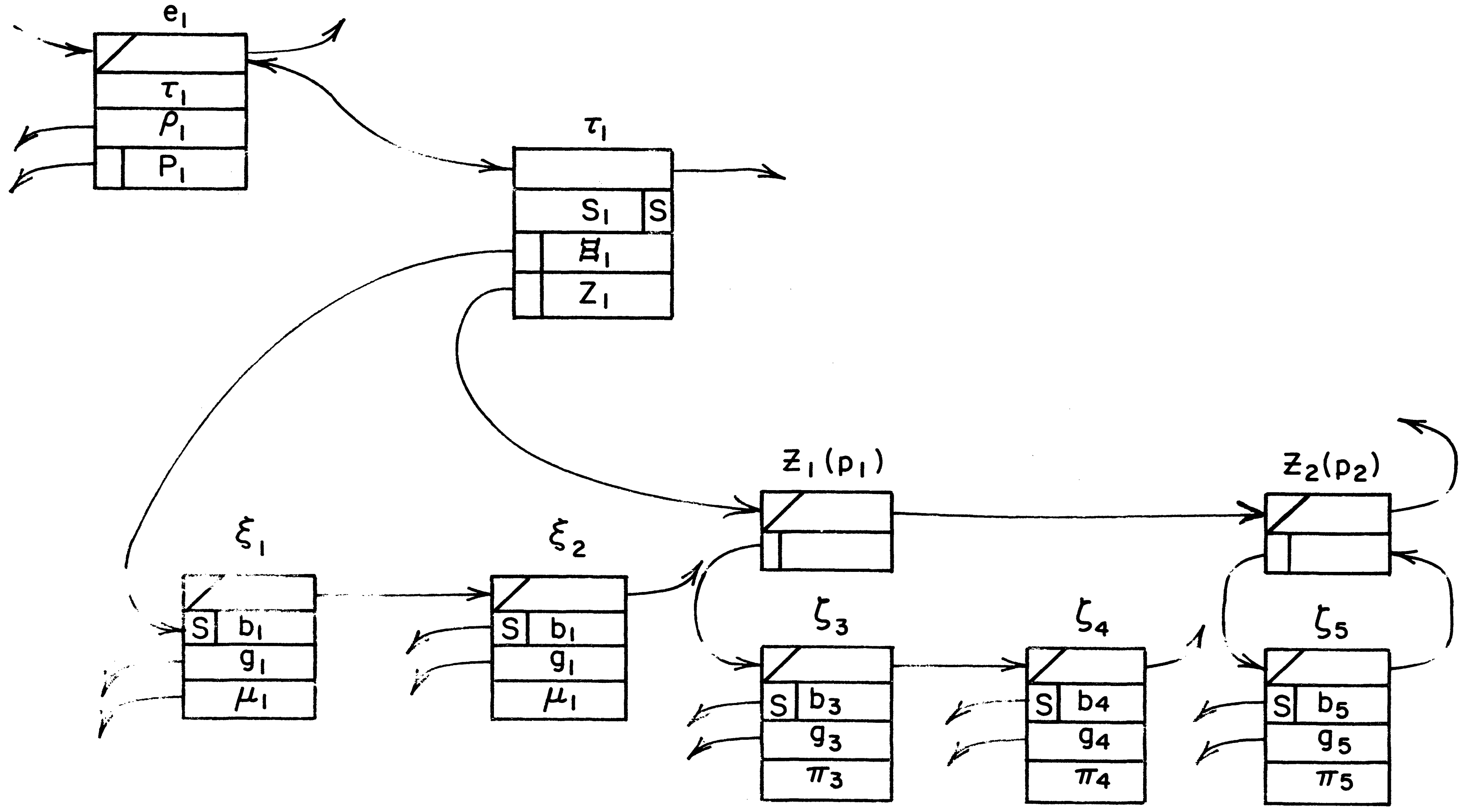


Figure 4.1

Fragment of network structure showing literal type definition.

The sets of states S_1, b_1, b_2, ..., b_5, and the increments g_1, ..., g_5 are not shown; their form will be discussed in section 4.6. The block τ_1 is referred to as a type block.

It should be noted, however, that at the time the element block was created (during generation phase) the type line τ in the element block contained a name of the type, rather than a pointer to a structure like that shown in Figure 4.1. It is not possible during element creation to form this type structure because the parameter values are not yet known. (It should be noted in Table 4.1 that variables N and γ were necessary to represent the state set and the events.) It is also not completely desirable to supply this structure immediately, even if it were possible, because it would considerably increase the amount of storage required during generation. This information is needed only during compilation, or if compilation has been partially performed.

In any case, the τ-line in the element block has two possible meanings, and during compilation either meaning may appear in the line. Since a pointer requires only 24 bits, and the name (a type number, actually) can be contained in an 8-bit byte or less, it is not difficult to distinguish between the two cases. By reserving the name "00" (hexadecimal) to literal valued types (i.e., those for which a structure is present), and using the format shown in Figure 4.2 a simple routine can determine whether name or pointer is present. If the pointer to the type structure is present, we say the type is literal valued. Otherwise, the type is said to be *named.*

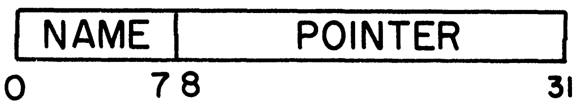


Figure 4.2 Format of τ-line of element.

4.5 *Type Structure Generation*

When the compiler creates an element itself, it will always have a literal type. When the element is created by the "create element" routine during generation phase, it will have a named type. The conversion of the named type to a literal type will be carried out only when the type structure of the particular element is needed by the compiler. The compiler requests the literal type of an element through a routine called the *type finder.* The element is specified and if the element is already of literal type, the pointer to the type block is immediately returned. If the element has a named type, the finder calls a particular *type-evaluation routine* which creates the type structure, using the parameter values of the element. There is one type-evaluation routine for each type name. These routines are written whenever a new type name is added to the repertoire of QAS.

The operation of the type structure programs is summarized in

Table 4.2. These programs are used during compile phase, and require the presence of the working network structure only.

4.6 Representation of Sets of States and Increments

The set S for each element is a set of n-tuples. The autocondition-sets and endocondition-sets of the events of the element are subsets of this set. For reasons of efficiency in the operation of the compiler, it must be possible to rapidly form unions, intersections, and differences of these sets. For efficient use of storage, these sets cannot be represented by rings of members; their number is too great.

For these reasons, a special data structure must be used for representing sets of n-tuples. The basic unit in this structure will be a structure representing rectangular sets of n-tuples, sets which are Cartesian products of sets of consecutive integers. Any state set, or subset, will be represented by a union of rectangular sets. This structure is illustrated in Figure 4.3. The basic line describing the set is a single word whose format is shown in Figure 4.4. This word contains the dimension, n, of the n-tuples, and is a ring head. The upper byte of the ring elements pointed to is zero, and, since sets of dimension zero are not permitted, the head is recognized by the non-zero value of this byte. The ring elements define rectangular blocks, with the ring itself representing the union of the rectangles represented. There are n + 2 half-words in each rectangle block (the first two being the ring element

Routine Name	Input Arguments	Operations Performed	Error Conditions	Remarks
<u>Routines (Compiler Phase)</u>				
Type Finder	Pointer to element block	If element type is literal, returns pointer to type structure. If element type is named (primitive), creates a type structure with parameter dependencies evaluated, links it to element block, and returns a pointer to it.	No such type name.	Calls type evaluation routine for named element type.
Type evaluators (one for each type)	Parameter list	Creates type structure for a particular type element, calculates values for all parameter-dependent lines in structure, and returns a pointer to the type block.	None	Calls parameter tester.
<u>Service Programs Required</u>				
(Pointer Operators)				
(Storage Manager)				
Parameter tester	Parameter list	Determines that all parameters are present, and returns.	Some parameter value or values not present. Fatal.	

Table 4.2 Type Structure Routines for Compiler

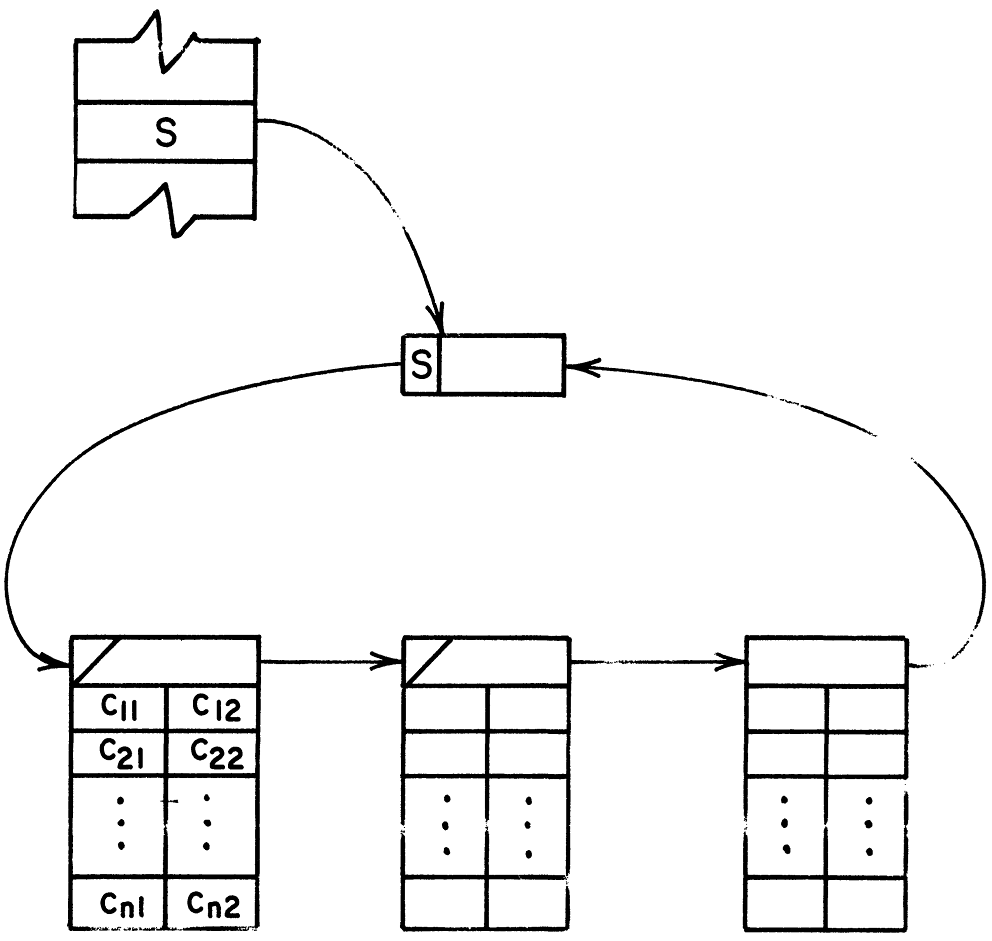


Figure 4. 3 The set structure for n-tuples.

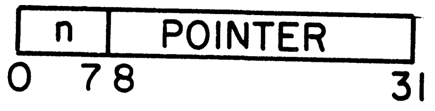


Figure 4. 4 The n-tuple set head word.

word). Each half-word after the first word consists of two bytes, the first of which supplies the lower bound on the rectangle in one of the dimensions, and the second of which supplies the upper bound. Set operators for intersection, union, and difference can be readily designed for this structure.

Increments (the g-lines in events) will be represented by the same type of head element as for sets, showing the dimension and a pointer. In this case, however, the pointer will be to a block of n bytes, each giving the increment in its corresponding dimension.

All blocks in these two structures should be rounded to full words in their allocation.

A set of operators for these two structures will be required for such things as intersection, union, sum, etc. One special operator, called back-projection, is needed by the compiler. It has two operands, a set and an "increment," and forms the structure for a set whose members are the members of the set operand <u>minus</u> the value of the "increment" operand. This set will be represented by the symbol $\eta(S, g)$, where S and g are set and increment operands, respectively.

5. THE COMPILATION PROCESS

The compiler is a program, called by a single command from SELMA, which transforms the network structure into a new structure suitable for numerical analysis. The new structure represents the matrix U of transition intensities (see eq. 2.1) in a compact form, so that the iteration may proceed rapidly without extensive paging (which would cause response to the "solve" command to be uncomfortably slow).

The compiler operates destructively upon the network structure, so that a copy of the network structure must be saved prior to the compilation. This function, which calls upon the documentation phase programs, is carried out by the QAS supervisor when it detects a "compile phase" command. In order to distinguish between the network structure and the workspace of the compiler (which is identical to the network structure when compilation begins), we will refer to this workspace as the working network area.

The compiler makes use of two areas, the working network area and the transition-table area. This latter area is the place where the new structure is placed after the compilation is finished. It is, obviously, one of the areas used by the analysis procedure during the analysis phase. This area is required only at the last stage of compilation, and is created during the compile phase. (If a translation table area already exists at the time a compile command is given, its contents will be replaced by the new transition table. Hence, a documentation phase "save transition table area" command must be given by SELMA before the compile if the old information is to be saved.)

The procedure for compiling the network involves a successive absorption of the connections. Each connection is considered in turn, and it and the elements it connects are replaced by a single equivalent element. This process continues until no connections remain. To illustrate, observe the network of Figure 5.1a. The connections of this network will be absorbed in the (arbitrary) order shown by the encircled numbers. The compilation proceeds as schematically indicated in Figure 5.1b, c, d, e, finally resulting in a single element having no ports, which is equivalent to the original network. This procedure provides a systematic means for tracing the influence of each autoevent upon the entire network's behavior. The resulting element consists of a long list of autoevents describing the probability intensity of every change in state which is possible. The translation of this information into the form of the transition table is then a relatively simple operation.

Figure 5.2 shows the flow diagram for the compiler. After it is invoked by a "compile" command, the compiler first determines that no ports remain in the network which are not connected to another port, by testing the ring whose head was labeled R in the symbol table of the (now) working network structure. If this test is successful, the

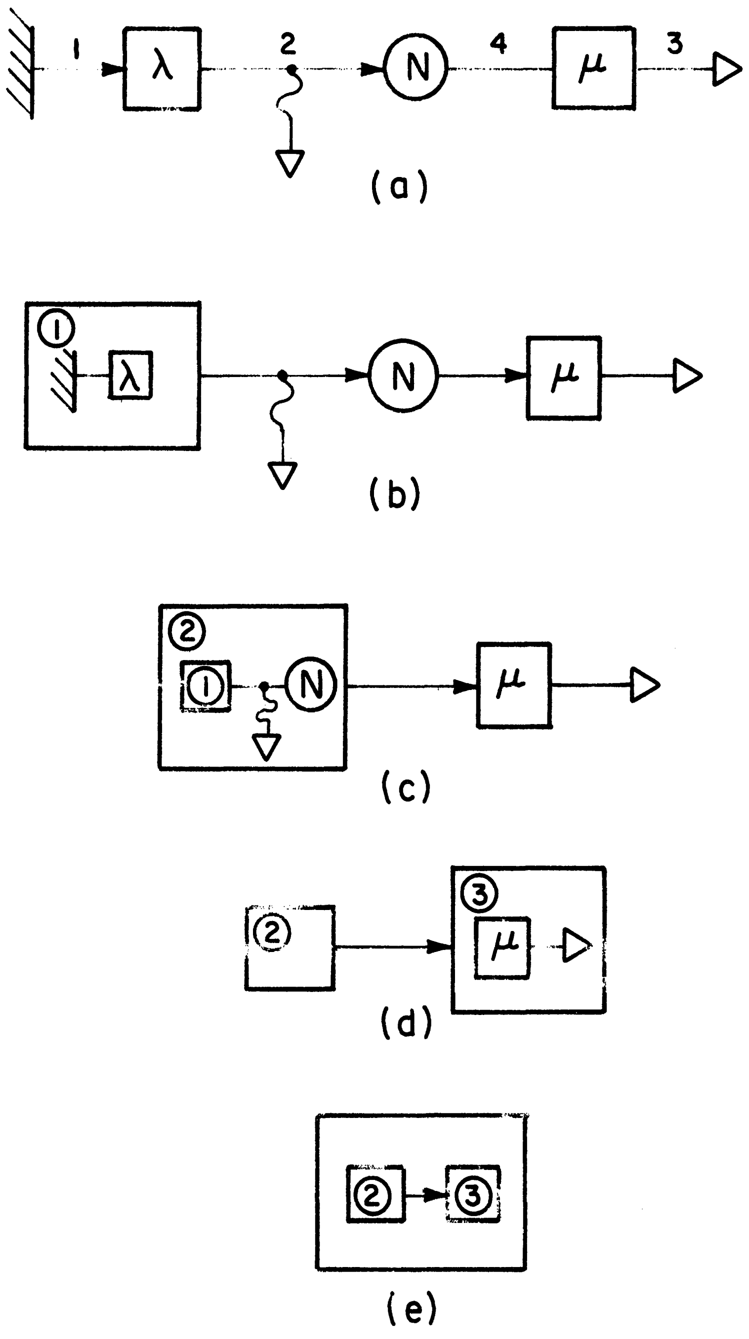


Figure 5. 1

Illustrating the compilation procedure.

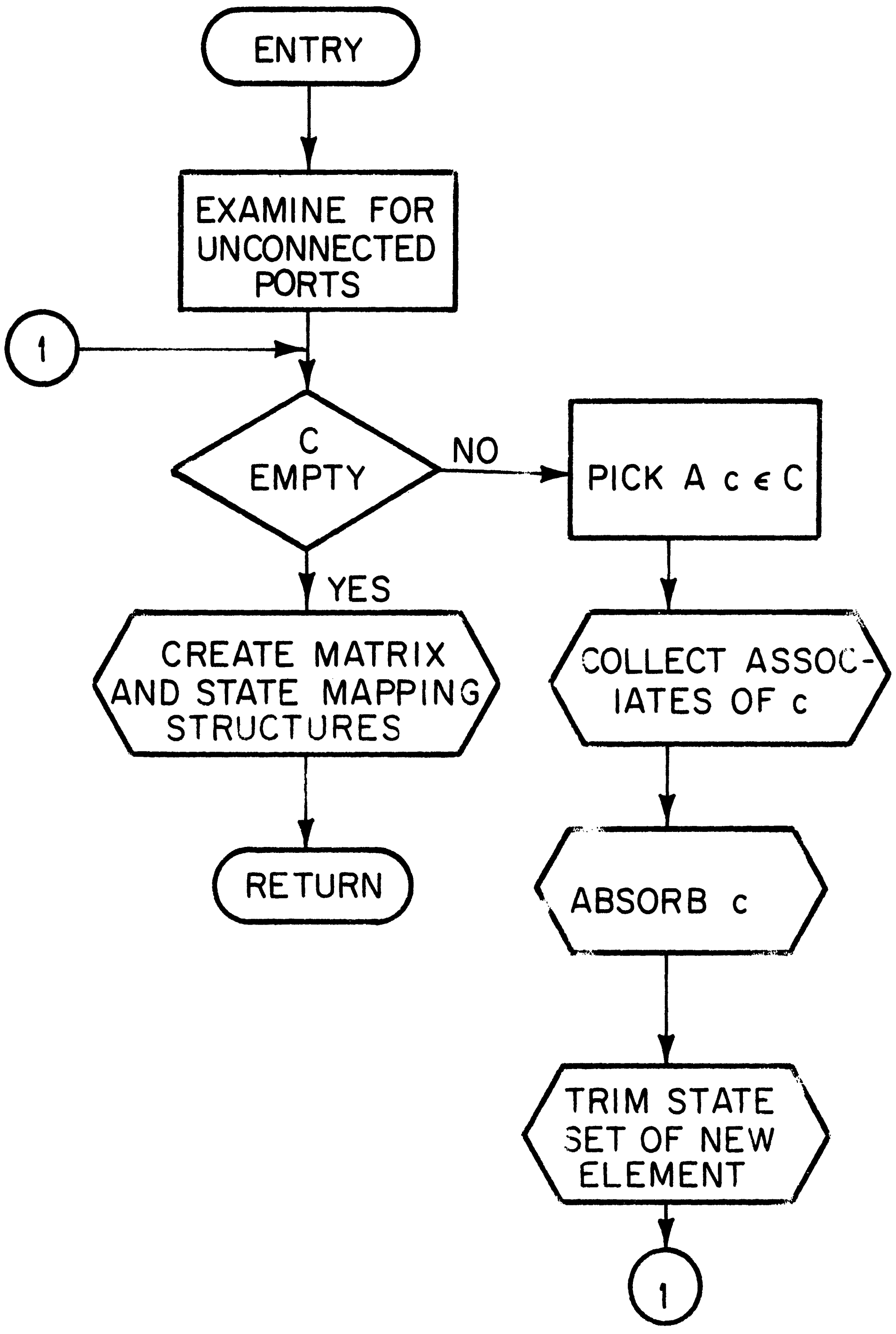

ENTRY
EXAMINE FOR UNCONNECTED PORTS
1
C EMPTY
NO
YES
PICK A c ∈ C
CREATE MATRIX AND STATE MAPPING STRUCTURES
COLLECT ASSOC-IATES OF c
RETURN
ABSORB c
TRIM STATE SET OF NEW ELEMENT
1

Figure 5.2

The compiler flow diagram.

compiler proceeds through a loop, successively absorbing connections until none remain.

The absorption of a connection has been divided into three distinct steps. In the first step, the elements joined by the connection to be absorbed (we will call these elements the associates of the connection) are collected into a single equivalent element having ports corresponding to each of those of the associates. This collected element then replaces the associates. In the second step the connection, which now joins only ports of a single element, is absorbed to create still another element. This time the element is equivalent to the collected element with its connection, and the final replacement takes place. (The two step operation eliminates the need to treat connections between ports of a single element differently from those between ports of different elements.) In the third step, the state set is reduced to eliminate certain common types of transient states.

The operation of the compiler and its major subprograms is summarized in Table 5.1. The remainder of this section will discuss the subprograms in more detail. The subprogram which creates the matrix and state mapping structures will be discussed in Chapter 6.

5.1 Collection of Associates

The operation called collection of the associates of a connection c has a simple algebraic interpretation. What it entails is the creation of a new element e* whose port set is the union of the port sets of the

Program Name	Input Arguments	Operations Performed	Output Parameters	Error Conditions
Compiler	Pointer to Network block	Checks that network is complete, then compiles, producing a matrix-outline structure for the network and a state mapping structure.	Pointer to matrix outline. Pointer to state mapping structure.	Unconnected ports present in network.
Major Subprograms				
Collect Associates	Pointer to connection	Calls for type finder for all associate elements, creates a single new equivalent element, replaces original associates by new element, links up the state variable structure.	None	1. (Type-finder detects missing parameter values.) 2. No associates. (Recovery: destroy connection)
Absorb Connection	Pointer to connection	Replaces element and self-connection by a single element.	None	Not a self-connection.
Trim State Set	Pointer to element	Removes certain transient states from the state set and the event definitions.	None	None

Table 5.1 Summary of Operation of Compiler and Its Major Subprograms

Program Name	Input Arguments	Operations Performed	Output Parameters	Error Conditions
Create state and transition matrix structures	Pointer to network block	Creates "state area" and "matrix area," and generates appropriate structures in them to describe the state, state mapping, and transition matrix. (Creation of areas done by call to QAS supervisor.) Clears working network area.	None	Network has two or more isolated parts.

Figure 5.1 (continued) Summary of Operation of Compiler and Its Major Subprograms

associates, whose state set is the Cartesian product of the state sets of the associates, and whose events are adjusted to correspond to the new state set.

Let the set of elements to be "collected" be E*, where

$$E^* = \{e_i : i \in I^*\}, \tag{5.1}$$

and let the collected element (the result of this operation) be e*

$$e^* = < \tau^*, \rho^*, P^* > \tag{5.2}$$

where

$$\tau^* = < S^*, \Xi^*, Z^* >, \tag{5.3}$$

the parameter set ρ^* is null, and the new port set of e* is

$$P^* = \bigcup_{i \in I^*} P_i. \tag{5.4}$$

Let $I^* = \{i_1, i_2, \ldots, i_{n^*}\}$, and select an arbitrary ordering $< i_1, i_2, \ldots, i_{n^*} >$ on the members of I*. The new state set can then be written

$$S^* = S_{i_1} \times S_{i_2} \times \ldots \times S_{i_{n^*}}, \tag{5.5}$$

where the product $A_1 \times A_2$ of two sets A_1, A_2 of n-tuples is defined as a set of n-tuples having a dimension which is the sum of the dimensions of A_1 and A_2, and for which the members of $A_1 \times A_2$ consist of all concatenations of members of A_1 with members of A_2. For example,

$$\{<1,1>,<2,1>\} \times \{<3,2>,<3,3>\} = \{<1,1,3,2>,<1,1,3,3>,$$
$$<2,1,3,2>,<2,1,3,3>\} \qquad (5.6)$$

(Notice that this operator is not precisely the <u>Cartesian product</u> operator, for which the example would give the set $\{<<1,1>,<3,2>>,<<1,1>,$ $<3,3>>,<<2,1> <3,2>>,<<2,1>,<3,3>>\}$.)

The autoevent set Ξ^* is the union of autoevents of the elements, suitably modified to account for the changed state space. In particular, let $\xi_\ell = <b_\ell, g_\ell, \mu_\ell>$ be an autoevent of an element e_{i_α} in E* (i.e. $\ell \in L_{i_\alpha}$, $i_\alpha \in I^*$). Then ξ_ℓ will be represented by the new autoevent

$$\xi_\ell^* = <b_\ell^*, g_\ell^*, \mu_\ell> , \qquad (5.7)$$

where

$$b_\ell^* = S_{i_1} \times \ldots \times S_{i_{\alpha-1}} \times b_\ell \times S_{i_{\alpha+1}} \times \ldots \times S_{i_{n^*}} \qquad (5.8)$$

and

$$g_\ell^* = <0, \ldots, 0, g_\ell, 0, \ldots, 0> \qquad (5.9)$$

with g_ℓ concatenated with zero-valued n-tuples in a position corresponding to S_α in S*. The set Ξ is

$$\Xi^* = \{\xi_\ell^* : \ell \in L_i, \ i \in I^*\}. \qquad (5.10)$$

To simplify notation, eq. (5.8) will be rewritten

$$b_\ell^* = S^* \times_{i_\alpha} b_\ell, \qquad (5.11)$$

defining "$\times_{i_\alpha}$" as an operator meaning "restrict the projection of S* in the components corresponding to S_{i_α} to the set b_ℓ". In a similar vein, eq. 5.9 will be rewritten

$$g_\ell^* = S^* \odot_{i_\alpha} g_\ell, \tag{5.12}$$

defining "$\odot_{i_\alpha}$" as an operator meaning "extend g_ℓ into the set S^*, with projection g_ℓ in the components corresponding to S_{i_α}, and projection zero in all other components."

For each port $p_j \in P_i$, $i \in I^*$, the exoevent function Z^* is given, similarly by

$$Z^*(p_j) = \{\zeta^*_m : m \in M_i(p_j),\ i \in I^*\} \tag{5.13}$$

where

$$\zeta^*_m = \langle b^*_m, g^*_m, \pi_m \rangle \tag{5.14}$$

$$b^*_m = S^* \times_i b_m \tag{5.15}$$

$$g^*_m = S^* \odot_i g_m \tag{5.16}$$

and where $M_i(p_j)$ is the index set, defined in section 4.3, of the original elements. Each $Z_i^*(p_j)$ is readily shown to be an exoevent set, satisfying the constraint on π_m given in eq. 4.8, provided only that $Z_i(p_j)$ was.

The <u>collection routine,</u> called by the compiler, makes the changes in the working network structure corresponding to the above equations. Its input is a pointer to a connection, and its output is a pointer to an element e* which is the "collection" of the connection's associates. The original elements (the associates) are destroyed.

This program, which is flow-charted in Figure 5.3, has three main parts. The first finds the associates, removes them from the E ring, and joins them to a newly formed ring E*. The second forms the state set S* of the collection E* using the natural order of the E*-ring as the reverse order (right to left) of the products of the S_i, $i \in I^*$. (The order is reversed for consistency with later operations, since the port set P* and event sets Ξ^* and Z* will be most easily formed by repeatedly joining successive components immediately behind the head.) In the course of this activity the types will be evaluated by calls to the "type finder" (cf. section 4.5). The third part successively extends the event sets and then accumulates both the port sets and the event sets of each element, destroying the elements as execution proceeds until E* is empty.

Figure 5.4 shows part of the structure upon entry to the collector. The encircled symbol represents a "bug" or pointer retained in the collector program. Figure 5.5 shows that part of the structure after the first two parts have been executed and the program has progressed to the point marked β in the flow diagram (Figure 5.3). Figure 5.6 shows the same part of the structure upon return from the collector. The structure of the state sets has been omitted from the diagrams and will be treated separately later, in section 5.6.

The first two parts of this program need no further discussion. Part three (point β in Figure 5.3) begins by taking the first element

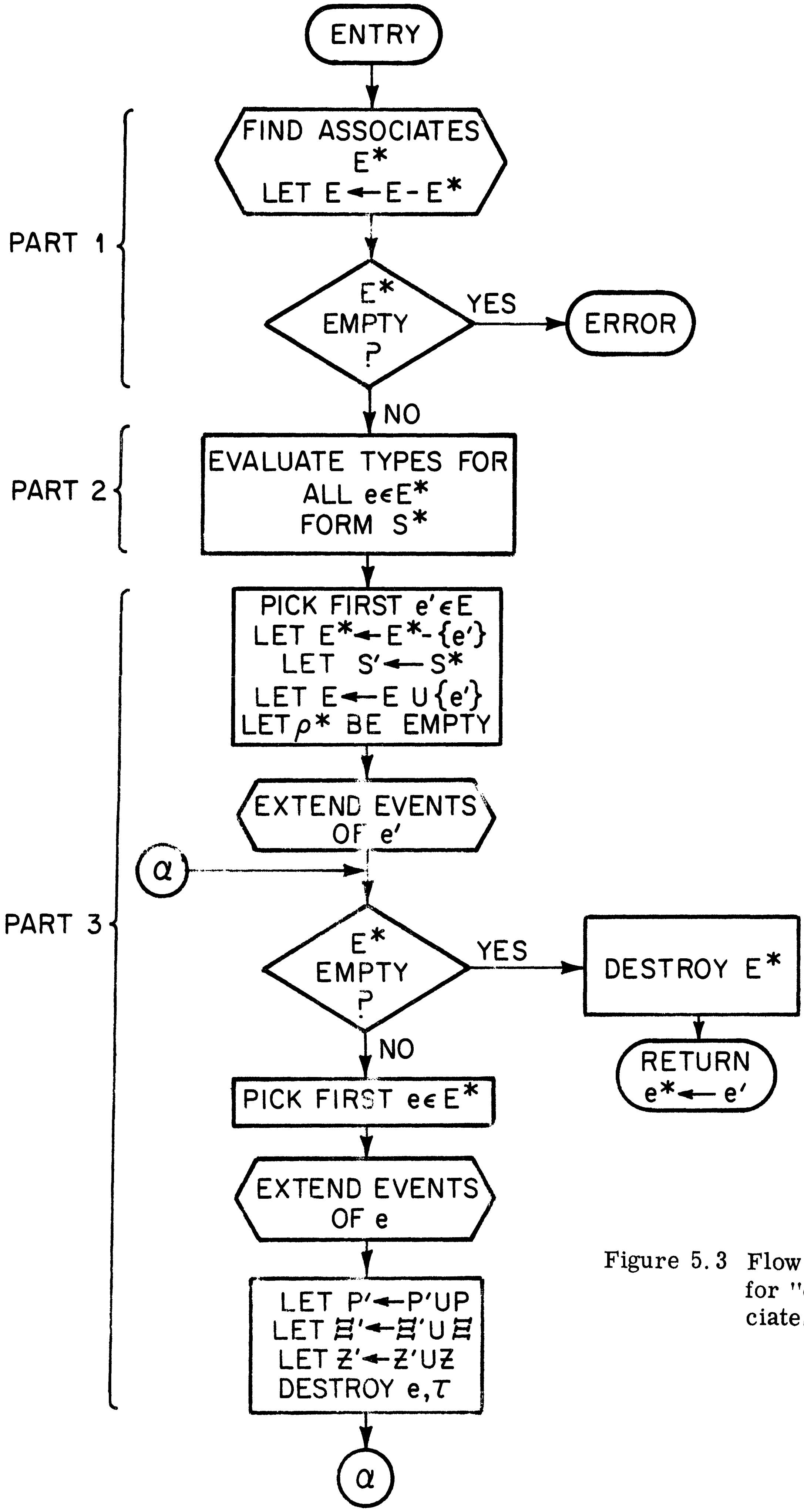


Figure 5.3 Flow diagram for "called associates" routine

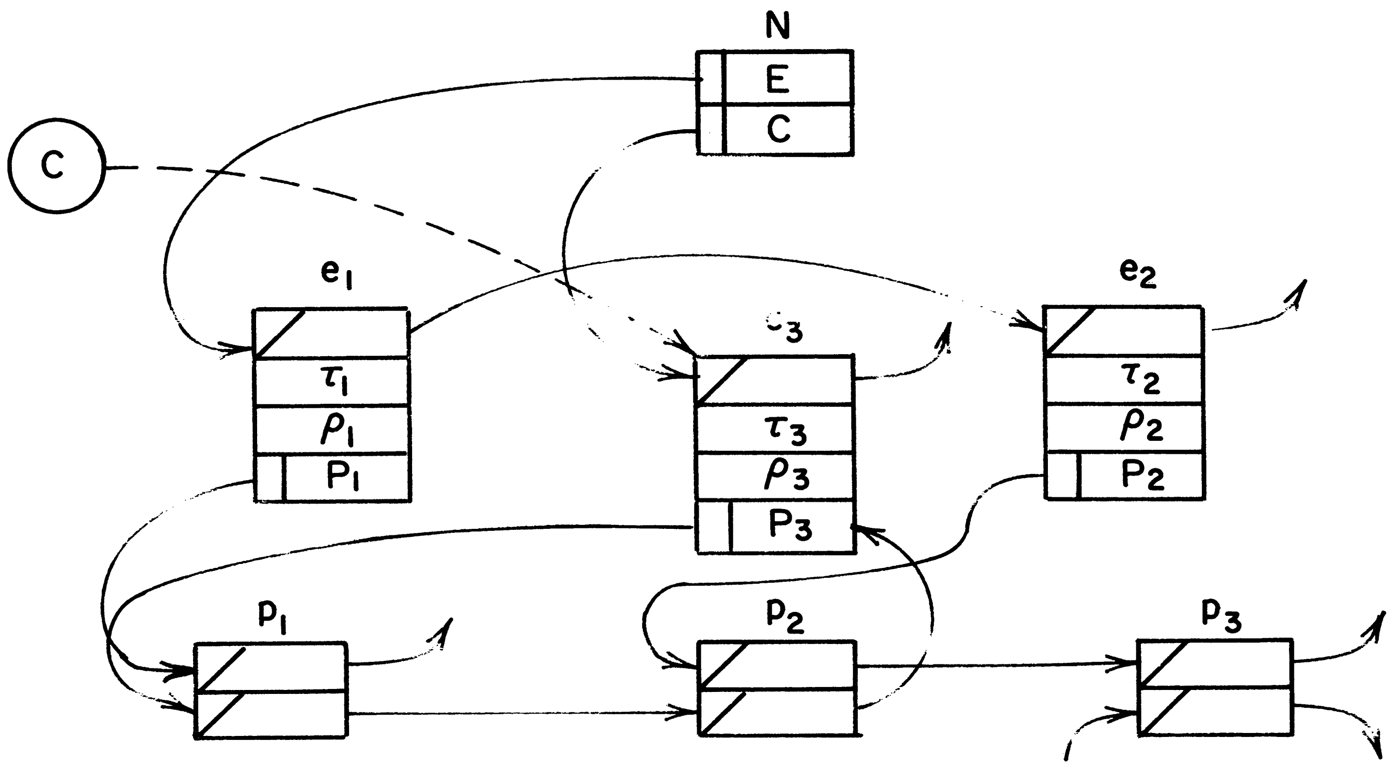


Figure 5. 4

Fragment of working structure upon entry to "collect associates" routine.

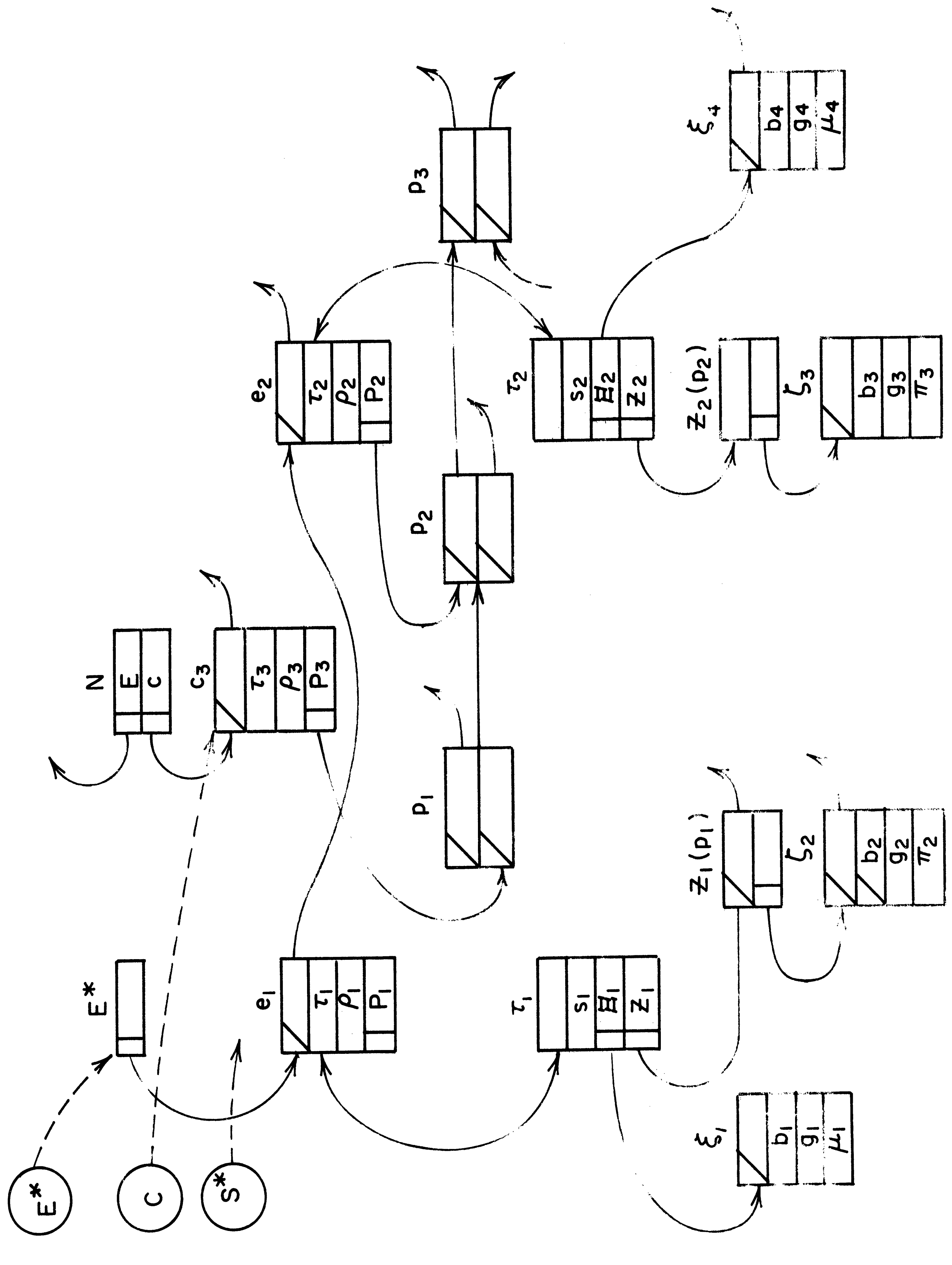


Figure 5.5. Status of working structure at β in Figure 5.2.

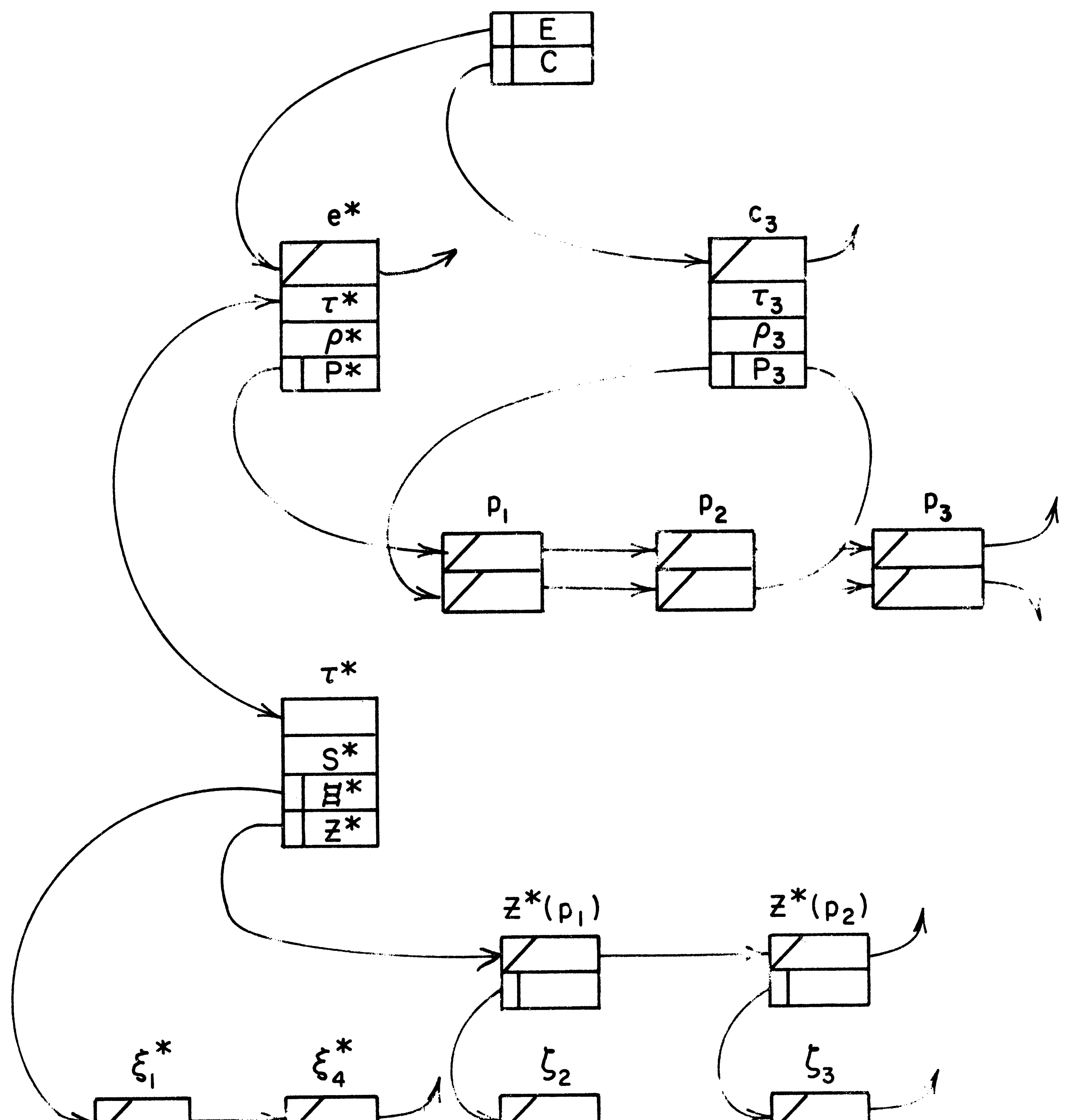


Figure 5. 6 Result of collection operation on figure 5. 3.

block of E* and letting it be the element block which will become the collected element e*. It is removed from E* and put back in E. The S* which was calculated in part two replaces the state set of this element. The parameter list of this element is replaced by an empty list (the type must henceforth be literal). From here on, following programming convention, this element will be referred to as e', until the end of the operation, when it becomes the e* of equation (5.2). The b_ℓ and g_ℓ of the exoevents and autoevents of e* are extended to the new state space, and replaced. That is, b_ℓ is replaced by $S^* \times_{i_\alpha} b_\ell$ and g_ℓ by $S^* \odot_{i_\alpha} g_\ell$ for each $\xi_\ell \in \Xi^*$ and each $\zeta_\ell \in Z^*$. This latter operation is done by a separate subroutine.

At this point ("α" in Figure 5.3) each remaining element in E* is taken, in turn, its events similarly extended to S*, and its ports and events added appropriately to the ports and events of e'. If P, Ξ, Z are the port set, autoevent set and exoevent function of the current element, then, since they are each represented by rings, they are added to P', Ξ', Z' by insertion

$$P' \longleftarrow P' \cup P \tag{5.18}$$

$$\Xi' \longleftarrow \Xi' \cup \Xi \tag{5.19}$$

$$Z' \longleftarrow Z' \cup Z \tag{5.20}$$

where the arrow represents "replaced by" and the union symbol here means "insert the ring members of the second operand between the head and ring members of the first operand." The head of the second operand is destroyed in the operation, which was called "ring concatenation" in Table 3.3.

The major subprograms of the collect routine are summarized in Table 5.2.

5.2 The Meaning of Connections

The exoevents at the ports of an element represent the conditions under which tasks can be removed from (or inserted into) the element via the port, and the consequence to the element's state of that removal (or insertion). The function of a connection is to indicate a relationship between the exoevents of a set of ports. It indicates the conditions under which a task is to be transferred between a pair of ports. For example, when a pair of ports is joined by a simple connection, a transfer of a task occurs immediately whenever the "output" port has a task which can be removed and the "input" port simultaneously can accept a task. The exoevents at the two ports are thought to "occur" spontaneously whenever these conditions are met. For each <u>type</u> of connection the rules determining the occurrence of the exoevents will differ, and the meaning of connections is conveyed by these rules.

Since we will be operating only on collected elements, the only time we are concerned with the meaning of a connection will be when it is a connection between ports belonging to a single element. The definitions that follow will assume that condition.

Program Name	Input Arguments	Operations Performed	Output Parameters	Error Conditions
Collect Associ-ates	Pointer to Connection	See Table 5.1	None	None
<u>Major Subprograms</u>				
Find Associ-ates	Pointer to Connection	Removes from E every associate of the connec-tion, creates new set E* containing these associates.	Pointer to set of as-sociates E*.	
Form State Set	Point to a set of elements E*	Calls type finder for all elements in E*, and forms the pro-duct of their state sets, in the order of appearance in the ring E*, forms D. Also forms name stack in first of E*.	Pointer to a state set S*. Pointer di-men-infor. block D.	(Type find-er detects missing parameter values.)
Extend Events	Pointer to a type block, τ. Pointer to a state set, S. Index of first di-mension, m.	Replaces b_ℓ by $S \times_m b_\ell$ for every $\xi_\ell \epsilon \Xi$ and every $\zeta_\ell \epsilon Z$. Re-places g_ℓ by $S \odot_m g_\ell$ similarly.	None	m too large.

Table 5.2 Summary — Associate Collection Routine

Program Name	Input Arguments	Operations Performed	Input Parameters	Error Conditions
<u>Service Routines Used</u>				
n-tuple set operators: product, intersection; also				
Restrict	Pointer to an n-tuple set b, Pointer to a state set S, Index of first dimension m.	Replaces b by $S \times_m b$.	None	m too large
n-tuple operators:				
Extend	Pointer to an n-tuple g, Pointer to state set S, Index of first dimension m.	Replaces g by $S \odot_m g$.	None	m too large
Ring operators: all.				

Table 5.2 Summary--Associate Collection Routine (continued)

The information describing the effect of such a connection upon its associated element will be described by a set of objects called spontaneous events. These events describe the conditions for, and immediate consequences of, transfer of tasks through the connection. These events can be thought to occur "spontaneously" whenever the state of the element becomes one of a set of forbidden states. For example, in the simple connection, a forbidden state would be one in which one port was "offering" a task and the other could "accept" one. The consequence would immediately be a further change in state.

Let the <u>spontaneous event set</u> Θ_k of a connection c_k joining ports of the element e_i be

$$\Theta_k = \{\theta_h : h \in H_k\}. \tag{5.21}$$

where

$$\theta_h = < b_h, g_h, \pi_h > \tag{5.22}$$

b_h is a subset of S_i

g_h is a constant n-tuple such that

for each $x \in b_h$, $x + g_h$ is in S_i

π_h is a probability

H_k is an index set.

The set b_h is the <u>forbidden state set,</u> g_h is the <u>increment</u> of the event, and π_h is the probability given that the state is in b_h that it will jump by an amount g_h. As for exoevent sets, the probabilities in a spontaneous event set must satisfy the restriction that, for each $x \in S_i$

$$\sum_{\substack{h: \\ x \in b_h}} \pi_h = 0 \text{ or } 1. \tag{5.23}$$

The spontaneous events are formed from the information conveyed in the exoevent sets of the connected ports in a manner which is distinct to the "type" of the connection. Let a connection c_k be of <u>simple type</u>. Let the ports joined by c_k be $P_k = \{p_{j_1}, p_{j_2}\}$ and let those both be ports of the element e_i, so that $P_k \subseteq P_i$. Recall that the exoevent sets of p_{j_1} and p_{j_2} are given by

$$Z_i(p_{j_1}) = \{\zeta_{m_1} : m_1 \in M_i(p_{j_1})\} \tag{5.24}$$

$$Z_i(p_{j_2}) = \{\zeta_{m_2} : m_2 \in M_i(p_{j_2})\} \tag{5.25}$$

$$\zeta_{m_1} = < b_{m_1}, g_{m_1}, \pi_{m_1} > \tag{5.26}$$

$$\zeta_{m_2} = < b_{m_2}, g_{m_2}, \pi_{m_2} > . \tag{5.27}$$

The spontaneous event set Θ_k for this simple connection is given by

$$\Theta_k = \{\theta_h : h \in M_i(p_{j_1}) \times M_i(p_{j_2})\} \tag{5.28}$$

where

$$\theta_{m_1 m_2} = < b_{m_1} \cap b_{m_2}, \; g_{m_1} + g_{m_2}, \; \pi_{m_1} \pi_{m_2} > . \tag{5.29}$$

This indicates that spontaneous events occur whenever the state is in both endocondition sets, and that the increment is the sum of that due to removing a task from one port and that due to inserting a task into the other port. The set of states for which no spontaneous event occurs is

$$\bar{b}_k = S_i - \bigcup_{m_1, m_2} (b_{m_1} \cap b_{m_2}). \tag{5.30}$$

For convenience in the programming, we will find an augmented spontaneous event set $\bar{\Theta}_k$ to be useful. It is defined, for the simple connection, as

$$\bar{\Theta}_k = \Theta_k \cup \{< \bar{b}_k, 0, 1 >\}, \tag{5.31}$$

where the additional event represents no change for states which are not forbidden. In this case, the probabilities add to 1 for <u>every</u> member of S_i. It is readily shown that eq. 5.23 is satisfied for the spontaneous event set Θ_k.

The other connection types will be discussed below. In what follows the element whose ports are being connected will be called e_i, and the exoevent set of any one of its ports p_{j_α} will be assumed to be written

$$Z_i(p_{j_\alpha}) = \{\zeta_{m_\alpha} : m_\alpha \in M_i(p_j)\} \tag{5.32}$$

with

$$\zeta_{m_\alpha} = < b_{m_\alpha}, g_{m_\alpha}, \pi_{m_\alpha} >. \tag{5.33}$$

The probability summation property will hold for all spontaneous event sets defined.

Considering a connection c_k of overflow type, let the output port joined by c_k be p_{j_0}, the preferred input port be p_{j_1}, and the overflow input be p_{j_2}, so that $P_k = \{p_{j_0}, p_{j_1}, p_{j_2}\} \subseteq P_i$. The spontaneous events consist of two classes: those which represent a preferred task flow, and those which represent an overflow. The first class occurs whenever the state is in both b_{m_0} and b_{m_1}, for each m_0 and m_1 in $M(p_{j_0})$ and $M(p_{j_1})$ respectively, since that is the only condition where the preferred flow is possible. The second class occurs whenever the state is in both b_{m_0} and b_{m_2}, but not in any of the b_{m_1}, $m_1 \in M(p_{j_1})$ (i. e., preferred flow is not possible), for each m_0 and m_2 in $M(p_{j_0})$ and $M(p_{j_1})$. Algebraically,

$$\Theta_k = \{\theta_h : h \in M(p_{j_0}) \times M(p_{j_1})\} \cup \{\theta_h : h \in M(p_{j_0}) \times M(p_{j_2})\} \tag{5.34}$$

$$\theta_{m_0 m_1} = < b_{m_0} \cap b_{m_1},\ g_{m_0} + g_{m_1},\ \pi_{m_0} \pi_{m_1} > \tag{5.35}$$

$$\theta_{m_0 m_2} = < b_{m_0} \cap b_{m_2} - (\bigcup_{m_1 \in M(p_{j_1})} b_{m_1}),\ g_{m_0} + g_{m_2},\ \pi_{m_0} \pi_{m_2} > \tag{5.36}$$

for $m_0 \in M(p_{j_0})$, $m_1 \in M(p_{j_1})$, $m_2 \in M(p_{j_2})$. The no-event state set is

$$\bar{b}_k = S_i - \bigcup_{h \in H} b_h, \tag{5.37}$$

where $H = (M(p_{j_0}) \times M(p_{j_1})) \cup (M(p_{j_0}) \times M(p_{j_2}))$, and the augmented spontaneous event set

$$\bar{\Theta}_k = \Theta_k \cup \{< \bar{b}_k, 0, 1 >\}. \tag{5.38}$$

The merge type connection, joining an input port p_0 of e_i, a (preferred) output port p_1, and (secondary) output port p_2 (one must, in defining any merging-type elements resolve the competition among inputs, and that is done here on a purely priority basis) has precisely the same spontaneous events as the overflow-type element. Thus they are indistinguishable, and will henceforth be given the same type-name, the "overflow-merge."

The random branch is defined, for the purposes of this report, as a branch for which flow can occur only if a task can be accepted at any connected input port of the element, and if a task is also available from the connected output port. Let the output port of the element be p_0 and the input ports be $p_1, \ldots, p_n$, where N is the number of branches in the connection. When this condition is met a task is transferred from p_0 to one of the ports, p_{j_α} say, with probability γ_α, $\alpha = 1, 2, \ldots, N$, where $(N, \gamma_1, \ldots, \gamma_N)$ is the parameter list of the connection. Algebraically,

$$\Theta_k = \{\theta_h : h \in M(j_0) \times \ldots \times M(j_N) \times \{1, \ldots, N\}\} \tag{5.39}$$

$$\theta_{m_0 \ldots m_N \alpha} = < b_{m_0} \cap \ldots \cap b_{m_N}, \; g_{m_0} + g_{m_\alpha}, \; \pi_{m_0} \ldots \pi_{m_N} \gamma_\alpha > \tag{5.40}$$

for all $m_\alpha \in M(p_{j_\alpha})$, $\alpha = 1, \ldots, N$. As usual

$$\bar{b}_k = S_i - \bigcup_h b_h \tag{5.41}$$

and

$$\bar{\Theta}_k = \Theta_k \cup \{< \bar{b}_k, 0, 1 >\}. \qquad (5.42)$$

Provided that the property

$$\sum_{\alpha=1}^{N} \gamma_\alpha = 1 \qquad (5.43)$$

holds, the probability summation property (eq. 5.23) will also hold for Θ_k.

5.3 The Absorption of Connections

The operation of connection absorption removes the ports connected from the associated element, prunes the exoevent sets at the removed ports from the exoevent function, forms the spontaneous event set, and then consolidates the autoevents, remaining exoevents, and spontaneous events, to produce a new element. The operation of this subroutine of the compiler is illustrated schematically in Figure 5.7, and its major parts are summarized in Table 5.3.

The first operation in the program determines the element joined, and alters the structure in preparation for the formation of the spontaneous events. The ports joined by the connection are removed, and appropriate alteration of the exoevent function is carried out, with the exoevent sets of the removed ports shifted to the connection structure. This operation is clearly illustrated in Figures 5.8 and 5.9, which show fragments of the network structure before and after the first block of Figure 5.7 is executed. In the course of finding the element associated

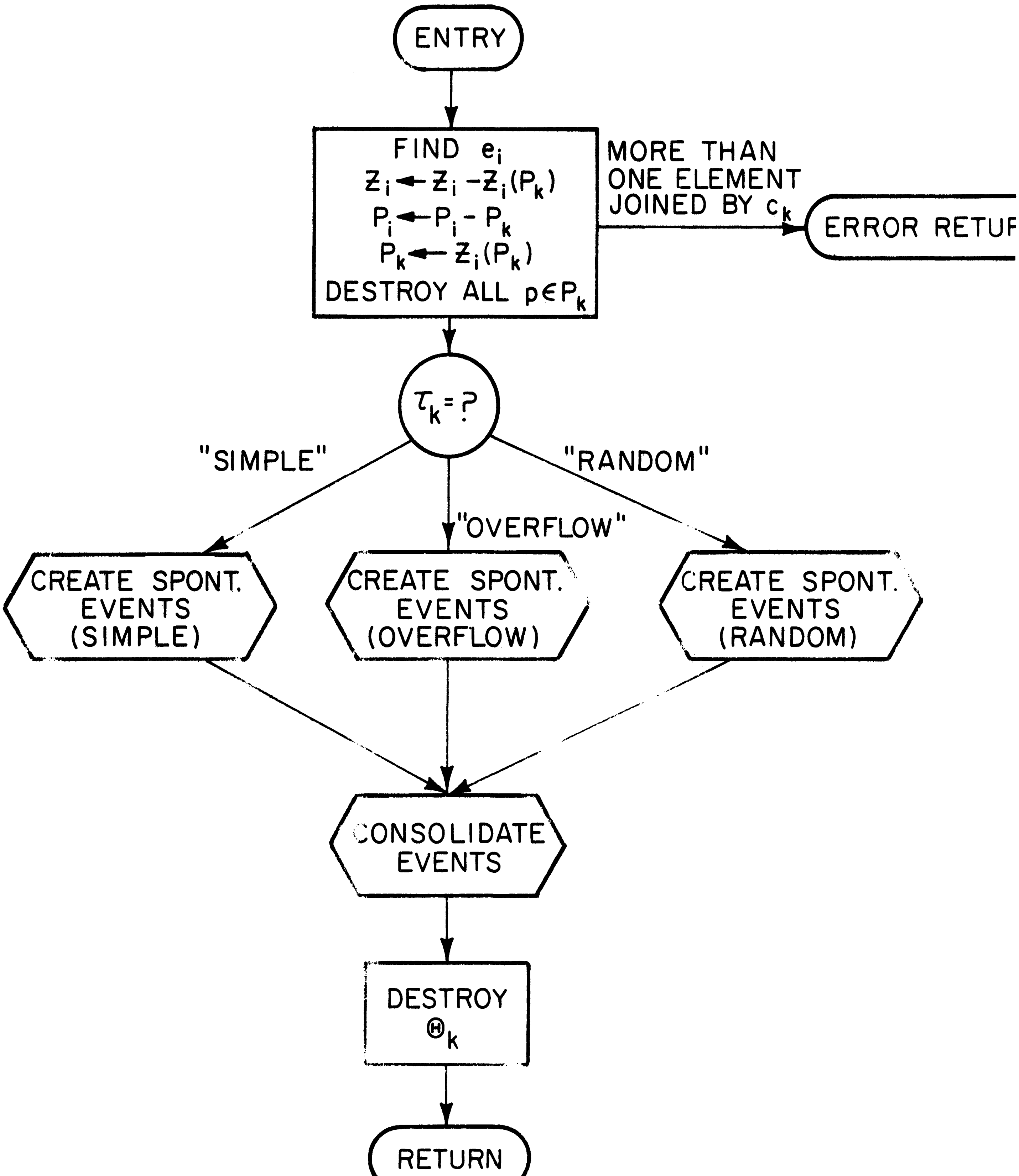

ENTRY
FIND e_i
$Z_i \leftarrow Z_i - Z_i(P_k)$
$P_i \leftarrow P_i - P_k$
$P_k \leftarrow Z_i(P_k)$
DESTROY ALL $p \in P_k$
MORE THAN ONE ELEMENT JOINED BY c_k
ERROR RETUF
$\tau_k = ?$
"SIMPLE"
"OVERFLOW"
"RANDOM"
CREATE SPONT. EVENTS (SIMPLE)
CREATE SPONT. EVENTS (OVERFLOW)
CREATE SPONT. EVENTS (RANDOM)
CONSOLIDATE EVENTS
DESTROY Θ_k
RETURN

Figure 5. 7

Flow diagram for absorption of connections.

Program Name	Input Arguments	Operations Performed	Output Parameters	Error Conditions
Absorb Connection	Pointer to connection	Replaces element and connection by a single element	None	Connection joins more than one element.
<u>Major Subprograms</u>				
Spontaneous Event Routines: Simple, priority, random	Pointer to connection (with exoevents) Pointer to state set	Combines exoevents to produce spontaneous events in a contiguous block of storage. Prunes state set. (One routine per connection type.) - Uses parameter list.	Pointer to spontaneous event set θ. Size of block containing θ structure.	Missing parameters in connection. Wrong number of points. Parameter restriction violated.
Consolidate Events	Pointer to spontaneous event set. Pointer to element.	Replaces all autoevents and exoevents of the element by new events reflecting the influence of the spontaneous events.	None	None

Table 5.3 Summary—Absorb Connection Routine

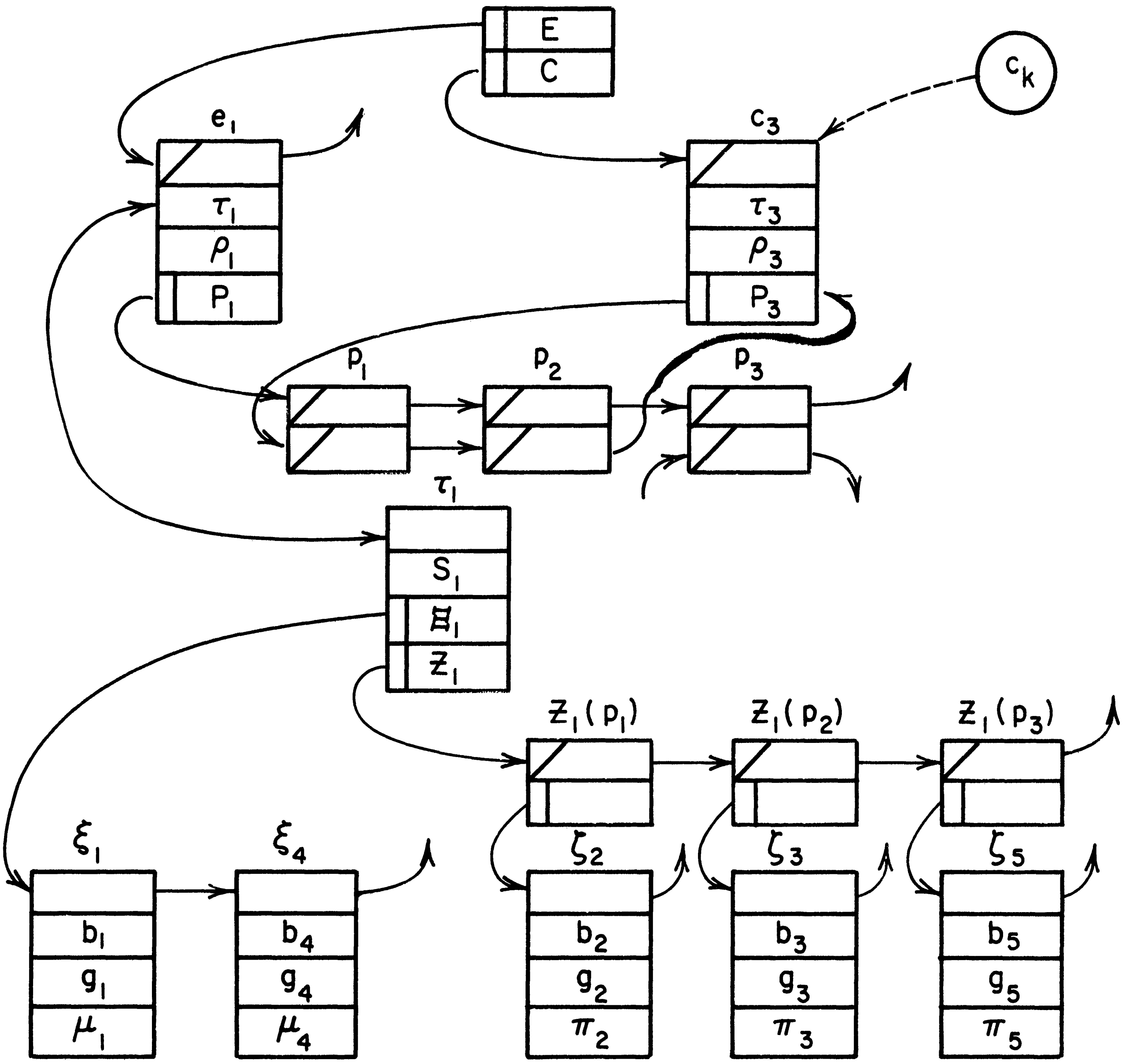


Figure 5. 8

Fragment of working network structure at beginning of connection absorption.

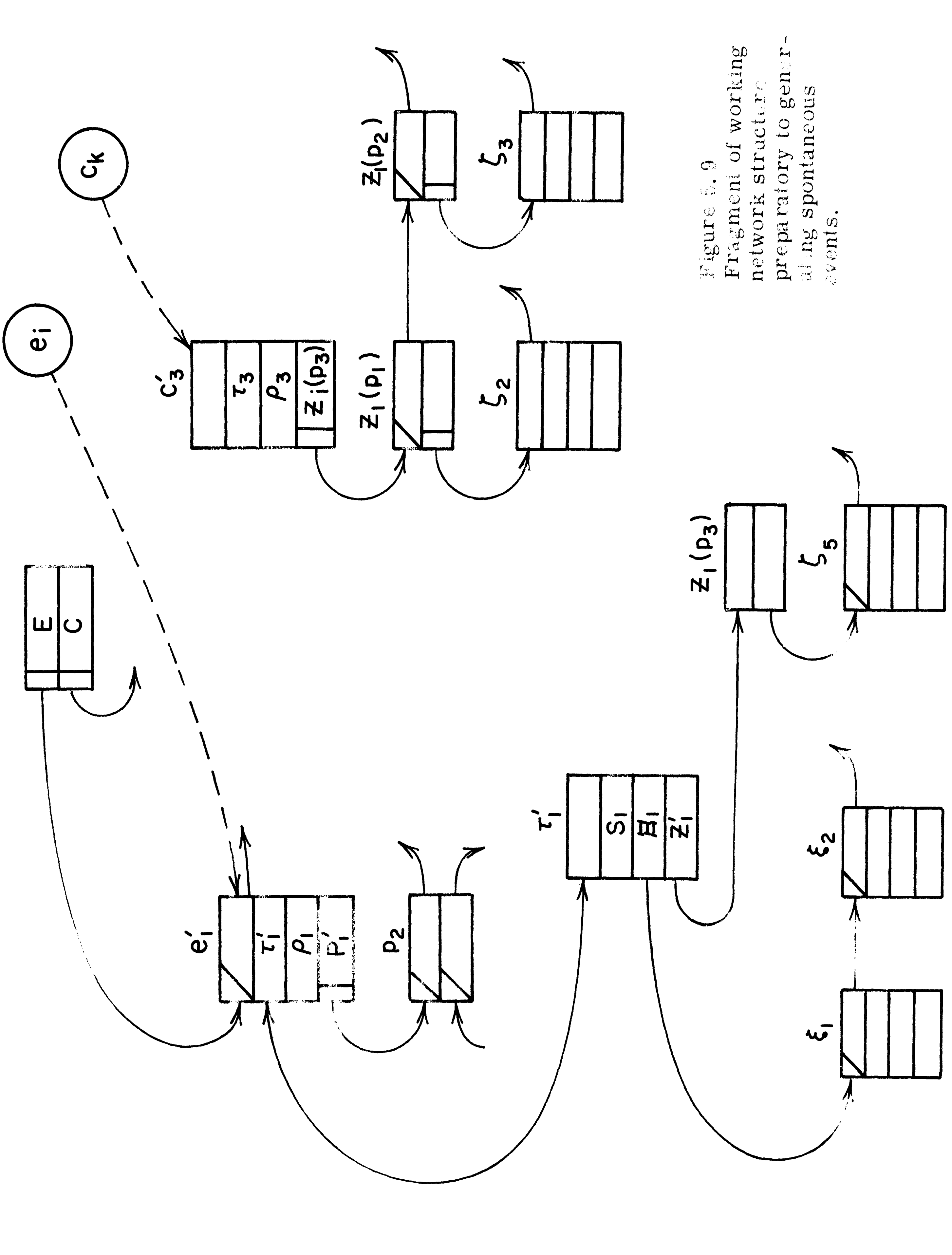


Figure 5.9
Fragment of working network structure preparatory to generating spontaneous events.

with the connection, this block will also discover if, by failure of the element collection program, the number of elements connected is not unity. This is a fatal error.

At the conclusion of these operations, an appropriate Spontaneous Event Routine is called, depending on the connection type. Since the spontaneous events will not be altered during the period of their existence, and since the operation of "getting" and "freeing" small blocks is relatively time-consuming, the Spontaneous Event Routines should create the entire Θ-structure so that it is physically contiguous. If this is done, only one fairly large block need be gotten from the free core manager, and it will be kept until the connection absorption is complete, at which time the entire structure can be destroyed by a single "free" command to the manager (see "Destroy Θ_k" in Figure 5.7).

The spontaneous event routine (a prototype of which is shown in Figure 5.10) constructs a list structure which corresponds to the set described in the previous section in its requested core space. The events will be generated one by one and added to a ring, while the state set S_i and no-event set $\bar{b}_k$ will be pruned with the formation of each event. Thus, at the conclusion of the formation of the last spontaneous event, the state set has had all forbidden states removed from it, and the augmented spontaneous event set $\bar{\Theta}_k$ has been formed. Whenever a forbidden state set b_h is found to be empty, the corresponding spontaneous event θ_h will be simply omitted. In this routine, also, the parameters

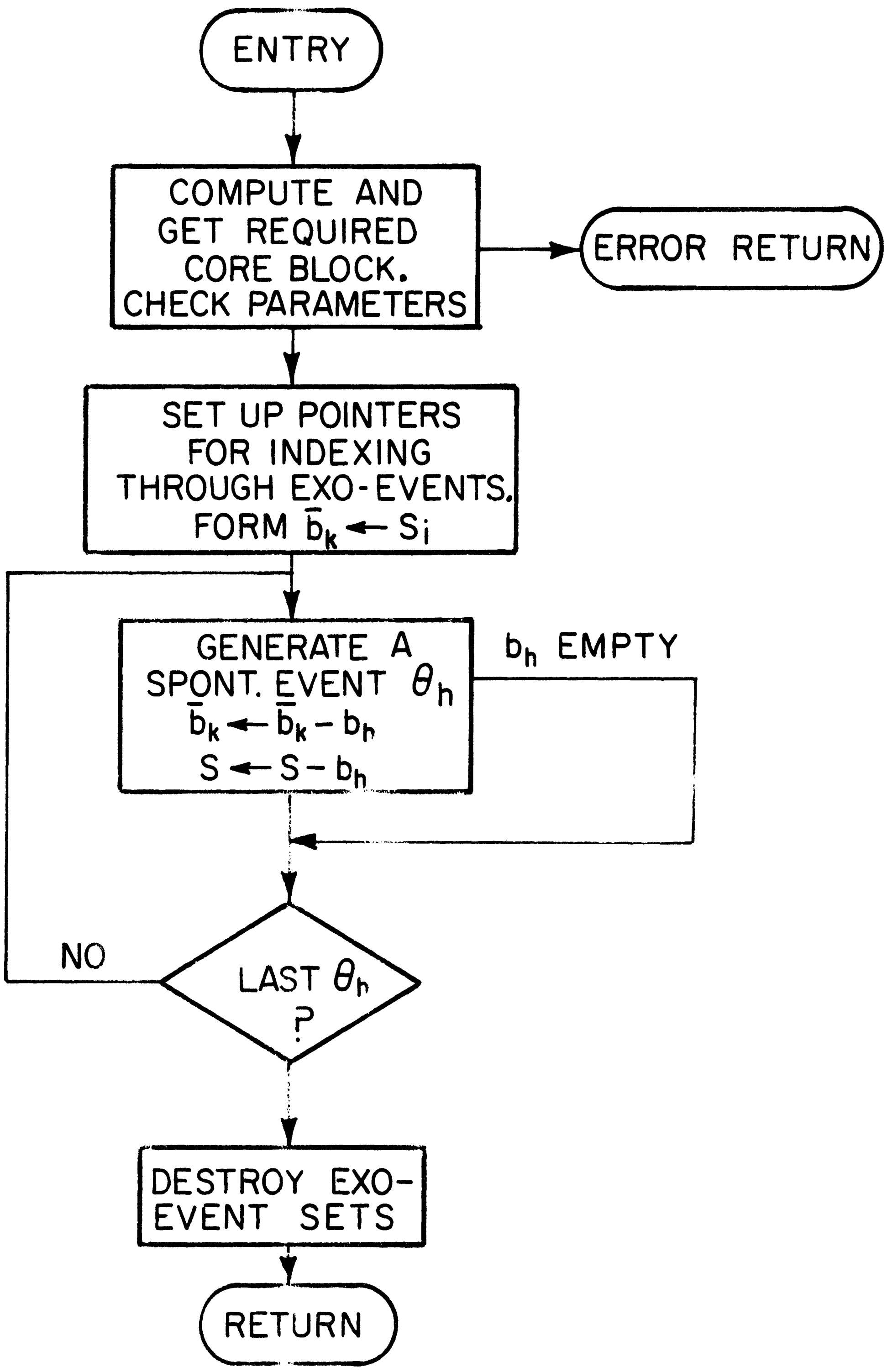

ENTRY
COMPUTE AND GET REQUIRED CORE BLOCK. CHECK PARAMETERS
ERROR RETURN
SET UP POINTERS FOR INDEXING THROUGH EXO-EVENTS. FORM $\bar{b}_k \leftarrow S_i$
GENERATE A SPONT. EVENT θ_h $\bar{b}_k \leftarrow \bar{b}_k - b_h$ $S \leftarrow S - b_h$
b_h EMPTY
NO
LAST θ_h ?
DESTROY EXO-EVENT SETS
RETURN

Figure 5.10

Prototype flow diagram for spontaneous event routines.

are checked, if this is a random branch, to see that the probabilities (γ_α in section 5.2) sum to unity. The exoevent sets for the connected ports are being destroyed after the spontaneous events have been formed , and the structure finally appears as illustrated in Figure 5.11.

5.4 Consolidation of Events

The operation of event consolidation replaces each event (either exoevent or autoevent) in the element e_i associated with the connection by a new event, or events, which includes the effect of the spontaneous events. The action to be accounted for in this operation is that produced when an event of the element causes a spontaneous event to occur.

Such is the case, for example, when a task completion in a server causes a transfer of the completed task through the connection joining its output port. The effect of the autoevent is to change the state. If the state is thereby changed to a forbidden state, the spontaneous event (task transfer) occurs, changing the state again. In this report, we assume that only one task may be required to pass through a connection upon occurrence of any event, so that the second state change cannot itself result in another spontaneous event at the same connection.

A similar argument applies to exoevents, as, for example, the consequences of a task being entered into the input of a queue. This will result in an increase in state which may cause a transfer of a task through the connection joining its output. That is, the state is changed

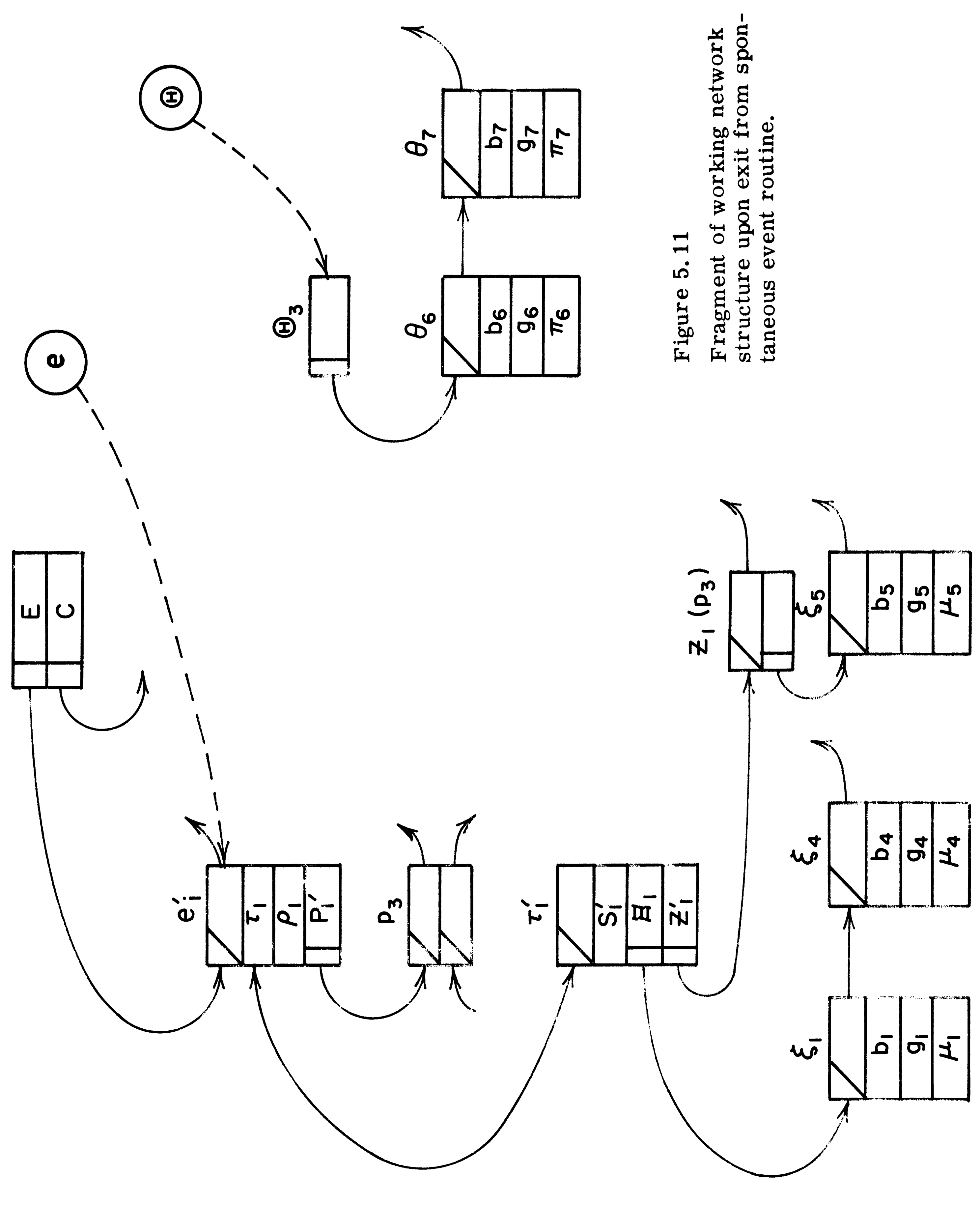


Figure 5.11
Fragment of working network structure upon exit from spontaneous event routine.

to a forbidden state and a spontaneous event causes a still further change in state.

The new event set is readily found from the old and the spontaneous events. Let $\bar{\Theta}_k$ be the augmented spontaneous event set

$$\bar{\Theta}_k = \{\theta_h : h \in \bar{H}_k\}. \tag{5.44}$$

$$\theta_h = < b_h, g_h, \pi_h > \tag{5.45}$$

and let Ξ_i be the auto-event set of the associated set e_i

$$\Xi_i = \{\xi_\ell : \ell \in L_i\} \tag{5.46}$$

$$\xi_\ell = < b_\ell, g_\ell, \mu_\ell > \tag{5.47}$$

Then the new auto-event set $\Xi_i{}^*$ of the associated element e_i will be

$$\Xi_i{}^* = \{\xi^*_{\ell *} : \ell * \in L_i \times \bar{H}_k\}. \tag{5.48}$$

These events are

$$\xi^*_{\ell h} = < b_\ell \cap b_h \cap \eta_{g_\ell}(b_h), \; g_\ell + g_h, \; \mu_\ell \, \ell_h > \tag{5.49}$$

for all $\ell \in L_i$, $h \in \bar{H}_k$, where, by definition $\eta_g(b)$ represents the set found by subtracting the vector g from every member of the set b. The set $b_\ell \cap b_h \cap \eta_{g_\ell}(b_h)$ represents all non-forbidden states in b_ℓ which, upon occurrence of the event ξ_ℓ, are carried into a (forbidden) state in b_h. It is these states only which are affected by the increment g_h.

Similarly let p_j be any port in P_i, and the exo-event set

$$Z_i(p_j) = \{\zeta_{m_i} : m_j \in M_i(p_j)\}, \tag{5.50}$$

$$\zeta_{m_j} = < b_{m_j}, g_{m_j}, \pi_{m_j} > , \qquad (5.51)$$

becomes

$$Z_i^*(p_j) = \{\zeta^*_{m^*_j} : m_j^* \in M_i(p_j) \times \bar{H}_k\}$$

where

$$\zeta^*_{m_j h} = < b_{m_j} \cap \eta_{g_{m_j}}(b_h), g_{m_j} + g_h, \pi_{m_j} \pi_h > . \qquad (5.52)$$

(The probability property, eq. 4.18, for exo-event sets is readily shown to be preserved.)

Many of the sets $b_\ell \cap \eta_{g_\ell}(b_h)$ and $b_{m_j} \cap \eta_{g_{m_j}}(b_h)$ may be found to be empty. In such a case, the event conveys no useful information and may be omitted from the corresponding set of events.

The program which will accomplish this transformation is the event consolidator, described by the flow diagram of Figure 5.12. This program calls on the "event set consolidator" which is described by the diagram of Figure 5.13. The symbol Φ is used in this diagram to represent an arbitrary event set

$$\Phi = \{\phi_d : d \in D\} \qquad (5.53)$$

$$\phi_d = \{b_d, g_d, \nu_d\} . \qquad (5.54)$$

This program proceeds around the Φ ring, inserting the new events behind it as it proceeds. Each time a ϕ_d has been treated completely, it is destroyed and the program proceeds to the next ϕ_d until it comes

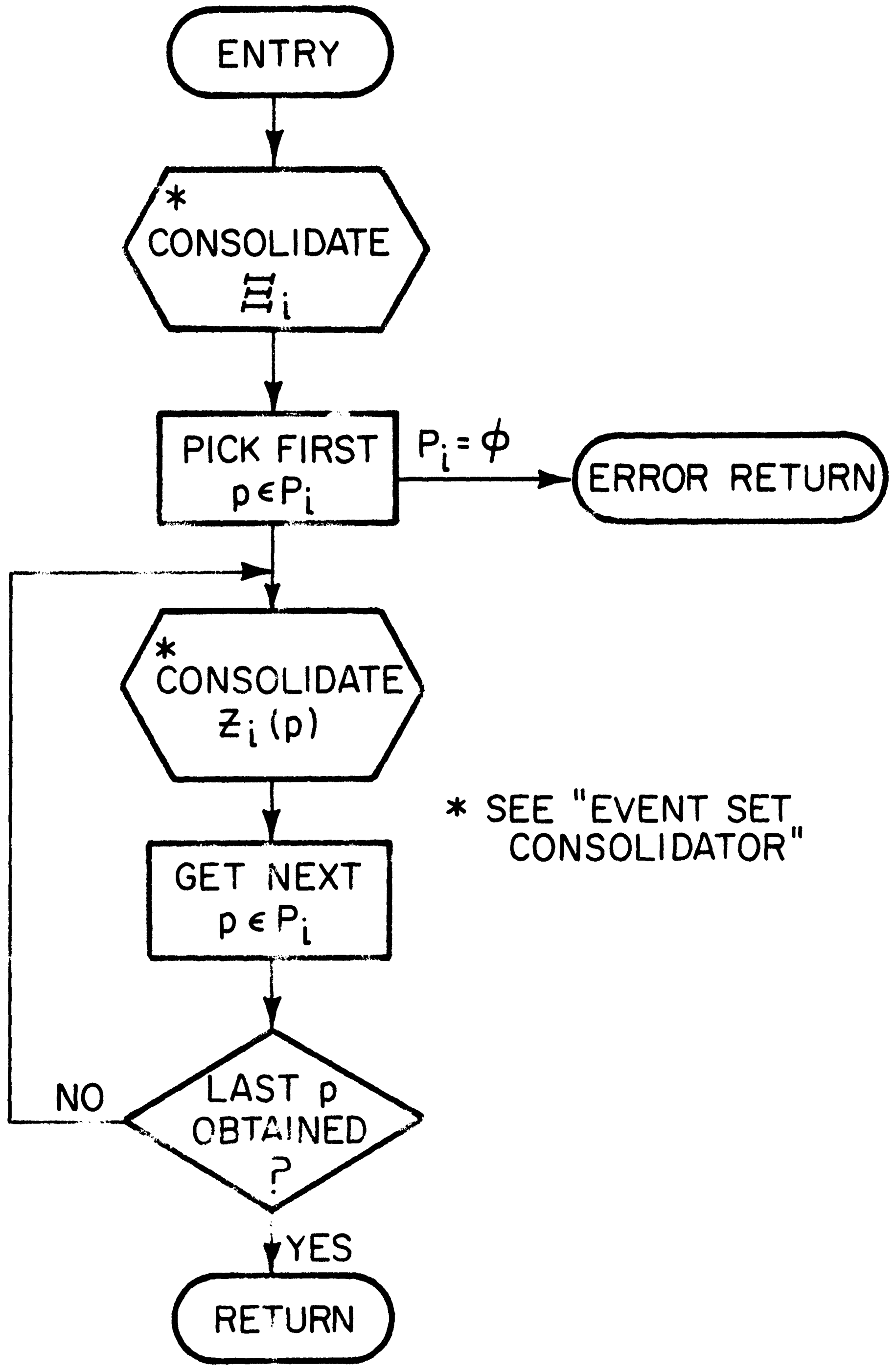


Figure 5.12

Flow diagram: Event consolidator.

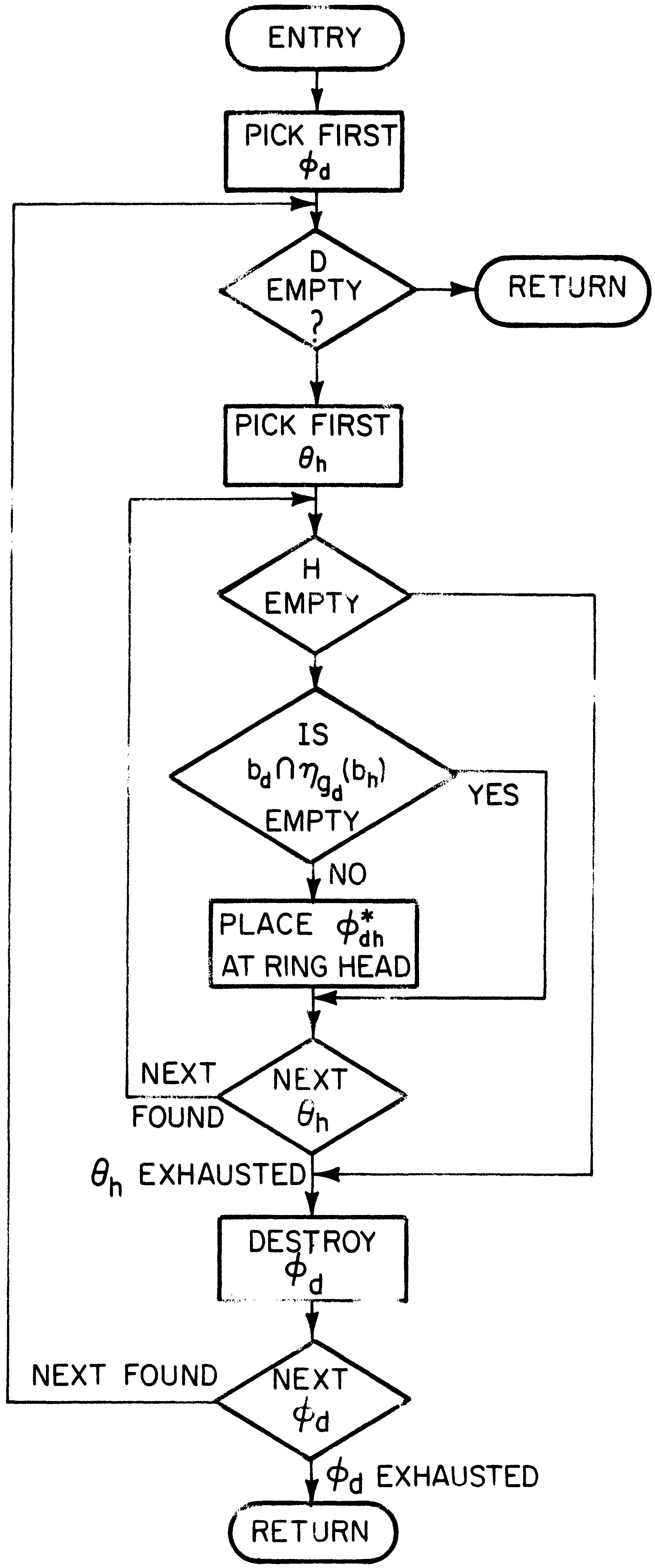


Figure 5. 13 Flow diagram: Event set consolidator.

back to the head of the ring. If either set is empty, the result is an empty Φ^*.

5.5 Trimming of the State Set

The consolidation procedure described above can, under certain circumstances, produce surprisingly large state sets. This will be particularly troublesome when a random branch was involved in the compilation at some stage. This is an manifestation of a general problem facing this translation which cannot be economically solved: the problem of transient states and reducible Markov chain models.

The points in the rectangular "state set" S_R of the (finite) Markov chain of the entire network fall basically into three categories

1) Forbidden states
2) Recurrent states
3) Transient states.

The forbidden states are points of S_R which result immediately in spontaneous events, which, in turn, take the process to a point which is not a forbidden state. The forbidden states are automatically pruned from the state set upon each consolidation, and hence are no problem.

The present difficulty revolves around the presence of transient states in the model. Transient states are legitimate states of the Markov model. However, since they have an equilibrium probability of zero, and since information bearing upon them ordinarily has no bearing on the calculation of equilibrium probabilities of other states, they are "excess baggage" and will waste valuable storage in the various data structures of QAS.

The ideal procedure would be to identify all transient states of consolidated elements as they are created, and to remove them from the state set. Unfortunately, there appears to be no systematic procedure for identifying all of the transient states without performing a calculation very similar to evaluation of all equilibrium probabilities, which defeats our purpose. (We want the state set reduced <u>before</u> we calculate equilibrium probabilities.)

The example of the random branch is a good one. Let a diagram contain a fragment like that of Figure 5.14. The states for which more

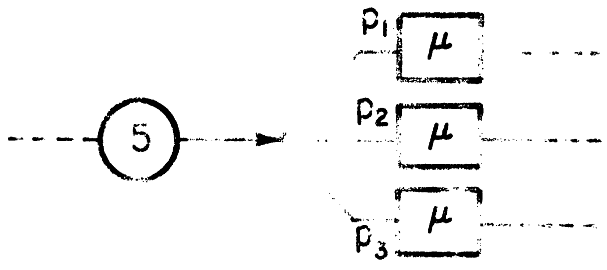


Figure 5.14

A network fragment containing a random branch.

than one server is busy cannot be entered from any state for which less than two servers are busy, since the random branch is blocked when one server is busy, and no "tasks" will be transferred. However, if the system starts with more than one server busy it will, upon exit of tasks from service, enter states with less than two servers busy. Thus, there is a fairly large number of transient states (37 recurrent states, 120 transient states, 4 forbidden states). As more elements are consolidated with this fragment, the ratio of transient to recurrent states will remain much the same, at best decreasing to 2:1. The states of the example were easy to identify through physical insight, a weapon not available to the compiler.

A general solution to the problem of eliminating a transient states is not available. A more specific "fix" must be one which is guaranteed valid for every network and every definable element and connection. In other words, it should not unnecessarily restrict the already restricted class of models which can be treated by QAS. Especially, it should not result in a restriction on inputs to the compiler which cannot be enforced syntactically before compilation is attempted.

Fortunately, there is a class of transient states which is relatively easily recognized and which includes those of the example. The recommended solution will eliminate this particular kind of transient state, and result in perfect pruning of states for the random branch. Figure 5.15 illustrates various cases of communication among transient (shown by circles) and recurrent states (shown by x marks) for a total of three states.

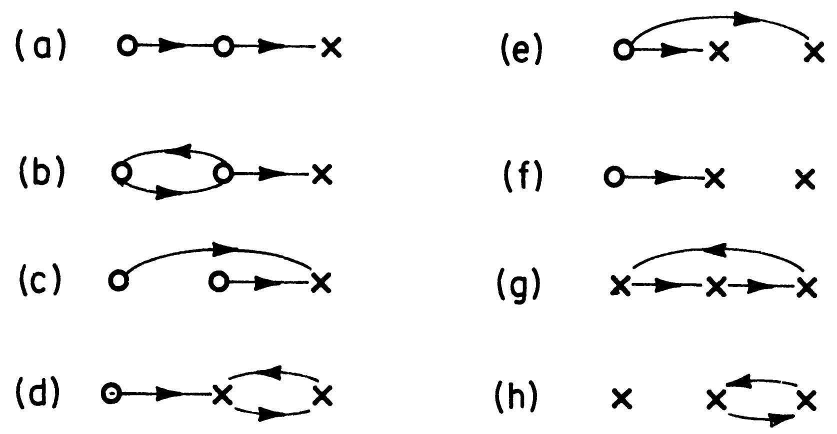


Figure 5.15

Communication among states (recurrent states shown by x, transient states by 0.)

The procedure recommended is that, subsequent to the absorption of each connection, the states which are not "target states" of any event be successively trimmed from the state set until no more such states can be found. Since case (a) of Figure 5.15 is representative of what happens in the random branch example, the need for successive trimming should be clear. The target states of an event ϕ_d (either auto- or exo-) are found by the expression

$$\eta_{-g_d}(b_d)$$

using the "back projection" operator η previously defined.

Figures 5.16 and 5.17 illustrate the programs required, and show how the removal of previously located transient states can be efficiently accomplished while new transient states are being sought. The set T_1 represents a set of known transient states, while the set T_2 is a set of states formed by starting with the state set and repeatedly deleting target state of events as they are sequenced through. Once T_2 is found to be empty, we know that no further transient states will be found by this method. However, the sequence must be continued if T_1 is not empty, so that all previously located transient states are removed from all events. Figure 5.17 is not particularly efficient as shown, since calculations of set-differences, and evaluations of η are performed even when some components (T_1 or T_2) are known to be empty. Suitable switch setting or branching can make this program highly efficient since the most probable situation is expected to be the case with T_1 empty, and since T_2 is expected to be frequently empty after only a few events are examined. Figure 5.18 shows one possible configuration which will take advantage of such information.

5.6 The Result of Collection, Absorption and Trimming

After repeated collection of associates and absorption of connections, the working network structure will eventually lack remaining connections. At that point, the network structure will ordinarily look like Figure 5.19, with a single element having no ports or exoevents,

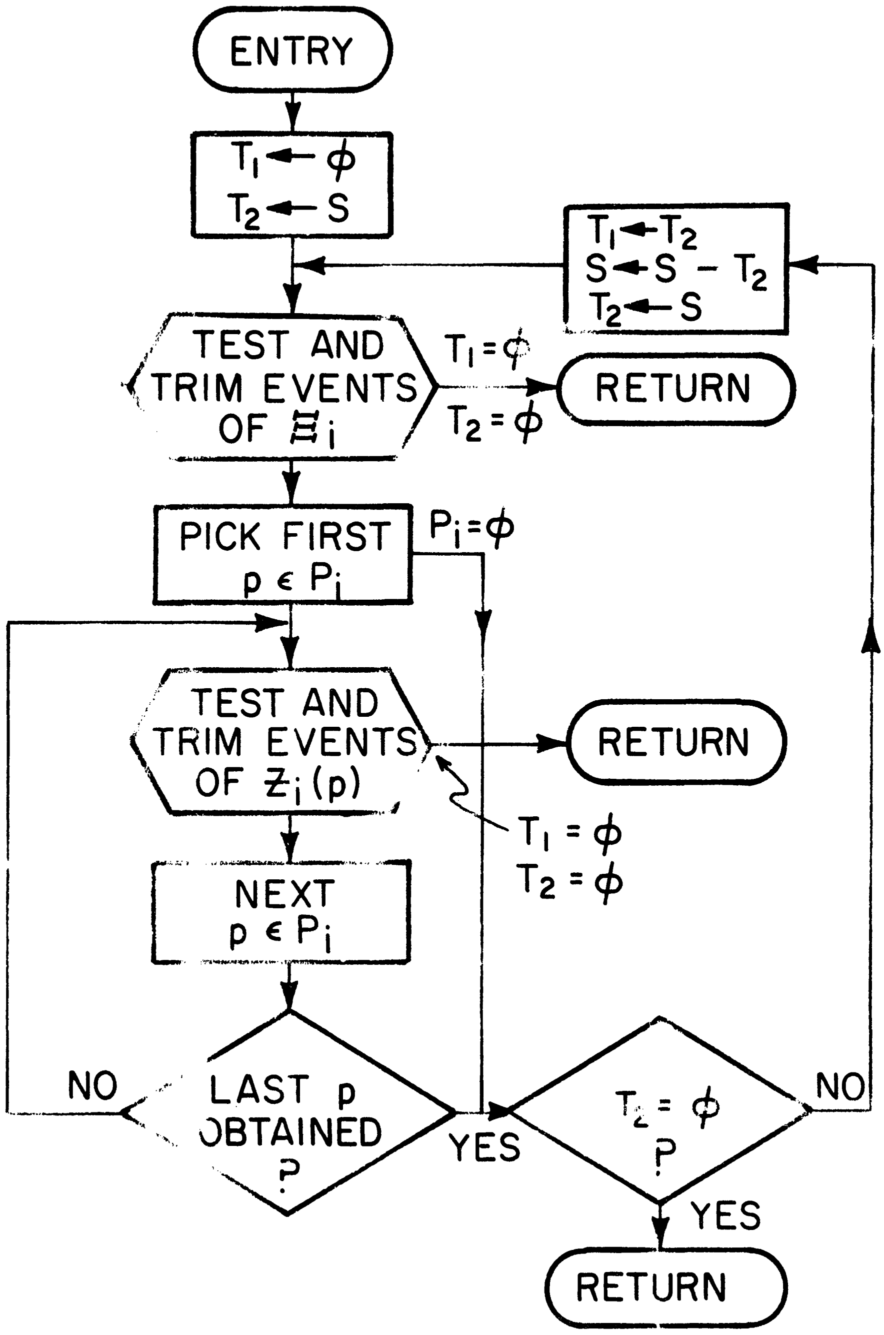


Figure 5.16

State set trimmer flow diagram.

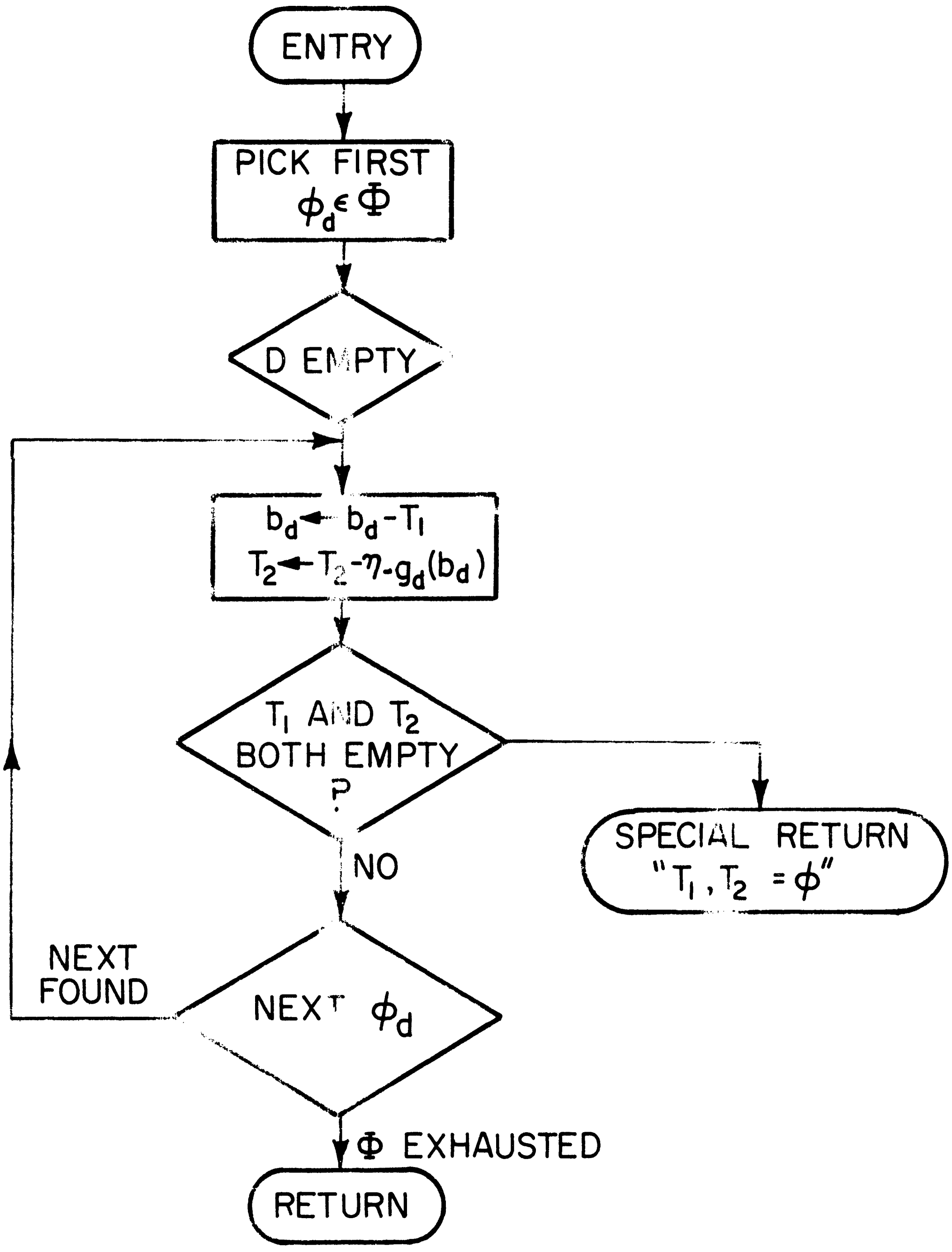

ENTRY
PICK FIRST $\phi_d \epsilon \Phi$
D EMPTY
$b_d \leftarrow b_d - T_1$
$T_2 \leftarrow T_2 - \eta \cdot g_d(b_d)$
T_1 AND T_2 BOTH EMPTY ?
NO
SPECIAL RETURN "$T_1, T_2 = \phi$"
NEXT ϕ_d
NEXT FOUND
Φ EXHAUSTED
RETURN

Figure 5.17

"Test and trim events" flow diagram.

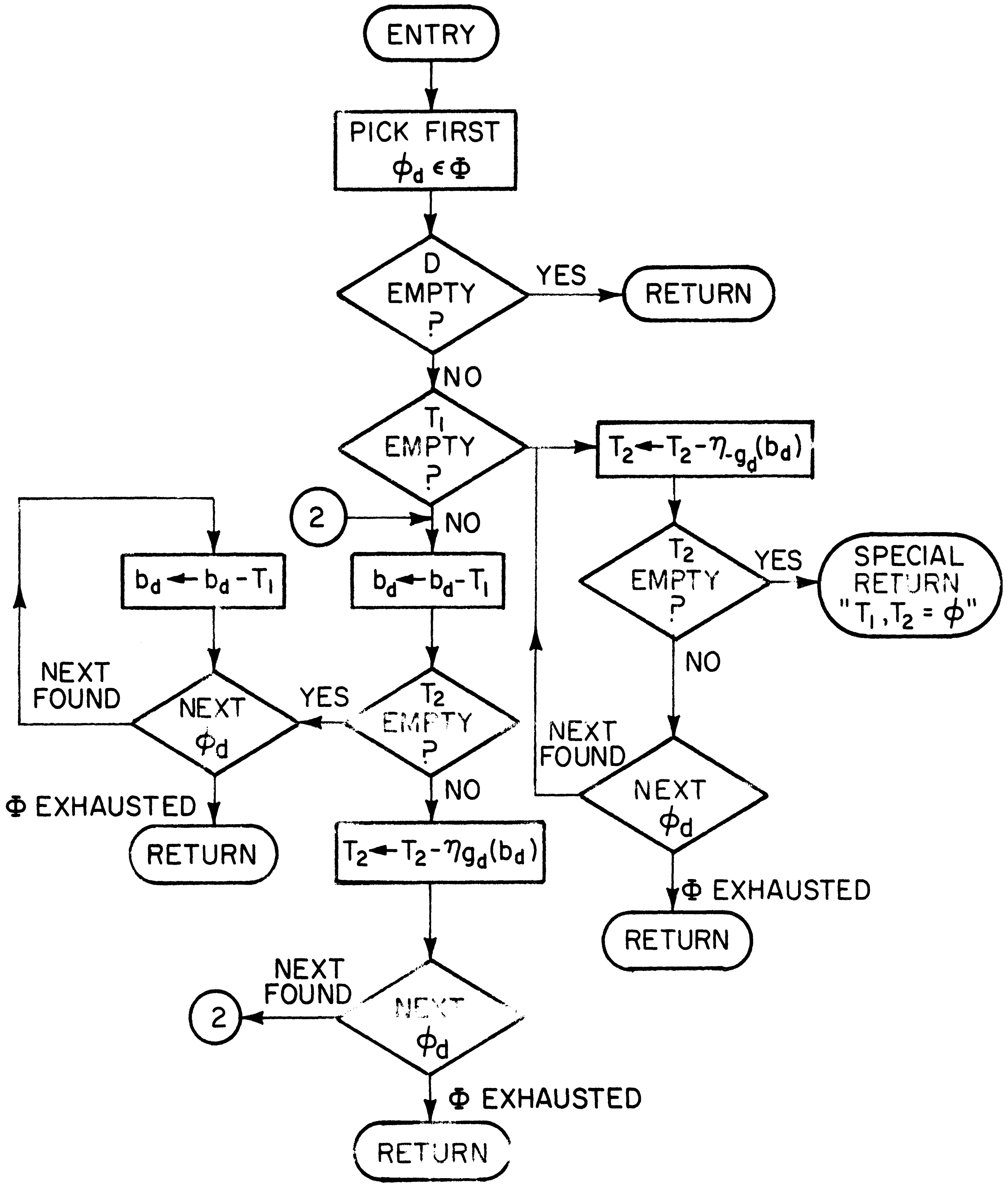


Figure 5. 18

A more efficient "test and trim" program.

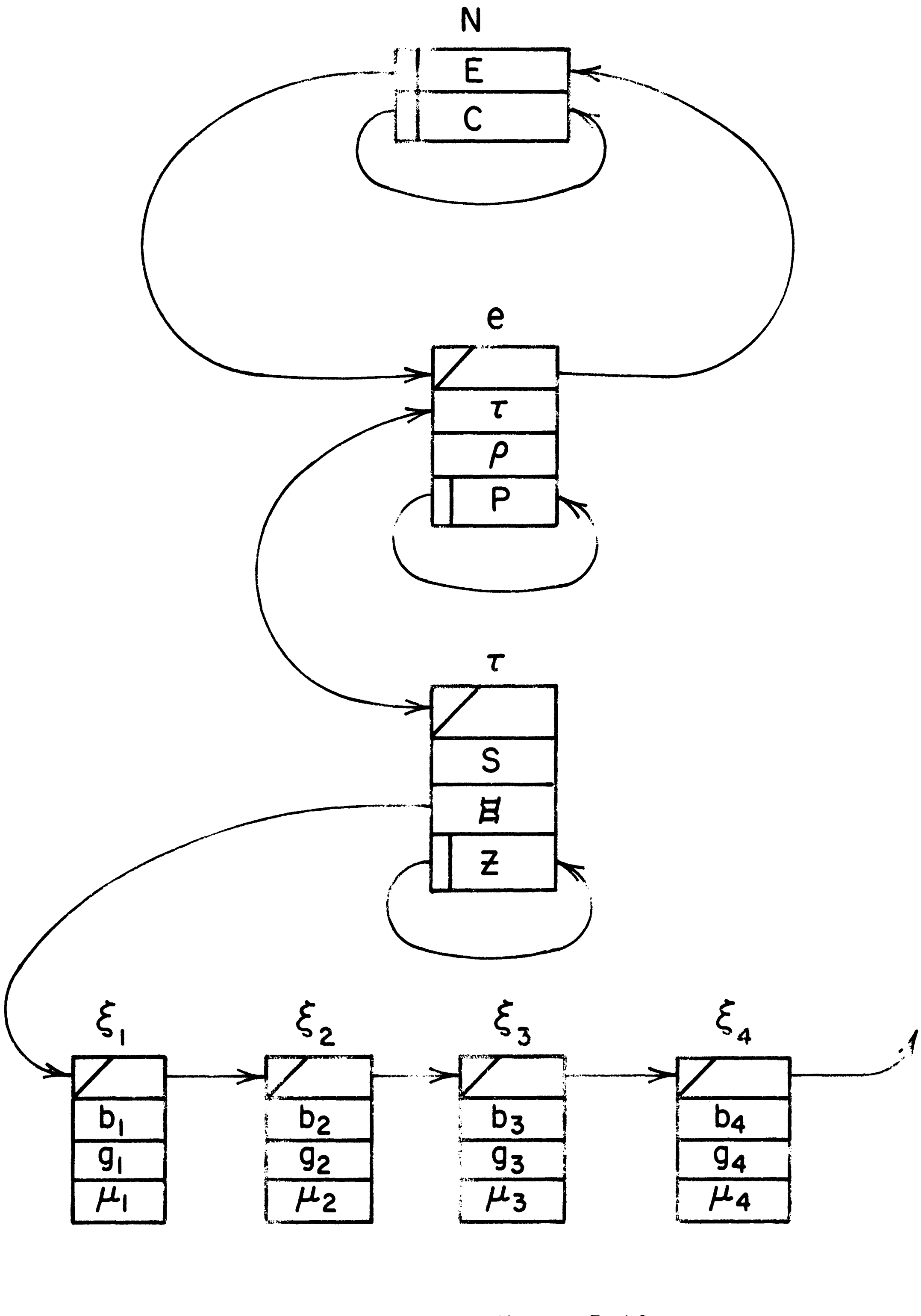


Figure 5.19

Typical result of absorption of all connections.

and a large set of autoevents. The information in this set of autoevents describes the information in the matrix of transition intensities for the network, which must be prepared next.

It is, however, possible that the network will consist of more than one element, with all the elements isolated from one another. There can still be no ports, since there are no connections, and there were originally no unconnected ports. This multi-element condition is the result of the original network consisting of two or more isolated parts. This implies that the model described was, in fact, several models, each of which has been compiled here. At present, this situation can be treated as an error condition. Nevertheless, it offers interesting potentialities for more flexible use of the system, and future implementation should probably allow for it. In that case, procedures would be needed for identifying which isolated network was to be solved, and for what variables. For the time being, however, it is easier to give an error return if the result of absorbing all connections is a network with more than one element.

6. THE STATE AND TRANSITION-MATRIX STRUCTURES

In the numerical calculation of equilibrium probabilities, it is necessary to repeatedly multiply the "matrix" of transition intensities by a probability "vector." The autoevent set resulting from the repeated absorption essentially describes the matrix of transition intensities: the set b_k describes a set of columns of the matrix which contain the intensity μ_k, while g_k describes how far off-diagonal the intensity μ_k is in each of these columns—hence defining the row in which each can be found. The state set resulting from the repeated absorption describes the index set of the probability vectors. (To each state there must correspond a unique probability in the probability vector).

Because of the multidimensional description of states, however, the structure that results from the repeated absorption is unsuited to rapid execution of the matrix-vector multiplication. To use this structure in its original form one must either spread the probability vector over a prohibitively large area of memory so that a very simple formula can be used to find a probability when a state is given, or one must spend enormous amounts of time in the process of state mapping repeatedly upon each multiplication. The large number of dimensions involved makes it likely that a simple linear mapping of a large rectangular prism containing all the states may actually use hundreds or even thousands of times as many locations as there are states. Even so, that mapping would have to be performed twice with each scalar multiplication in the matrix multiplication, and would be dominant in determining the time required for the operation. Any more efficient mapping would require much more time and be completely unreasonable. Such time demands would defeat the primary purpose of this project, which is to provide sufficiently rapid (and inexpensive) response to enable the user to experiment with models on the computer in a symbiotic manner.

As a consequence, the only alternative is to prepare a new structure which is well suited to the matrix-vector multiplication-operation operation, and permits rapid execution with minimal storage requirements.

The expense incurred is the addition of another stage of translation between the consolidated network structure and this new structure. This operation is a part of the compiler and was shown in the flow diagram in Figure 5.2. The objective is to allow the matrix-vector multiplication to proceed with such computational efficiency that the time taken is only ten to twenty percent greater than that taken by the scalar multiplications and additions which are inherent in the matrix operation. This section will describe first the state mapping function and the structure which supports it. It will then describe the structure to be used to represent the matrix. Section 6.3 will finally describe the procedure used to create these structures.

6.1 The State Mapping Function

To obtain an efficient use of memory for the probability vectors, the state set should be mapped onto a consecutive set of integers. This has the advantage of minimizing the number of pages to be used for the probability vectors, and hence decreasing the elapsed time (rather than cpu time) of solution under dynamic paging in the computer. The state set has been represented as a union of rectangles (in n dimensions). Let the state set to be mapped be S, and let

$$S = S_1 \cup S_2 \cup \ldots \cup S_{N_S} \tag{6.1}$$

where S_ν is a rectangle,

$$S_\nu = \{v_{\nu_{11}}, \ldots, v_{\nu_{12}}\} \times \{v_{\nu_{21}}, \ldots, v_{\nu_{22}}\}$$
$$\times \ldots \times \{v_{\nu_{n1}}, \ldots, v_{\nu_{n2}}\} \tag{6.2}$$

Each $v_{\nu_{ij}}$ is an integer, corresponding to appropriate values in the n-tuple structure and n is the dimension of the set, for $\nu=1, 2, \ldots, N_S$.

The mapping will associate a unique integer with every state by providing a simple expansion of each rectangle. Let δ_{ν_0} be the cardinality of the union of the rectangles $S_1, \ldots, S_{\nu-1}$:

$$\delta_{\nu_0} = \# \left[\bigcup_{i=1}^{\nu-1} S_i \right] = \sum_{i=1}^{\nu-1} \# (S_i) \tag{6.3}$$

and let

$$\delta_{10} = 0. \tag{6.4}$$

Then $\delta_{\nu_0} + 1$ represents the state number corresponding to the lowest numbered state in S_ν. The numbering of the remaining states within S_ν is according to the usual rectangular mapping used in most programming systems. Let δ_{ν_i} be the product of cardinalities of the previous i-1 dimentions of S_ν, so that

$$\delta_{\nu_1} = 1 \tag{6.5}$$

$$\delta_{\nu(i+1)} = \delta_{\nu_i}(v_{\nu_{i2}} - v_{\nu_{i1}} + 1), \quad i=1,\ldots,n. \tag{6.6}$$

Then, accordingly, a vector $x = < x_1, \ldots, x_n >$ which is a state in S_ν will be mapped to the integer

$$\bar{x} = \delta_{\nu_0} + (x_1 - v_{\nu_{11}})\delta_{\nu_1} + \ldots + (x_n - v_{\nu_{n1}})\delta_{\nu_n} . \tag{6.7}$$

In this manner, each of the states in S has a distinct mapping $\bar{x}$ and the states in S map to a set of consecutive integers, $\{1, \ldots, \delta_{N_S 0}\}$. Complete information required to use the state mapping, Eq. (6.7), is given by the two-dimensional array $< \delta_{\nu_i} : \nu=1, \ldots, N_S, \; i=0, 1, \ldots, n >$. The <u>state mapping structure</u> produced by the compiler is simply a MAD (or FORTRAN)-type two-dimensional array of halfword integers, assuming (reasonably) that the number of states will always be less than 65,536.

Operators required to construct this structure include one (a "state structure generator") which prepares the structure when the state set in n-tuple set form is given, and another (a "state mapper") which uses this matrix <u>and</u> the n-tuple form of S to obtain the mapped value of a given state according to Eq. 6.7 .

6.2 The Transition Matrix Description

The matrix structure must describe the autoevents Ξ in such a form that no state mapping need be performed during matrix-vector multiplication. It should permit rapid execution and efficient storage for the common forms encountered.

The requirements seem to demand that the structure be in some type of outline form whereby events which occur for rectangular subsets of the state sets can be compactly represented and evaluated through direct indexing on the data of the structure. One such structure was proposed in [3] for a somewhat more general class of problems, and that reference gives some useful ideas. A more specialized structure which appears to be more efficient will be incompletely presented here.

The structure is illustrated in Figure 6.1. A value-line (μ_1 or μ_2 in the figure) indicates the value of a collection of identically valued matrix elements. A block-line (β_0, $\bar{g}$ in the figure) indicates the first row index (β_0) of a subset (called a "block") of the collection of identical matrix elements, and a number ($\bar{g}$) which, when added to a column index, gives a corresponding row index for each member of the block of matrix elements. The members of the block are identified by a series of indexing-lines (β, δ in the figure), which indicate increments

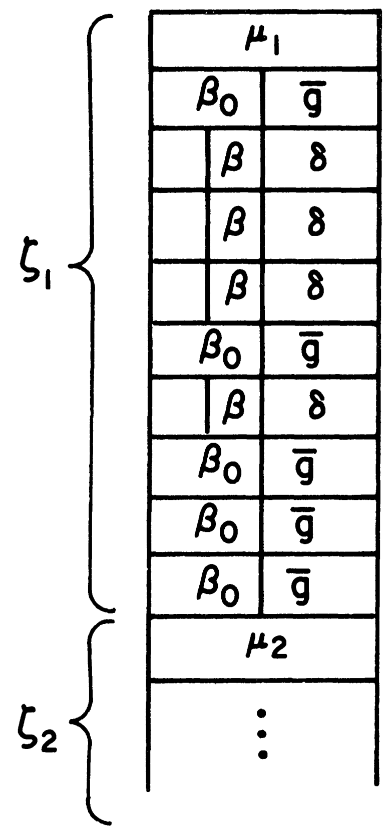


Figure 6. 1 Illustrating a matrix structure.

(δ) to be applied to the column index to get another column index, and counts (β) of the number of times the increment is to be applied. Each line is a word: the β_0's, δ 's, and $\bar{g}$'s are halfwords, the β's are bytes, and the remaining bytes are coded with sufficient information so that line types can be recognized.

This structure, when the details are worked out, should be capable of describing any matrix at least as efficiently as by a list of triples (value, row index, column index), and much more efficiently when element values are repeated diagonally in a regular pattern, as will be the case for QAS transition intensity matrices using the above-described state mapping procedure. Because the information contained in the indexing-lines is exactly the information required for quickly loading and testing "index-registers," rapid execution of the matrix-vector multiplication should also be assured.

To further illustrate, consider the matrix U:

$$\mathcal{U} = \left[\begin{array}{ccc|ccc|ccc|ccc|ccc}
x & & & & & & \lambda & & & & & & & & \\
\mu & x & & & & & & \lambda & & & & & & & \\
\mu & & x & & & & & & \lambda & & & & & & \\
\hline
 & \mu & & x & & & & & & & & & & & \\
 & & \mu & & x & & & & & & & & & & \\
 & & & & & x & & & & & & & & & \\
\hline
 & & & & \mu & & x & & & & & & \lambda & & \\
 & & & & & \mu & & x & & & & & & \lambda & \\
 & & & & & & & & x & & & & & & \lambda \\
\hline
 & & & & & & & \mu & & x & & & & & \\
 & & & & & & & & \mu & & x & & & & \\
 & & & & & & & & & & & x & & & \\
\hline
 & & & & & & & & & & \mu & & x & & \\
 & & & & & & & & & & & \mu & & x & \\
 & & & & & & & & & & & & & & x
\end{array}\right]$$

where only two sets of elements of the matrix are shown. These elements would be described in the matrix structure as in Figure 6.2. The first block is the set of values "μ" in the fourth, fifth, seventh, eighth, tenth, eleventh, thirteenth, and fourteenth rows. Thus the β_0 index is four. Each element of this block is in a column whose index is two less than its row. Thus the $\bar{g}$ for this block is -2. The value repeats twice with unit spacing, and that pair repeats four times with spacing of three. Two other blocks consisting of single elements (i.e. b-lines only) complete the treatment for μ. For "λ" there is a

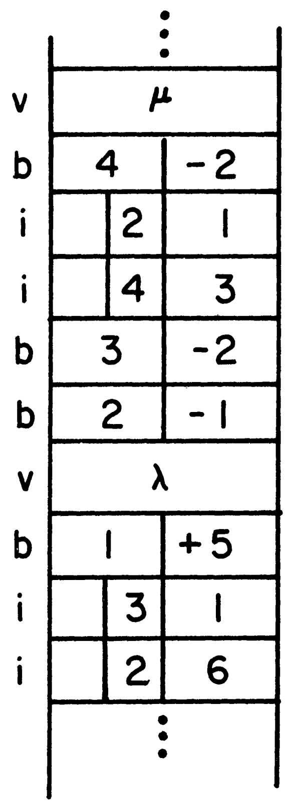


Figure 6.2 The structure for the matrix in eq. (6.9).

single block starting in the first row (β_0=1, $\bar{g}$=5, etc.).

Multiplication of this matrix by a row vector proceeds by considering each block of the matrix structure in succession. The value μ is placed in a register and index registers are set up for nested loops, with depth of nesting equal to the number of i-lines in the block. The β_0 gives the initial displacement of index of the appropriate element on the vector to be multiplied by μ, and $\bar{g}$ indicates the increment to be used to locate the index of the element of the result vector into which the product is to be accumulated.

To be more concise, if y and x are vectors and

$$y = xU,$$

and if a vector y is initially zero, then $y(\beta_0 + \bar{g} + i_1 + i_2 + \dots)$ will be replaced by

$$y(\beta_0 + \bar{g} + i_1 + i_2 + \ldots) + \mu * x(\beta_0 + i_1 + i_2 + \ldots)$$

repeatedly for each $i_1 = 0, \delta_1, 2\delta_1, \ldots, \beta_1\delta_1$; $i_2 = 0, \delta_2, 2\delta_2, \ldots, \beta_2\delta_2$; $\ldots$; $i_q = 0, \delta_q, 2\delta_q, \ldots, \beta_q\delta_q$; where q is the number of i-lines in the block. This replacement is repeated for each block of the matrix structure. The result is the vector y.

By suitable use of registers, this operation should not require significantly more time for execution than is taken by the scalar multiplications and additions alone.

6.3 Creation of Transition-Matrix and State Mapping Structures

The proposed outline structure for matrix representation is well suited to the storage of transition intensity matrices because it is efficient for both isolated matrix elements and for nested, diagonally repetitive elements. These two cases occur with great frequency in transition intensity matrices of Markovian queueing networks.

The final subprogram of the compiler has the task of providing the state mapping structure and the transition matrix structure. These are created in two separate areas: the state area, and the matrix area. These areas are created by suitable calls to the QAS supervisor.

The state and matrix constructor (a subprogram of the compiler) contains two subprograms. The first creates the state area and its structures. The second converts an autoevent into a series of lines in the matrix structure. The state and matrix constructor calls the latter subprogram once for each autoevent in the network. The characteristics of these subprograms are summarized in Table 6.1.

The state area must contain three items: the state set (in its multidimensional form), the ring which identifies the state variables, and the state mapping structure. The first and third are required for the state mapper (which provides a mapped state when a vector is given), the second is required in the specification of results (cf., section 7). All three are required in the generation of the matrix structure and the compilation of the results matrix. Figure 6.3 illustrates a reasonable form for the structure in this area. The symbol table is used precisely as it was in the network and working network structures. The procedure for constructing the state mapping structure should be clear from eqs. 6.3 - 6.6 .

Two difficulties will be encountered in programming the matrix-event generator. In converting an event

$$\xi_k = < b_k, g_k, \mu_k > \tag{6.9}$$

to matrix form, we must note that: first, although a part of b_k may be rectangular, it will map into a "block" in general only if it is also a subset of the state rectangles; and, second, the "increment" g_k will

Program Name	Input Arguments	Operations Performed	Output Arguments	Error Conditions
Create State and Transition Matrix Structures	Pointer to Network	See Table 5.1	None	Network has two or more isolated parts
<u>Subprogram of Above</u>				
State area generator	Pointer to state set	Calls Supervisor to create state area, copies state variable structure and state set into it. Creates mapping structure. (Destroys results matrix, if present.)	None	None
Matrix-event generator	Pointer to an auto-event. Pointer to next line.	Translates auto-event into a set of lines in matrix structure.	Pointer to next line.	
<u>Service Programs</u>				
State mapper	A vector of bytes	Uses state area to determine mapped state.	Value of mapped state	Wrong dimension; vector not in S.
Incorporate rectangle in matrix structure	Pointer to a rectangle b, subset of a state set rectangle S_ν	Creates block-line and indexing lines describing the rectangle b. Moves construction pointer to next line.	Pointer to next line.	None

Table 6.1 State Mapping and Matrix Construction Subprograms (of Compiler)

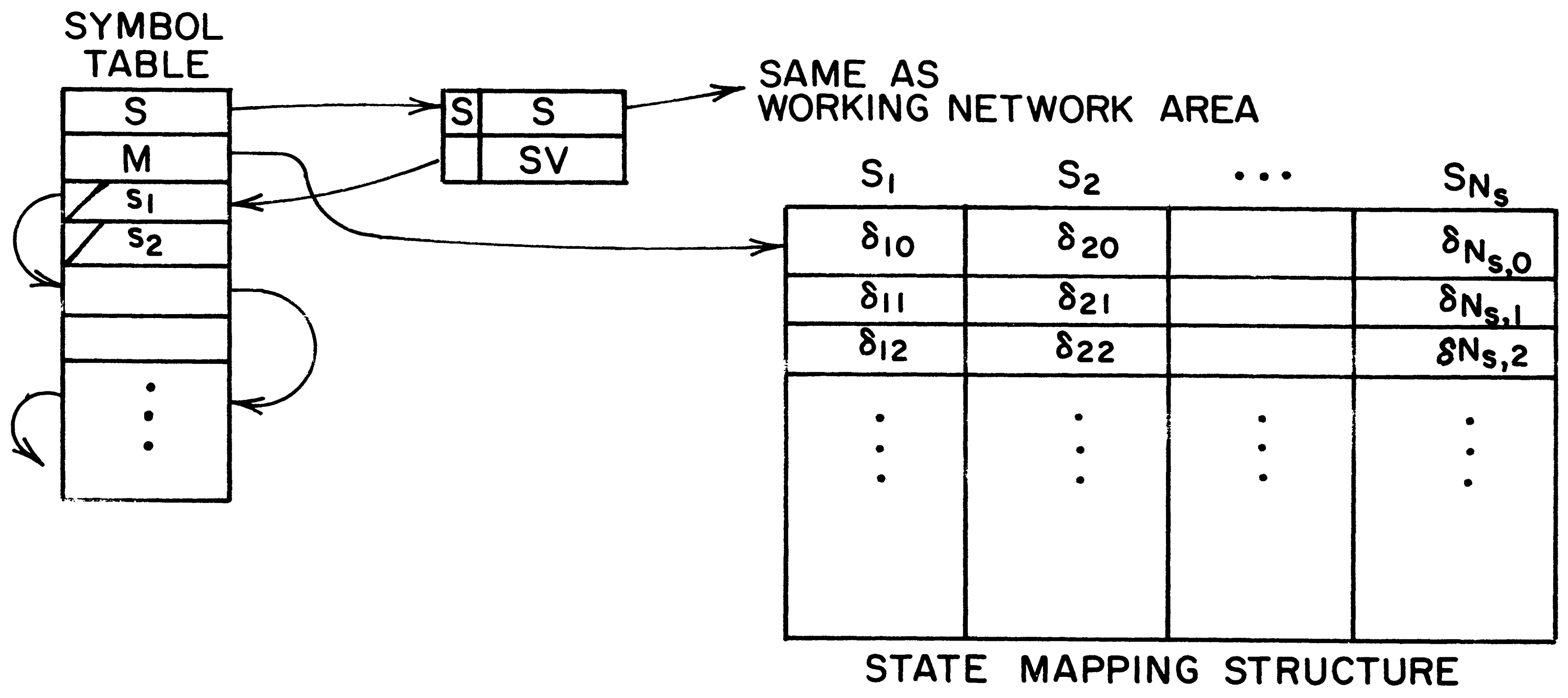


Figure 6. 3

The state area structure (fragment).

map into a constant only when it does not represent a crossing of a boundary between state rectangles. Thus, if the state set were of two dimensions and consisted of two rectangles S_1 and S_2, and b_k were a rectangle contained in S_1, b_k may still need to be represented by more than a single "block" in the matrix, as illustrated in Figure 6. 4. Here the event is assumed to take each state in the direction of S_2.

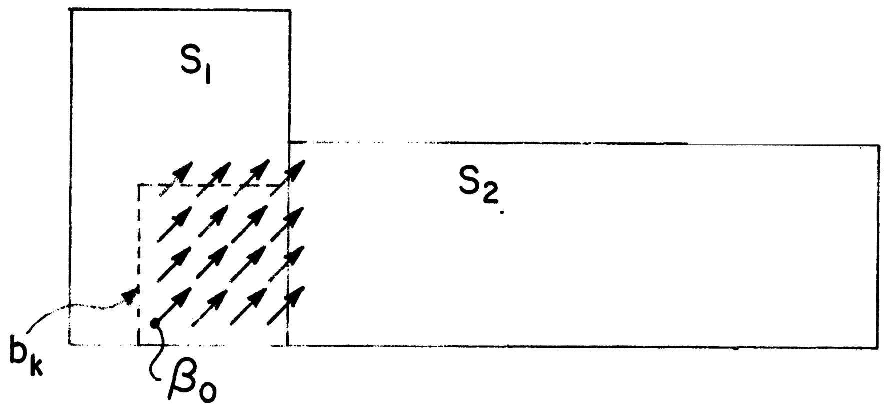


Figure 6. 4

Illustrating "splitting" of b_k for constancy of g.

All but the rightmost column would have a constant mapped increment $\bar{g}_k$, found by simply weighting the i-th component of g_k by the respective δ_{1i}. However, it can be readily shown that the mapped increment $(\bar{g}'_k)$ would generally be different for each of the states in the rightmost column of b_k.

Some procedure like that diagrammed in Figure 6. 5 would have

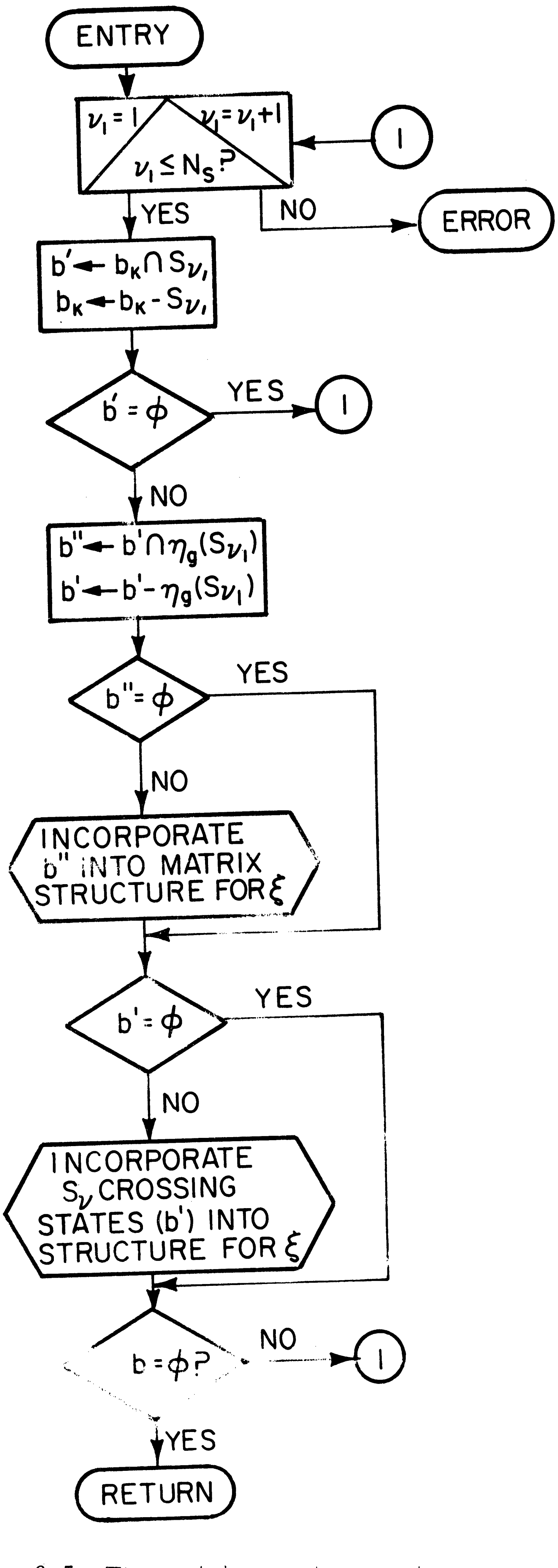

ENTRY
$\nu_1 = 1$
$\nu_1 = \nu_1 + 1$
$\nu_1 \leq N_S$?
YES
NO
ERROR
1
$b' \leftarrow b_K \cap S_{\nu_1}$
$b_K \leftarrow b_K - S_{\nu_1}$
$b' = \phi$
YES
1
NO
$b'' \leftarrow b' \cap \eta_g(S_{\nu_1})$
$b' \leftarrow b' - \eta_g(S_{\nu_1})$
$b'' = \phi$
YES
NO
INCORPORATE b'' INTO MATRIX STRUCTURE FOR ξ
$b' = \phi$
YES
NO
INCORPORATE S_ν CROSSING STATES (b') INTO STRUCTURE FOR ξ
$b = \phi$?
NO
1
YES
RETURN

Figure 6. 5 The matrix-event generator.

to be used to take account of these difficulties. Here, one first determines which of the S_ν rectangles contains which portion of b_k. These are designated b', and are removed from the b_k. Then one determines which of the b' cross into a different state rectangle. These are designated b'', and consist of a rectangle or set of rectangles readily mapped by a block or blocks. The remainder of the b' must be treated individually by a routine (labeled "incorporate S_ν crossing . . . ").

Experience with handworked examples suggests that the boundary crossing cases can fairly frequently be added to existing blocks, but in a nonpredictable way, and that this is sufficiently frequent to warrant considering a scan in the "incorporate S_ν crossing. . . " routine which adds stray terms to existing blocks. However, the cost of such a scan and the typicalness of the handworked examples are still imponderables. In any event, the advantage of treating transitions within a state rectangle by blocks in the structure, rather than individually, seems unquestionable. There should be a small number of state rectangles representing large numbers of states, giving major advantage to the technique.

It should be declared also that since execution of the matrix-vector multiplications will probably not use every indexing line as a register (hence fast) operation, and since order of indexing-lines within a block is irrelevant, <u>the procedures should place the indexing-line with highest repetitivity (β) first</u> in every case, on the assumption that only the first one will involve exclusively register-to-register indexing. The extension of this idea if more registers are available is obvious.

Upon completion of the execution of the state and matrix constructor, the working network area contains no useful information, and will be cleared.

7. THE SOLUTION FOR EQUILIBRIUM PROBABILITIES

Once the matrix structure has been formed, it can be used to calculate the equilibrium probability of each state. A single command will initiate the process, which makes use of three areas: the matrix area, a state probability area, and a working probability area. The latter two areas, which will contain vectors of state probabilities, are created upon entry to this process. The working probability area is temporary and will be destroyed upon completion of the process.

The input of this program is the matrix, a specification of desired accuracy, and a specification of the maximum number of iterations to be allowed (which provides a limitation to the investment in a run, protecting the user from excessive expense for ill-conditioned problems). The output is the equilibrium state probability vector, which is left in the state probability area, and an error estimate, also placed in the state probability area. Table 7.1 summarizes the characteristics of this program.

The numerical technique to be followed is an iteration process of the form

$$\pi_{n+1} = \pi_n A \tag{7.1}$$

where A is a stochastic matrix, and π is a state probability vector. When A is formed from a matrix of transition-intensities in a particular way, and when the initial iterate π_0 is suitably chosen, then the

Routine Name	Input Arguments	Operations Performed	Output Parameters	Error Conditions
State Probability Solver	Matrix, tolerance, iteration count	Sets up equilibrium probability vector areas, and computes the equilibrium probabilities for Markov chain represented by the matrix.	Convergence error, return-condition switch	None

Table 7.1 Program for Solution of Equilibrium Probabilities

limit of this iteration is the equilibrium probability vector

$$\pi = \lim_{n \to \infty} \pi_n \tag{7.2}$$

A detailed treatment of the numerical technique is provided in [1], which describes the prototype program, RQA-1.

The matrix stored in the matrix area is not precisely the matrix of transition intensities, since it does not include diagonal elements. Let $d^T = \langle d_1, \ldots, d_{N_S} \rangle$ be a column vector consisting of the diagonal elements. Let Q be the matrix actually stored in the matrix area. Then the matrix U of transitional intensities is

$$U = Q - D \tag{7.3}$$

where

$$d = Q\underline{1} \tag{7.4}$$

and $\underline{1}$ is a column vector of appropriate dimension whose elements are all unity. Then, according to eq. (12) of [1], with a suitable change of notation, the stochastic matrix A is given by

$$A = \Delta(Q + D) + I \tag{7.5}$$

where

$$\Delta = \frac{.99}{\max\limits_i d_i} . \tag{7.6}$$

Thus the iteration process is

$$\pi_{n+1} = \pi_n(\Delta Q) - \pi_n \Delta D + \pi_n . \tag{7.7}$$

This program follows the basic outline of the RQA-1 program, which we will assume is familiar to the reader. It differs, however, in several major ways.

First, because the matrix stored describes Q rather than U, the iteration process is different. The values of Δ and d must be calculated, ΔQ is formed, the previous iterate π_n is copied from the probability area into the working probability area, and then $\pi_n(\Delta Q)$ is accumulated onto the probability area, forming $\pi_n(\Delta Q) + \pi_n$. Finally, d is used to accumulate $\pi_n \Delta D$ onto the same area, to complete eq. (7.7).

Second, since the matrix Q is formed automatically, there is considerably less demand for testing and diagnostics, and the sizes of the arrays are known and do not need to be calculated.

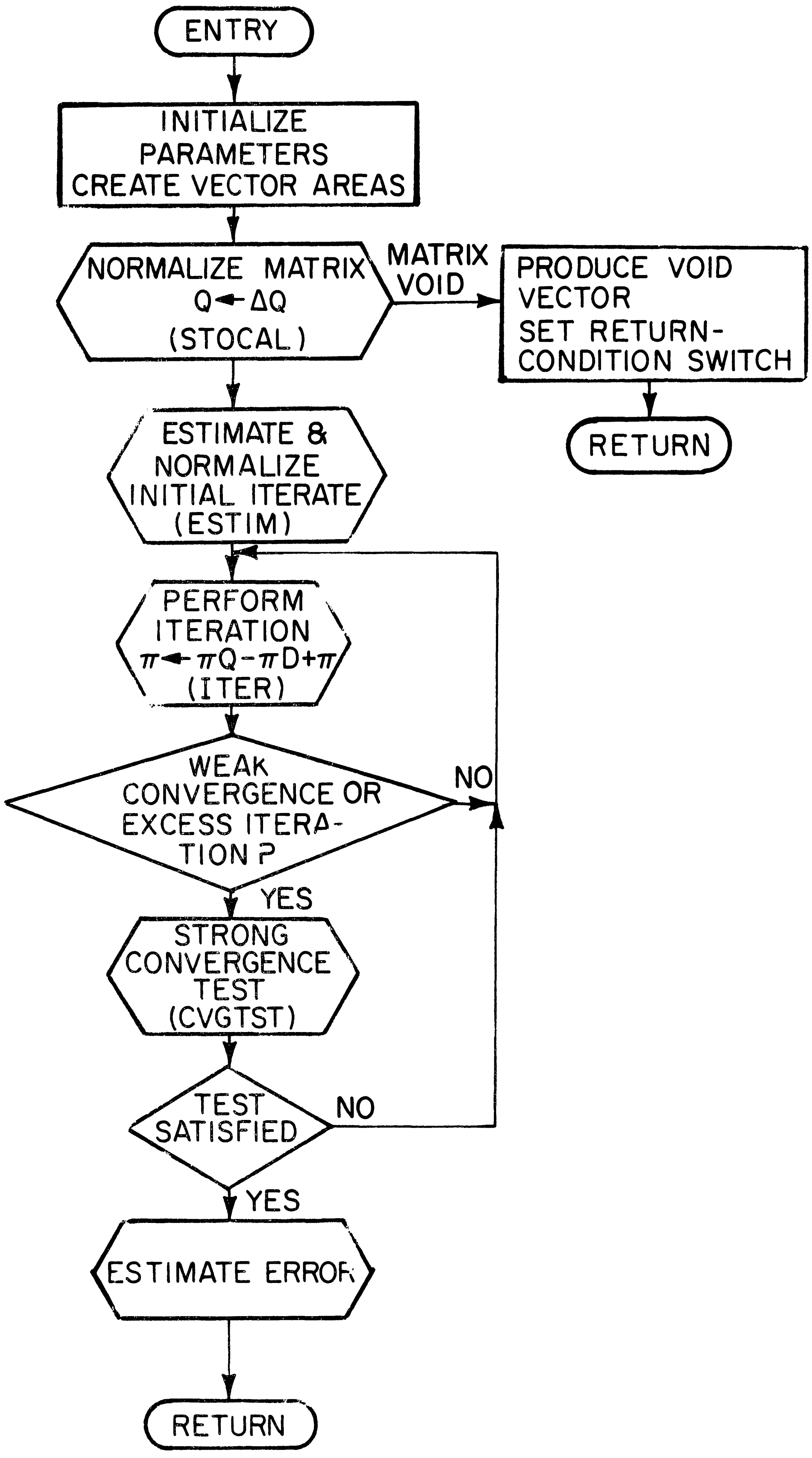
Third, there will be no repeated runs in any particular systematic form, so that the choice of how to form the initial iterate is narrowed. It is also narrowed by the greater remoteness of the user, since he has no knowledge of the state space and is unlikely to want to provide his own initial iterate.

Fourth, this program is designed as a subroutine, rather than as a main program, and the discipline subroutine (DISCPL in RQA-1) is no longer needed.

Finally, output is the responsibility of another program, the result of this program being merely the probability vector.

A flow diagram for this program is shown in Figure 7.1, where the names of corresponding RQA-1 programs are inserted (in parentheses) where appropriate. A default is provided so that, if a matrix has not been prepared, the program supplies a trivial solution. This prevents system shutdown over such errors, with the error having a clear meaning to the user.

It is expected that an interface program will be created which will allow a batch or teletype user to directly supply a list of auto-events and state rectangles, and a description of a results matrix (see section 8), and obtain a solution without involving the compiler and the network generation programs. (In this way we obtain a very powerful, primitive model solver which is akin to, but an improved version of, RQA-1.) This interface program, which calls on several QAS subroutines including the "State Probability Solver," will be called RQA-2.


ENTRY
INITIALIZE PARAMETERS CREATE VECTOR AREAS
NORMALIZE MATRIX Q←ΔQ (STOCAL)
MATRIX VOID
PRODUCE VOID VECTOR SET RETURN-CONDITION SWITCH
RETURN
ESTIMATE & NORMALIZE INITIAL ITERATE (ESTIM)
PERFORM ITERATION π←πQ−πD+π (ITER)
WEAK CONVERGENCE OR EXCESS ITERATION ?
NO
YES
STRONG CONVERGENCE TEST (CVGTST)
TEST SATISFIED
NO
YES
ESTIMATE ERROR
RETURN

Flow diagram for state probability solver.

Figure 7.1

8. SPECIFICATION AND CALCULATION OF RESULTS

The results desired by the user of this system will not ordinarily be the state probabilities in their raw form. Rather, they will be various quantities which are readily computed from the state probabilities. For example, one may desire the probability distribution of the number of tasks in a particular queue; or the mean queue length; or the probability that a particular server is in use; or the mean rate of service completions in that server; or the rate of transfer of tasks through a particular port; or the probability that a task is blocked at that port.

Each of these represents a weighted sum of state probabilities, or a set of weighted sums. In other words, there is a matrix Γ such that the result vector w (perhaps with dimension one) is

$$w = \pi\Gamma \quad . \tag{8.1}$$

In addition, many other useful results, while not of this type, are readily found from results which are of this type. For example, the expected waiting time in a queue (which is the mean number of tasks in the queue, divided by the mean rate of task transfer into its input); or the mean time between service completions of a server (which is the reciprocal of the mean rate of service completions). Such results can be treated as functions of weighted sums

$$W = f(w) \tag{8.2}$$

which are describable by subroutines.

Furthermore, in any particular solution, the user may have several results which he would like to observe simultaneously, so that the vector w may consist of several results strung together, and the function f may be a set of functions on different parts of the vector.

There are four separate operations which QAS must be prepared to carry out to produce output:

(1) The specifications of results must be retained in a storage structure.

(2) The matrix Γ must be prepared.

(3) The vector w must be computed.

(4) The result w must be transformed into the format required for SELMA.

The commands associated with the first operation will be added to the generation phase commands, while the last three of these operations will together constitute a phase called the results phase.

8.1 The Specification of Results

The need for a separate structure which "remembers" the requested results stems primarily from the desirability of allowing SELMA users to specify results at any time during the development of the network model. Often he will want comparable results for several modifications of the network, and will want to change the network without changing specifications. Often, too, he will want to make minor alterations, as an afterthought, long after he has decided what results are to be observed. The specification can not be transformed to its final form (a matrix Γ) until the network is complete and the state mapping structure determined. Thus, the results specifications must be saved, awaiting a command to prepare the matrix and to solve for the results at the appropriate time.

The structure which accomplishes this should be available during the generation of the network, however, and consequently should be placed in the <u>network area.</u> During generation, changes in the network may invalidate existing result specifications, and procedures which are responsible should make adjustments. For example, a queue whose expected length has been specified as a result may subsequently be eliminated from the network. During the operation of the "destroy element" command in network generation phase, a test should be made to determine if a result specification is involved, and if so to eliminate the result specification at the same time.

The reference of a specification to an element or a port is syntactically like a connection, in that a destruction of the object requires a choice of destroying the connection or refusing to destroy the object. Thus, one form which this structure can take is a ring structure form, adding new attributes to the element and/or port blocks and connecting them to appropriate "result blocks." Since the number of results specified will generally be very small, most such attributes would normally have their default (empty) values. A more efficient, but less uniform, structure would provide a simple table of result specifications which can be searched by generation routines and rewritten in the (improbable) event that alterations are required.

A minimum set of result specifications is tabulated in Table 8.1. The purpose of each of these specifications should be self-evident, except for the sixth. By means of this latter specification, it is possible to establish the probability that an input port is in a blocking state, or that an output port has a blocked task. All of the specifications in the table represent weighted sums of state probabilities.

Since commands altering the results specifications will be intermixed arbitrarily with generation commands, and their structures are in the same area, these commands are best treated as additional routines in the generation phase program. Table 8.2 indicates a minimal set of results specification routines needed to carry out alterations of the results specifications.

8.2 The Calculation of Results

The calculation of results, which can result from a single QAS command issued after the compilation of a network, involves, as we've said, three operations: preparation of the matrix Γ computation of the vector w (cf., eq. 8.1), and formatting for output to SELMA. In this section, we will assume that the last operation constitutes a

Type	Title	Modifiers (Arguments)	Result Specified
1	Distribution	Element Name	A vector of marginal probabilities over the range of the state variable of the element is to be computed, giving the probability that each value of the state variable is assumed.
2	Expectation	Element Name	The expectation of the state variable of the element specified is to be computed.
3	Probability	Element name, two integers	The integers specify an interval of the state variable of the element specified. The probability that the state variable is found on that interval is to be computed.
4	Autoevent rate	Element Name	The mean rate of occurrence of all autoevents of the element is to be computed.
5	Flow rate	Element Name, port number	The mean rate of task flow through the port is to be computed.
6	Endocondition probability	Element Name, port number	The probability that the state of the element is in the endocondition set at the port is to be computed.

Table 8.1 Result Specifications

simple supplying of the appropriate portions of the vector w with the copies of the results specifications so that the result can be identified by SELMA. Thus, we assume that SELMA takes responsibility for making necessary further transformations, either of the form of eq. (8.2) or into graphical displayable form.

(If this is not the case, and if QAS is to provide more general results than those which were described in Table 8.2, or if QAS must prepare the output graphics, then further structures must be supplied to accept format specifications and function identifications in the results calculation phase.)

The commands for this phase are summarized in Table 8.3. The only nontrivial operation in the figure is the preparation of the matrix Γ. For this purpose an area is created, called the results matrix area, where the structure describing Γ will be kept. The matrix Γ should probably be represented in matrix outline form, with the modification that the increments g should be replaced simply by the row index of the result. This reflects the fact that repetitive element values in Γ will generally be along columns instead of diagonals. Naturally, the vector-matrix multiply operator will need to be different, reflecting this change.

To illustrate how this process would proceed, consider a column of the matrix corresponding to the probability that a particular queue has length two. The elements of this column will consist of ones at

Routine Name	Input Arguments	Operations Performed	Error Conditions
Create result specification	Type, modifier list	Adds specification to result-specification structure	None
Delete result specification	Type, modifier list	Deletes specification from structure	No such specification (non-fatal)
Delete result references	Element name	Deletes all specifications referring to the specified element	None
Clear results specifications	None	Clears the result-specification structure	None

Table 8.2 Summary of Results Specification Commands
(to be added to generation routines, Table 3.2)

Routine Name	Input Arguments	Operations Performed	Output Variables	Error Conditions
Prepare Results Matrix	None	Prepares a results matrix (in the results matrix area).	None	None
Calculate Results	None	Computes result vector, formats it, and sends it to calling system (SELMA) with suitable identification.	None	None

Table 8. 3 Summary of Results Calculation Routines

each row corresponding to a state for which the queue length is two. This set of row indices is the set found by restricting the state set to the value 2 in the appropriate dimension. An operator ("set restriction") for creating such a set of n-tuples already exists (see Table 4. 4), and is readily used. A set of iteration blocks for the matrix outline corresponding to this column can be generated from the set of n-tuples using programs developed for the transition matrix translation.

The operation of preparing columns of the results-matrix are correspondingly simple for every specification-type in Table 8. 1 except type 5. For this type, the flow rate through a port is to be calculated. Equivalently, the probability intensity of occurrence of a particular spontaneous event must be shown for each state of the network. Since the autoevents which can cause the spontaneous event are not obvious, the desired probability intensities are not easily found, particularly from the information in the network structure.

Nevertheless, the result specified in type 5 specification is valuable, and worthy of the difficulties. What is required is a form of partial compilation of the network, exploring only those connections whose spontaneous events can cause the spontaneous event under examination. By this procedure, each autoevent of the associates of the connection are examined to find the subset of their autocondition sets which are "taken into" the forbidden states of the connection by the autoevent.

The intensity (μ) of this autoevent is added to the matrix outline for the state subset so found. The exoevents of the associates are similarly examined, and whenever a nonempty subset of the endocondition set at a port is found which is "taken into" the forbidden set by the exoevent, the connection at the offending port is absorbed. The new autoevents are then examined as before, and the process repeats until no more absorptions are necessary. This process will usually involve absorption of only one or two of the connections, and can make good use of existing compiler subprograms. All the intermediate results can be kept in the working network area as the compiler does.

Alternatively, the network can be compiled with the connection in question being absorbed last. The desired information is readily found in the process of the final absorption, for which a special subroutine can be written that prepares the column of Γ instead of actually absorbing the connection. Of course, this method is expensive since it duplicates the process of compilation.

The program which prepares the results matrix continues to add-on columns to the Γ matrix, suitably identified with the specifications, until all specifications have been treated. Thus all results are computed simultaneously when the matrix is multiplied by the equilibrium probability vector.

The computation of the matrix Γ as an intermediate result, rather than computing the results directly, will facilitate future generalization of QAS whereby parameter values will be permitted to be changed without necessitating recompilation of the transition-matrix or the results-matrix.

9. EPILOGUE

It has been necessary to describe the system in considerable detail in order to work out the critical interrelationships between the major processes and data structures. In this concluding section, a series of observations will be made about some auxiliary matters which are not central to the main interrelationships, and hence do not need to be presented in such detail. These observations represent a summary of organization concepts, notions for useful generalization, and certain reservations about unresolved theoretical questions.

9.1 Organization

We have described a system for the solution of simple stochastic networks as an amorphous collection of well-defined routines which operate upon a collection of well defined data structures. The organization of this system is expected to be imposed by the calling system (in this case SELMA), and ultimately subject to the whim of a user. Such an organizational philosophy is a natural one for a programming system designed for conversational use, and imposes a minimum constraint upon the thought sequence of the user.

To implement this organizational philosophy, each routine should be designed to function in any position in a command sequence without catastrophe. The consequence of the routine's operation should be the most logical and consistent one that could be expected. In some cases that might have to be an error diagnostic, and a default operation. In any case, the response must be consistent with previous operations.

For example, if a "generation command" (altering the network) is received after a "compile" command, and then a "calculate-probabilities" command is received, the most logical response to the latter is to provide the probabilities for the altered network, rather than use the results of the "compile" (a "transition matrix"). Thus, upon altering a network, the transition matrix from any previous "compile" should be destroyed, and the "calculate-probabilities" should either give an error indication (non-fatal) when no transition matrix is present, or it should automatically call the "compile" program. (If, for some reason, the user desires that the transition matrix be saved before the network is altered, he should use a "documentation" command.)

Similar potential inconsistencies occur in the alteration of results specifications, the calculation of results, and the use of state mapping structure and state probabilities. All of these conflicts can be resolved by insisting that the information in all areas be applicable, at all times, to the current network and results specification, and automatically clearing such areas whenever an inconsistency arises. If this is done, no control over the order of commands will be required of SELMA by QAS. (SELMA may, however, impose such control itself.)

For convenience, the commands, and their associated phases and areas are listed in Table. 9.1. In the column marked "areas cleared," areas in parentheses are those automatically cleared because of induced inconsistency.

9.2 Documentation

Documentation phase commands will provide the ability of a user to save any network structure, transition matrix-state mapping, results specification, or results as a named file, and to restore them to QAS for resumption of analysis. This documentation must be done in such a way that QAS can, at a later time, be restored to an operating condition with the saved network.

Moreover, the file will have to contain sufficient information to restore the displays of SELMA, via dataphone, since the PDP-9 has no facility for long-term saving of display files.

9.3 Deferred Evaluation of Parameters

One of the most crucial improvements which this system requires is a provision which permits compilation of networks and results specification with parameters in the elements' parameter lists treated as variables. With such a provision, the operation of compilation will be expanded into three operations: compilation, evaluation, and calculation. The first operation compiles the network, preserving an arithmetic description of how each parameter-dependent constant can be found. The second operation uses the arithmetic descriptions to place the parameter-

Phase	Commands	Areas (initial)	Areas Acquired	Areas Cleared
Generation	Create Element Connect Assign Parameter Disconnect Destroy Element Alter Gen. Par.	Network	---	(Transition Matrix) (State Mapping) (State Probability) (Results Matrix)
	Create Result Spec. Delete Result Spec. Delete References Clear Result Spec.	Network	---	(Results Matrix)
Compile	Compile	Working Network	Transition Matrix State Mapping	Working Network
Calculate Probabilities	Calculate Probabilities	Transition Matrix	State Probability Working Probability	Working Probability
Calculate Results	Prepare Matrix	Network State Mapping	Results Matrix	---
	Calculate Results	Results Matrix State Probability	---	---
Documentation	(not determined)	---	Any	---

Table 9.1 Summary of Phases, Commands, and Areas Used

dependent constants in the appropriate locations of the transition matrix structure. Then, when a user is exploring a single model with many values of the parameters, the compilation is performed only once and the evaluation (and calculations) are performed for each set of parameter values.

With anticipated modes of use, based upon current usage of RQA-1 which has a similar feature, this will reduce the number of compilations required by an order of magnitude or more. Since compilation is an extensive operation, this is an important economic improvement, and will also provide a dramatic improvement of response time.

Indications are that compile time will not be dramatically increased by implementing this capability, and that evaluation can take considerably less time than compilation. An experimental prototype of one system for accomplishing this was developed [8] for use on the IBM 7090 and demonstrated feasibility.

REFERENCES

1. Wallace, V. L. and R. S. Rosenberg, RQA-1, The Recursive Queue Analyzer, Technical Report 2, Systems Engineering Laboratory, Department of Electrical Engineering, University of Michigan, Ann Arbor, February 1966.

2. Wallace, V. L. and R. S. Rosenberg, "Markovian Models and Numerical Analysis of Computer System Behavior," Proc. IFIPS Spring Joint Computer Conference, vol. 28, 1966, pp. 141-148.

3. Jackson, J. H., "Matrix Storage Scheme Suitable for System 360," Memo to 07449 File, Systems Engineering Laboratory, University of Michigan, Ann Arbor, December 3, 1965.

4. MTS, Michigan Terminal System, Computing Center, University of Michigan, Second Edition, December 1, 1967.

5. Newman, W. M., The ASP-7 Ring Structure Processor, Computer Technology Group Report 67/8, Imperial College, London, October 1967.

6. Wallace, V. L., "Preliminary Description of a Free Core Manager," Memo to 07842 File, Systems Engineering Laboratory, University of Michigan, Ann Arbor, November 9, 1967.

7. Sutherland, W. R., The On-Line Graphical Specifications of Computer Procedures, Lincoln Laboratory Technical Report 405, M. I. T., Lexington, Massachusetts, May 1966. (Appendix C, "The CORAL Language and Data Structure").

8. Jackson, J. H., "Internal Structure of the IBM 7090 Experimental Version of the Deferred Evaluation System (DES-0)," Memo to 07842 File, Systems Engineering Laboratory, The University of Michigan, Ann Arbor, September 15, 1966.

9. Randall, L. S., I. S. Uppal, G. A. McClain, J. F. Blinn, An Implementation of the Queue Analyzer System (QAS) on the IBM 360/67, Systems Engineering Laboratory Technical Report 43, Department of Electrical Engineering, University of Michigan, Ann Arbor, 1969. Concomp Project Technical Report 22, Computing Center, University of Michigan, Ann Arbor, 1969.

10. Wallace, V. L., On the Representation of Markovian Systems by Network Models, Systems Engineering Laboratory Technical Report 42, Department of Electrical Engineering, University of Michigan, Ann Arbor, 1969; also Concomp Project Technical Report 21, Computing Center, University of Michigan, Ann Arbor, 1969.

11. Jackson, J. H., SELMA: A Conversational System for Graphical Specification of Markovian Queueing Networks, Systems Engineering Laboratory Technical Report 45, Department of Electrical Engineering, University of Michigan, Ann Arbor, 1969; also Concomp Technical Report 23, Computing Center, University of Michigan, Ann Arbor, 1969.

APPENDIX A

Preliminary Description of a Free-Core Manager

This is a brief proposal for a programming system which gives and collects blocks of free core during execution of list-manipulating programs. It is assumed here that three properties are required by the programs which use this system:

1. Blocks of core in use must be format free. In other words, there may be no reserved words or bits in blocks which are in use.
2. Blocks of arbitrary size may be requested or released, but certain sizes will be much more frequently used than others.
3. No relocation is possible for a block once it is put to use.

This proposed system is believed to be a satisfactory compromise between time, storage efficiency, and programming effort under the above circumstances.

The area in which the blocks are to be allocated will be considered a large, named vector. Thus, if there is reason to segregate blocks for economic or other reasons to different parts of virtual memory, it is possible to exert such control. (For example: different parts of the list structures to be generated may be in use at different times, and it may be uneconomical to keep them all in virtual memory at once.)

The first n words of each area should represent some number of categories (say eight). Half of each of these words will be pointers to a member of the category or back to itself if there are no members. That is, they each form the head of a ring. The other half word indicates the number of members in the ring. The members of the categories are blocks of free memory in a particular range of sizes. (Usually, most categories will consist of a single commonly used size, with one category being a "miscellaneous" category.) These free memory blocks will have the format shown in Figure A 1. Notice that one-word "blocks" have a special format. For this reason, it is necessary that one-word blocks always be in the first category.

There should be at least three calls to the manager system

```
ALLOC (AREA, SIZE)
GET (SIZE, AREA, ERROR)
FREE (LOC, SIZE, ERROR)
```

Their functions will be as follows:

ALLOC (AREA, SIZE)

Here "AREA" is the name of the vector of length SIZE which is to be used by the manager as its working region. SIZE must be less than or equal to the dimension declared (elsewhere, in the using program) for the vector. ALLOC may be used as often as desired, setting up as many regions as necessary. ALLOC sets up the category heads in the vector suitably linked so that all categories are empty except "misc" which contains one large free block representing the rest of the vector. Of course, ALLOC also puts the right head and tail words on this free block.

GET (SIZE, AREA, ERROR)

GET is a function which locates a free block of the specified size in the specified area. It has value equal to the address of the first word in the located block. If a block of the exact size is not available, GET splits a larger block. If no larger block is available a return is made to the specified error statement label. The second and third arguments are optional with default values "same as last specified in an ALLOC, GET or FREE call", and "system return", respectively.

The procedure for GET is charted in Figure A 2. The operations marked with an asterisk (*) are adjustable procedures which can be controlled so that the sizes of remainders after splits have a distribution close to that of expected requests. They should be under some control of the user, as also the choice of sizes in each category (which is made by a procedure in the FREE subroutine).

FREE (LOC, SIZE, ERROR)

FREE is a subroutine which places a block back into an appropriate category of free core. LOC is the address of the first word of the block to be released, SIZE is its size (an integer), and ERROR is the statement label to which return is made when the entire block is not contained in a current working region. The latter argument is optional, with default value "system return".

In order to avoid splintering of free core, which would result after a long sequence of GET's and FREE's, FREE joins the block being released to its neighbors if they are also free. The chart of Figure A 3 shows how this is accomplished. The symbols "F1(...)" and "F2(...)" represent the left half word and the right half word of ..., respectively.

The operation marked with an asterisk (*) has a portion, loosely under user control, which decides which size blocks to put in which categories. It is through the "*" programs that the assignment strategies can be adjusted to minimize unnecessary splintering by taking account of statistical conditions of the user's list structure. For example, use of categories should be with approximately equal probabilities, and splits of large blocks to assign smaller ones should generally produce remainder blocks of a size which is likely to be needed.

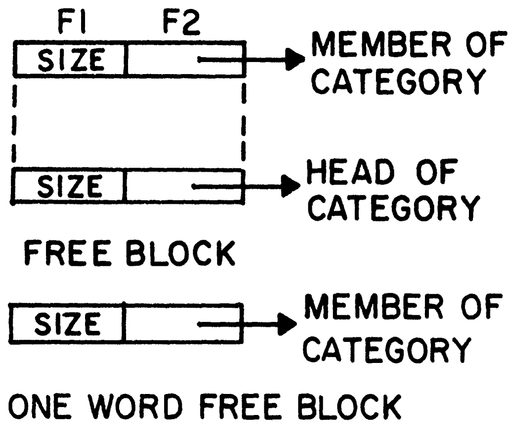


Figure A1

Free Block Formats

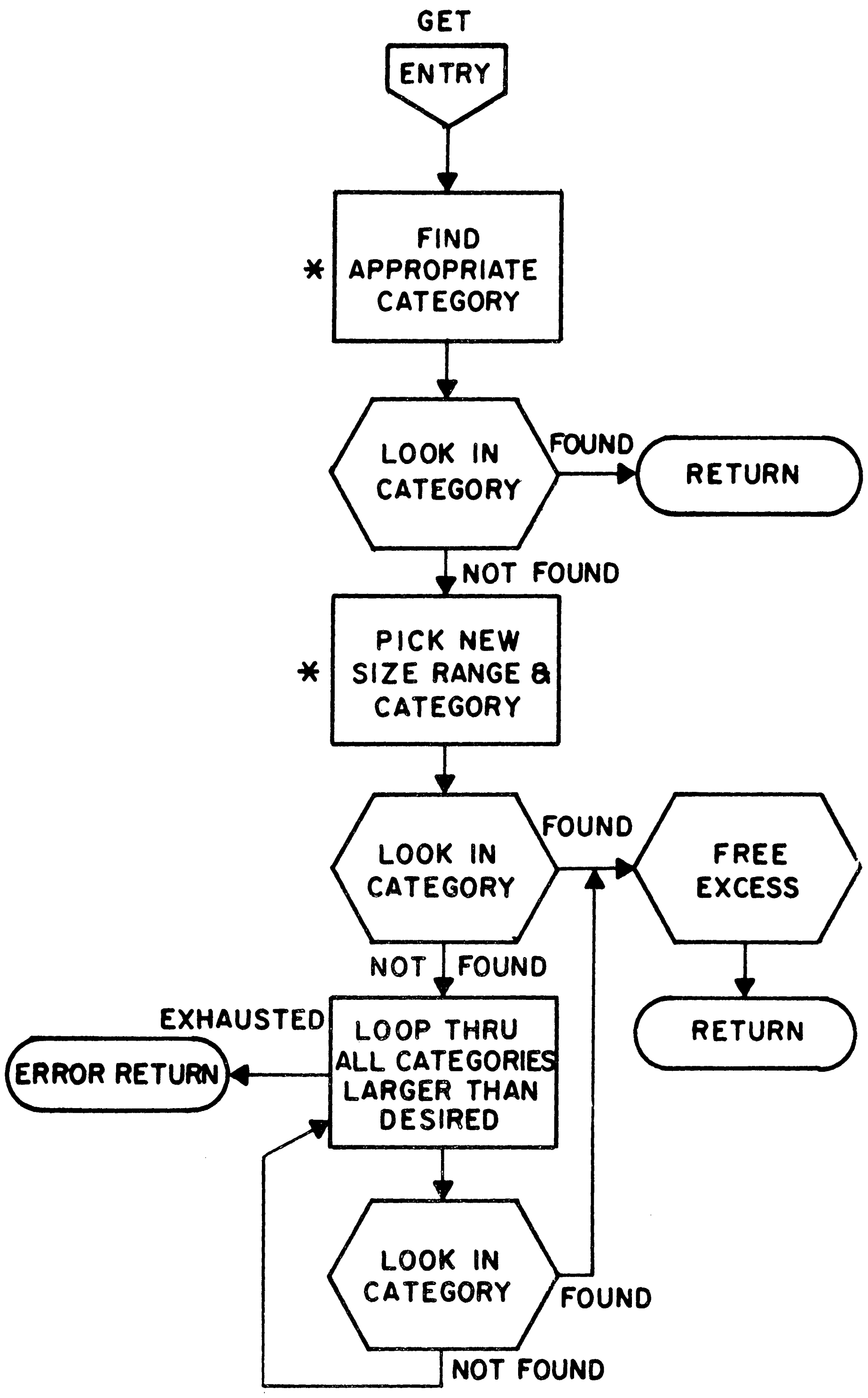


Figure A2
Flow Chart for GET Function

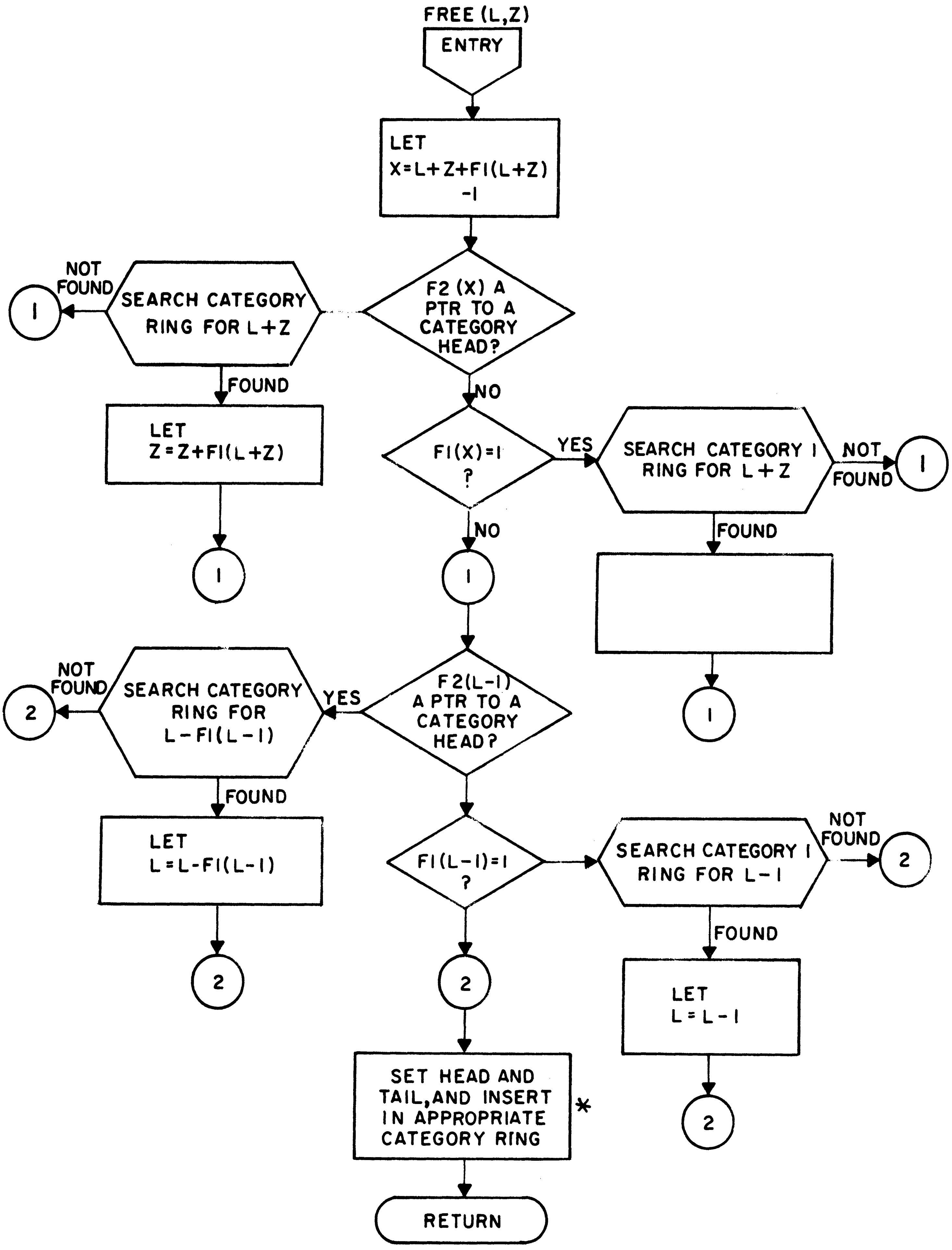

FREE (L,Z)
ENTRY
LET X=L+Z+F1(L+Z) -1
F2 (X) A PTR TO A CATEGORY HEAD?
SEARCH CATEGORY RING FOR L+Z
NOT FOUND
1
FOUND
LET Z=Z+F1(L+Z)
NO
F1(X)=1 ?
YES
SEARCH CATEGORY 1 RING FOR L+Z
NOT FOUND
FOUND
NO
F2(L-1) A PTR TO A CATEGORY HEAD?
YES
SEARCH CATEGORY RING FOR L-F1(L-1)
NOT FOUND
2
FOUND
LET L=L-F1(L-1)
F1(L-1)=1 ?
SEARCH CATEGORY 1 RING FOR L-1
NOT FOUND
FOUND
LET L=L-1
SET HEAD AND TAIL,AND INSERT IN APPROPRIATE CATEGORY RING
*
RETURN

Figure A3
Flow Chart for FREE Subroutine

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* THE UNIVERSITY OF MICHIGAN CONCOMP PROJECT	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP
3. REPORT TITLE A SYSTEM FOR THE SOLUTION OF SIMPLE STOCHASTIC NETWORKS	
4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)* Technical Report 14	
5. AUTHOR(S) *(First name, middle initial, last name)* KEKI B. IRANI and VICTOR L. WALLACE	
6. REPORT DATE September 1969	7a. TOTAL NO. OF PAGES 139 7b. NO. OF REFS 11
8a. CONTRACT OR GRANT NO. DA-49-083 OSA 3050 b. PROJECT NO. c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) Technical Report 14 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)* SEL Technical Report 31
10. DISTRIBUTION STATEMENT Qualified requesters may obtain copies of this report from DDC.	
11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Advanced Research Projects Agency

13. ABSTRACT

This report details the data and program structures for a conversational programming system which translates commands describing a Markovian queueing network into a matrix of transition intensities, and which provides equilibrium distributions and related solutions of the network according to requested specifications.

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
data structures						
Markovian queueing networks						
networks						
mathematical modeling						