PLAN

Part 1 Introduction to machines

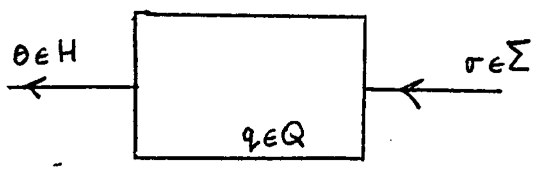
1.1. Finite state machines

This is the simplest model of a dynamic, discrete system. The use of the 'Black-box' concept with an input-output mechanism dependent on the internal state behaviour is a powerful modelling tool.

Let Q be a finite set of internal states
Σ be a finite set of input symbols
H be a finite set of output symbols
and let

$F : Q \times \Sigma \dashrightarrow Q$
and $G : Q \times \Sigma \dashrightarrow H$

be two partial functions, respectively the <u>next-state</u> and <u>output-</u>functions.



The interpretation is this:-

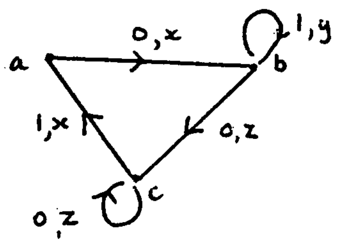
given a system in state $q\in Q$, with an incoming input $\sigma\in\Sigma$ then, at the next time instant, the state changes to
$F(q,\sigma)\in Q$
and an output
$G(q,\sigma)\in a$ is generated.

For notational convenience we replace F by a set
$F_\sigma\colon Q \dashrightarrow Q \quad (\sigma\in\Sigma)$
of next-state functions and G by a set
$G_\sigma\colon Q \dashrightarrow H \quad (\sigma\in\Sigma)$
of individual output functions by the definition

$$\left.\begin{aligned} qF_\sigma &= F(q,\sigma) \\ qG_\sigma &= G(q,{}_\sigma) \end{aligned}\right\} \forall\ q\in Q,\ \sigma\in\Sigma$$

This convention also treats the function notation as a postfix operation.

<u>A simple example</u>



$Q = \{a,b,c\}$
$\Sigma = \{0,1\}$
$H = \{x,y,z\}$

		a	b	c
F	0	b	c	c
	1	–	b	a
G	0	x	z	z
	1	–	y	x

Let Σ^* be the set of all finite strings of Σ,
$\frown$ the empty string, and $\Sigma^+ = \Sigma^* \setminus \{\frown\}$.

We can define a sequential partial function for each $q \in Q$ as follows,

$$f_q : \Sigma^* \longrightarrow H^* \quad \text{by}$$

$f_q(x)$ = the output produced when input string

$x \in \Sigma^*$ is applied to the machine in starting state q. (If at some stage there is no intermediate function defined the machine halts with a partially read input.)

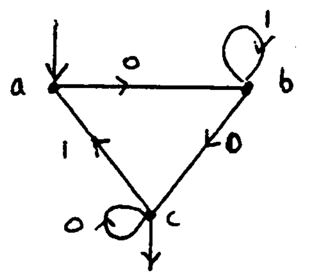
Formally we define: -

$$\left.\begin{aligned} f_q(\frown) &= \frown \\ f_q(\sigma) &= qG\sigma \\ f_q(x\sigma) &= f_q(x) f_{qF_x}(\sigma) \end{aligned}\right\} x \in \Sigma^+, \sigma \in \Sigma$$

where $F_x : Q \longrightarrow Q$ is defined by extension of F_σ under composition.
Further details in [13].

1.2 Recognizers

In some situations we are interested less in the transformation of input strings into output strings than in the answering of a simple yes-no decision problem about the categorization of the set of strings into two disjoint sets. Thus we may present strings to a machine in a specified initial start state and consider only the final state of the computation. Thus the previous example could give rise to the following recognizer;-



Here a is a starting state and c is a terminal state (indicated by unadorned arrows).

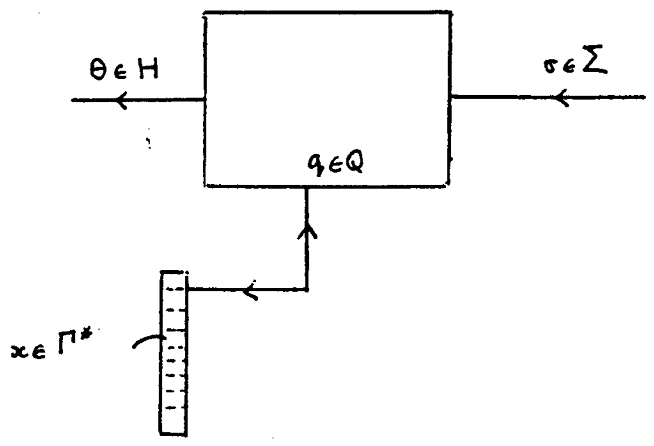
This machine will recognize strings like 010, 01100, 01101010 etc., but not strings like 101, 011, 01101 etc.

The set of strings recognized by a machine is called its behaviour or the language recognized by it.

Such machines are clearly useful in, for example, syntax checkers but they are limited because they possess no explicit memory capability. We now consider this problem.

1.3 Machines with stacks

Suppose that we add a storage device that can store strings of symbols from some alphabet Γ , in such a way that at each state change, depending on the state, the input symbol and the contents of the store either a new string of symbols can be placed on 'top' of the stack store or some of the stack store contents can be removed. This device then allows us to process more complex languages - the context-free languages for example, and can deal with the problem of remembering, say the number of left or right brackets that have featured in a string being analyzed by a syntax checker.



1.4 X-machines

We start with the definition of the X-machine and show how this definition relates to other concepts such as Turing Machines and finite state machines. Then we examine some elementary aspects of the theory of X-machines. It should be remarked that although these machines were introduced in 1974 ([4]) they have not received much attention.

Let X be any non-empty set, henceforth referred to as the fundamental data type, and Φ a finite set of relations defined on X. Thus Φ consists of relations of the form $\emptyset : X \to X$. If one prefers we can regard each $\emptyset$ as a function, which is possibly incompletely specified, from the set X into the set $P(X)$, the set of all subsets of X, (also known as the power set of X).

Intuitively X represents the set of data to be processed and $\emptyset$ are the functions or relations that carry out the processing. In some cases the data type X can represent internal architectural details, such as contents of registers etc. and it is in this way that the model can assume its full generality.

Clearly we need to specify some relationship between the input and output information of the overall system and the data type X, especially when X contains information that is not directly involved with the system input and output. This is done by specifying two sets, Y and Z, to represent the input and output information respectively. In many cases, as in much processing, these sets are free semigroups or subsets of free semigroups (ie languages over some finite alphabet).

Two coding relations:

$$\alpha : Y \to X$$

and

$$\beta : X \to Z$$

describe how the input is coded up prior to processing by the machine, and how the subsequently processed data is then prepared (or decoded) into a suitable output format. Some examples will demonstrate how this works in a few basic cases.

Finally we need to describe some suitable control structure that will actually determine how the processing is performed. This structure is very similar to the state transition graph of a finite state machine and will appear familiar. However, this appearance masks a model of considerable computational power since much of the similarity with finite state machines is concerned with the control of the processing and not with the type of processing that the machine performs. Nevertheless, the similarities with finite state machines are extremely useful since they allow us, at times, to apply techniques for the analysis of machines that have proved to be tremendously successful.

The final ingredient is the <u>state space</u> of the machine, which consists of a finite set, Q, of states and a function

$$F: Q \times \Phi \to P(Q)$$

called the <u>state transition function</u>.

For many purposes this state space can be described using a graph which has the elements of Q at the nodes (vertices) and for each $q, q_1 \in Q$, $\phi \in \Phi$ there is a labelled arc

$$q \xrightarrow{\phi} q_1$$

precisely if $q_1 \in F(q, \phi)$.

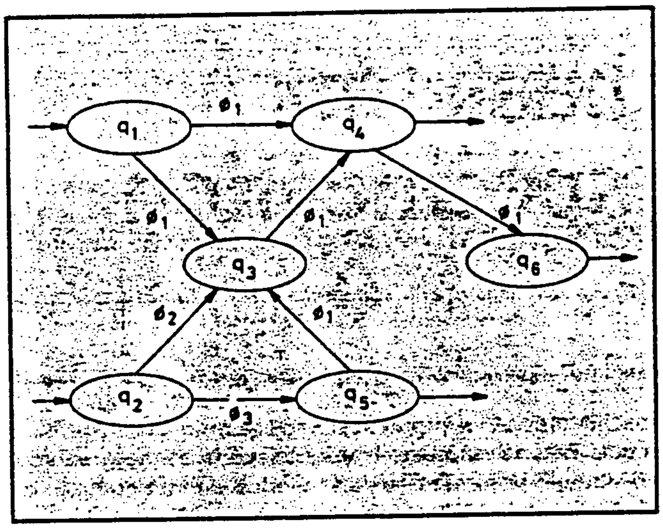
It is also necessary to identify a subset $I \subseteq Q$ of <u>initial</u> states and a subset $T \subseteq Q$ of <u>terminal</u> states. An initial state will be indicated in the state space by being the target of an unlabelled and sourceless arrow, eg

$$\longrightarrow q ,$$

whereas a final state will be described by being the source of an unlabelled and targetless arrow, thus

$$q \longrightarrow .$$

An example of a state space is now given.



The state space of an X-machine

In the diagram states q_1 and q_2 are initial states and states q_4, q_5 and q_6 are terminal states. This example is of a non-deterministic machine, witness the two arrows leaving q_1 labelled with $\emptyset_1$. It is also incomplete in the sense that no arrow labelled with $\emptyset_1$ leaves state q_2.

The formal definition of an X-machine is presented in the following definition.

Definition. An X-machine is a 10-tuple :-

$$M = (X, \Phi, Q, F, Y, Z, \alpha, \beta, I, T) ;$$

where

X, Y, Z are non-empty sets;
Φ is a set of relations on X;
Q is a finite non-empty set;
F: $Q \times \Phi \rightarrow P(Q)$ is a, possibly partial, function;
α: $Y \rightarrow X$ and β: $X \rightarrow Z$ are relations;
$I \subseteq Q$ and $T \subseteq Q$ are subsets.

Remark. The relations appearing in the definition are often functions or partial functions in many examples. The definition is presented here for the record in its most general setting. The set $P(Q)$ denotes the power set (or set of subsets) of Q.

We call Y the input type and α the input relation. The set Z is the output type and β is the output relation.

The process of computation that this machine performs can be described by choosing an element $y \in Y$ from the input type and studying how this element is processed.

First the input relation is applied to the element y to produce an element or set of elements $\alpha(y)$ of X.

Next a path in the state space of the machine is selected that starts from a state in I and ends in a state from T. There may, in a non-deterministic or incomplete machine, be many or none. If a path is selected it will determine a sequence from Φ^* using the labels of the arcs of the path in order. If the labels of the arcs are

$$\emptyset_1, \emptyset_2, \ldots, \emptyset_n$$

then the word

$$\emptyset_1 \circ \emptyset_2 \circ \ldots \circ \emptyset_n$$

defines a composite relation (or function) on the set (or type) X. (In this notation we apply the relation $\emptyset_1$ then $\emptyset_2$ and so on which is a common practice in algebra but may seem unusual elsewhere!)

When this composite relation is applied to $\alpha(y)$ we obtain an element or subset of X and this yields an element or subset of the output type Z on applying β.

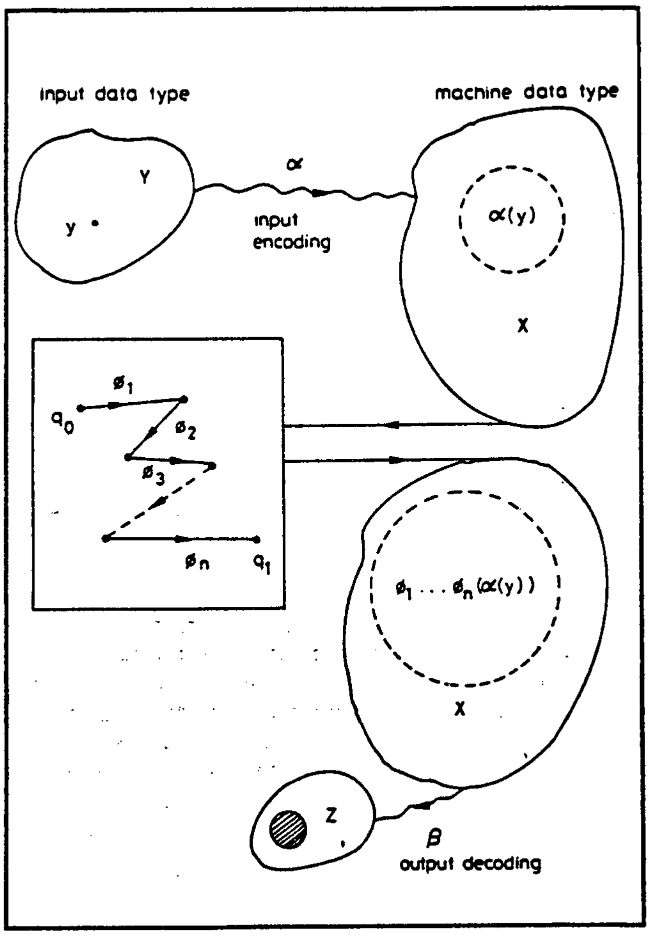
The result of the computation is thus

$$\beta((\emptyset_1 \circ \ldots \circ \emptyset_n)(\alpha(y))).$$

If at any stage we find that the result of a partial computation

$$(\emptyset_1 \circ \ldots \circ \emptyset_k)(\alpha(y))$$

is the empty set for some $k \leq n$ then we will regard that computation as halting and the output, if any, is obtained by applying β as before.



An X-machine computation
$\phi_1 \phi_2 \ldots \phi_n$ is the label of a successful path
ϕ_i: $X \to X$ are relations
q_0 = start state q_1 = final state

<u>1.5 Some examples of X-machines.</u>

The most general model of computation so far investigated in any detail is the Turing machine model and its equivalent theories. There is, however, a newcomer to the scene that is claimed to be more general, namely the Quantum computer of Deutsch. We do not intend to enter the controversy surrounding this new model and its relevance to computer science at this stage, merely note its existence. We will, however, demonstrate that the Turing machine is just a special case of the X-machine defined above.

Before we examine the connections between X-machines and other machines we need to introduce some terminology.

Let Σ be any non-empty set. Some relations will now be defined on the set Σ* of all finite sequences or words in Σ.

For any $\sigma \in \Sigma$ we define some fundamental relations :

$$L_\sigma : \Sigma^* \dashrightarrow \Sigma^*$$
$$(\forall x \in \Sigma^*) \quad xL_\sigma = \sigma x$$

$$L_\sigma^{-1} : \Sigma^* \dashrightarrow \Sigma^*$$

$$(\forall x \in \Sigma^*) \quad xL_\sigma^{-1} = \{ y \in \Sigma^* \mid \sigma y = x \}$$
$$R_\sigma : \Sigma^* \dashrightarrow \Sigma^*$$
$$(\forall x \in \Sigma^*) \quad xR_\sigma = x\sigma$$

$$R_\sigma^{-1} : \Sigma^* \dashrightarrow \Sigma^*$$

$$(\forall x \in \Sigma^*) \quad xR_\sigma^{-1} = \{ y \in \Sigma^* \mid y\sigma = x \}$$
$$left : \Sigma^* \times \Sigma^* \to \Sigma^* \times \Sigma^*$$

(a,b)left=
*(reverse(tail(reverse(a))),head(reverse(a))*b)*

(The purpose of the last string processing function will become clearer when we consider a later example, essentially it transfers the last symbol of the first word to the front of the second word. The standard functions *reverse*, *head* and *tail* are assumed to be defined already as is concatenation, *.)

The Turing machine model. The essential features of a Turing machine consist of an alphabet Σ, a finite set of states Q and a finite set of n-tuples (n=4 or 5) which describe the behaviour of the machine under various circumstances. The set of 5-tuples that we will use here will be elements of the form

$(q, q_1, \theta, \theta_1, d)$

where $q, q_1 \in Q$; $\theta, \theta_1 \in \Sigma \cup \{\hat{}\}$ where ^ denotes a blank ; and either $d = L$ or $d = R$.

The interpretation of such a tuple is that if the machine is in state q and the current symbol being scanned is θ then the next state is q_1, the symbol θ_1 is printed on the tape instead of θ and the read-write head is moved 'left' if $d = L$ and 'right' if $d = R$. Further details and examples of Turing machines will be found in many texts on the theory of computer science .

Added to this is a start state q_0 and a set $T \subseteq Q$ of terminal states. The initial tape contains a string of characters from the set Σ^* which is input to the machine in the state q_0. Processing consists of applying a sequence of appropriate tuples so that if at any stage the machine is in state q and is reading the tape symbol θ then any tuple of the form

$$(q, q', \theta, \theta', d) \quad \text{where } q' \in Q,\ \theta' \in \Sigma \cup \{\wedge\},$$

$d \in \{ L, R \}$ can be applied to yield the next state q', the symbol θ replaced by the symbol θ' and the tape head moved either left or right.

If the tape head moves left then the processing takes a tape of the form

$$[\sigma_1\sigma_2\ldots\sigma_k,\sigma_{k+1}\ldots\sigma_n]$$

with the head reading the symbol σ_k and either produces a resultant tape of the form

$$[\sigma_1\sigma_2\ldots\sigma_{k-1},\sigma_k'\sigma_{k+1}\ldots\sigma_n]$$

where σ_k' is the new symbol printed on the tape after applying the tuple or

$$[\sigma_1\sigma_2\ldots\sigma_{k-1},\sigma_{k+1}\ldots\sigma_n].$$

For a right move the resultant tape is of the form

$$[\sigma_1\sigma_2\ldots\sigma_k'\sigma_{k+1},\sigma_{k+2}\ldots\sigma_n]$$

or

$$[\sigma_1\sigma_2\ldots\sigma_{k+1},\sigma_{k+2}\ldots\sigma_n].$$

In some cases the tuple may involve the replacing of a symbol on the tape by a blank.

In the context of an X-machine we first define the set X as

$$X = \Sigma^* \times \Sigma^*$$

The set of states is Q and the initial and terminal states as in the Turing machine case. For each tuple of the form

$$(q, \sigma, q', \sigma', L)$$

we insert an arrow from q to q' labelled by the relation

$$R_\sigma^{-1} \times L_{\sigma'}$$

on X. For each tuple of the form

$$(q, \sigma, q', \sigma', R)$$

we insert an arrow from q to q' labelled by the relation

$$\emptyset = (R_\sigma^{-1} \times 1)\circ(R_{\sigma'}^{-1} \times 1)\circ left$$

etc. The definition of the input and output relations for the X-machine are given next.

$$\alpha , \beta : \Sigma^* \to \Sigma^* \times \Sigma^*$$

$$(a)\alpha = (\wedge , a)$$

$$(a , b)\beta = a$$

This interpretation is of a Turing machine that behaves as a function on Σ^*. If the machine halts during a computation this means that there is no arrow leaving the current state which has, as a label, an applicable relation. The result is then obtained by use of the decoding relation.

Finite state machines. The classical model of a finite state machine can be represented as an X-machine in the following way.

Let Q be a finite state set, Σ a finite input set and Ω a finite output set then a finite state machine is a quintuple

$$A = (Q, \Sigma, \Omega, F, G)$$

where

$$F : Q \times \Sigma \;{-\!\!+\!\!\to}\; Q$$

and

$$G : Q \times \Sigma \;{-\!\!+\!\!\to}\; \Omega$$

are partial functions defining the next state and output functions.

The X-machine is defined as follows, The set $X = \Omega^* \times \Sigma^*$, the set of states is Q and the sets of final and initial states are also equal to

Q. The set of relations Φ are defined as follows. If $q, q' \in Q$, $\sigma \in \Sigma$, $\theta \in \Omega$ are such that $F(q,\sigma) = q'$ and $G(q,\sigma) = \theta$ then we insert an arrow from state q to state q' labelled by the relation

$$\emptyset = R_\theta^{-1} \times L_\sigma$$

The input and output codes are given by

$\alpha : \Sigma^* \longrightarrow X$ where $\alpha(a) = (\wedge, a)$, $\wedge$ being the empty string ;

and $\beta : X \longrightarrow \Omega^*$ where $\beta(a,b) = a$.

If it is necessary to only carry out computations starting from a given initial state we will define I to be the singleton set containing this state.

The X-machine computes exactly the same sequential function as does the original finite state machine.

In the previous section we gave the general definition of an X-machine and illustrated this with some examples to show that the concept is fully general. In this section we will briefly review some of the theory of X-machines, although at this time this theory is not as well developed as it might be.

The definition of the behaviour of an X-machine can be made in terms of the function or relation that it computes or in terms of the language it recognizes.

Let $M = (X, \Phi, Q, F, Y, Z, \alpha, \beta, I, T)$ be any X-machine. If

$$c : q_0 \xrightarrow{\emptyset_1} q_1 \xrightarrow{\emptyset_2} q_2 \longrightarrow \dots \xrightarrow{\emptyset_n} q_n$$

represents a sucessful path in the state space of M, so that $q_0 \in I$ and $q_n \in T$, then the relation

$$|c| = \emptyset_1 \circ \emptyset_2 \circ \dots \circ \emptyset_n : X \longrightarrow X$$

will be called the relation defined by that labelled path.

The behaviour of M is then

$$|M| = \bigcup |c| \quad : X \longrightarrow X$$

where the union is taken over all the successful paths in the state space.

The relation computed by the machine is then defined as

$$f_M = \alpha \circ |M| \circ \beta : Y \longrightarrow Z$$

For the recognition of languages we define the output set to be $\{\wedge\}$ and the output function $\beta : X \longrightarrow Z$ yields a subset

$$A = \wedge f_M^{-1}$$

of Y.

The article [1] discusses some of the applications of this material. We can develop a methodology for the description of systems by a combination of the data type methods of the first sections with the machine based methods of the latter ones. In situations when architectural features of the system are important, these can be incorporated into the X-machine by defining the set X suitably, perhaps including models of registers etc.

Part 2 Modelling systems with machines

2.1 Hardware description models

2.1.1. 3-bit shift register

Inputs:- Control signals { SR = shift right, SL = shift left }

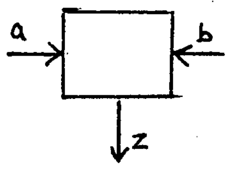
Data inputs { 0, 1 }

States Q = {000,010,...,111}

Table State

	000	001	010	011	100	101	110	111
SL0	000	010	100	110	000	010	100	110
SL1	001	011	101	111	001	011	101	111
SR0	000	000	001	001	010	010	011	011
SR1	100	100	101	101	110	110	111	111

2.1.2 CIRCAL - Phase detector



two input ports, one output port.

$$\begin{tikzcd}
111 \arrow[r, "ao"] \arrow[d, "bozo"'] & 011 \arrow[r, "bo"] & 001 \arrow[r, "al"] \arrow[d, "blzo"] & 101 \arrow[lll, bend right, "bl"'] \\
100 \arrow[rrr, bend right, "ao"'] & 110 \arrow[l, "bo"] \arrow[u, "aozl"] & 010 \arrow[l, "al"] & 000 \arrow[l, "bl"] \arrow[u, "alzl"']
\end{tikzcd}$$

The states are {000,001,011,...,111}. The inputs are {ao,bo,al,bl}.

These model the values at the input ports, and the outputs are zo and zl.

The state diagram describes the state changes and outputs in an obvious way.

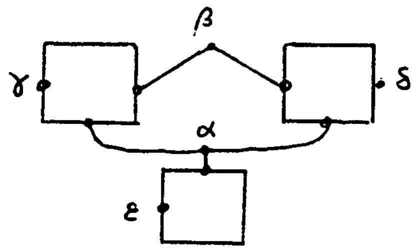
CIRCAL is a language that allows the diagram to be represented symbolically, eg.

PD(111) <- bozo PD(100) + ao PD(011)

Where PD is the name of the component (Phase detector) and the line means:-

"from state 111 either an input 60 causes a change to 100 and an output żo or an input ao causes a change to 011."

When several components are interconnected this can be represented as a diagram where ports with identical labels are joined together:-



and these diagrams can be represented symbolically using the language.

This language can be used both as a behavioural model and a design language at various levels thus enabling the design process from functional specification to fabrication layout design to proceed in a unified and systematic way.

See [5].

2.2 Lexical analyzer

This modelling of a lexical analyser is useful for a variety of software development reasons. It illustrates a particularly simple way of design refinement.

Consider some programming language for which certain lexical categories are defined. Suppose that we wish to identify strings of input characters in terms of this categorization and pass them on with identification labels to further (semantic analysis).

To start the design process off we will consider some specific lexical categories and design a machine that will process them.

Let L = the set of letters
D = the set of digits
E = the set of punctuation, operator etc. symbols.

Special characters : $ for end of file etc.

<u>Lexeme definitions</u>

(I) Identifiers = $L.\ (L \cup D)^*$
(N) Numbers = D^+
(R) Reals = $(D^+.\{.\}.D^*) \cup (\{.\}.D^+)$

Rules for identifying lexemes :

(i) Lexemes are separated by blanks (which have no other significance).
(ii) Ends of lines behave like blanks.
(iii) The next lexeme is the longest legal possibility.
(iv) These are all the lexemes.

A BNF definition would be more usual for the syntax of the language.

A finite state machine could not store the contents of a string until the lexeme types had been identified. We need to complement the machine with a simple buffer that is then used to store symbols as they are read and then flushed at the end. To this end we assume the existence of a buffer which can contain strings of symbols. There are several actions that the machine will need to carry out on this buffer during processing.

Buffer actions :

(i) A - *append* the current input symbol to the buffer.
(ii) S - *scan* the next symbol in the input string.
(iii) F - *fail*, the most recent string of symbols is illegal.
(iv) O (*) - *output string with label* * , where
 * is identifier (I)
 or number (N)
 or real (R)
 etc.

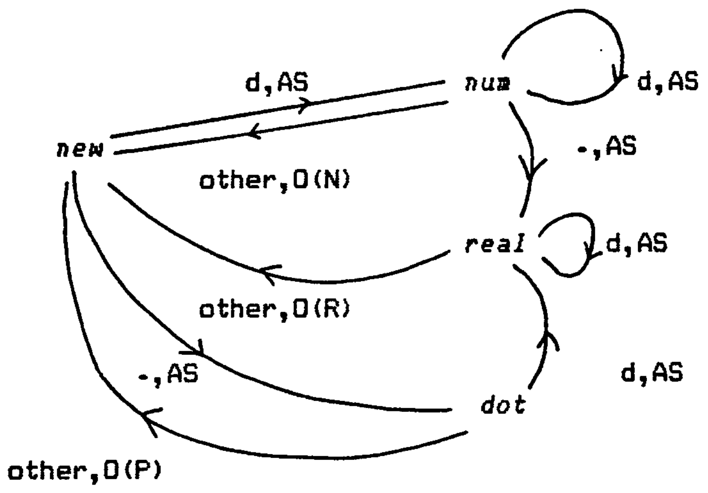
We build up the machine incrementally with a common start state called *new*.

Submachine for identifiers :

$$\begin{tikzcd}
\textit{new} \arrow[r, bend left, "{1,AS}"] & \textit{identifier} \arrow[l, bend left, "{other, O(I)}"] \arrow[loop above, "{1,AS}"] \arrow[loop below, "{d,AS}"]
\end{tikzcd}$$

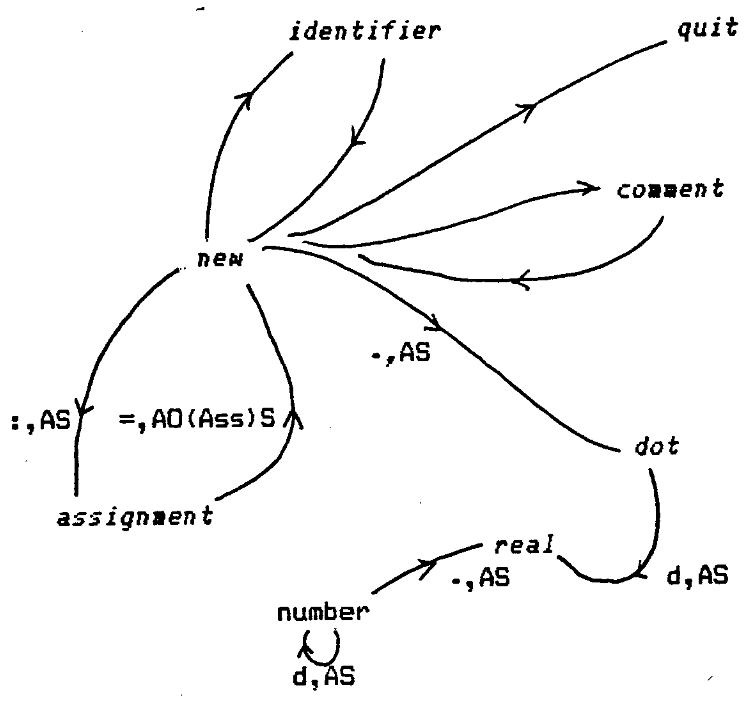
(any symbol other than a digit or a letter in state *identifier* will cause the buffer to flush the identifier string together with an identifier label and reset the machine to *new*, AS means *append* the current symbol and then *scan* the next one.)

Submachine for numbers and reals :



(Here P means punctuation)

Continuing in this way we can build up a complete machine to deal with the analysis of the lexemes defined for the language. Part of this machine is shown below.



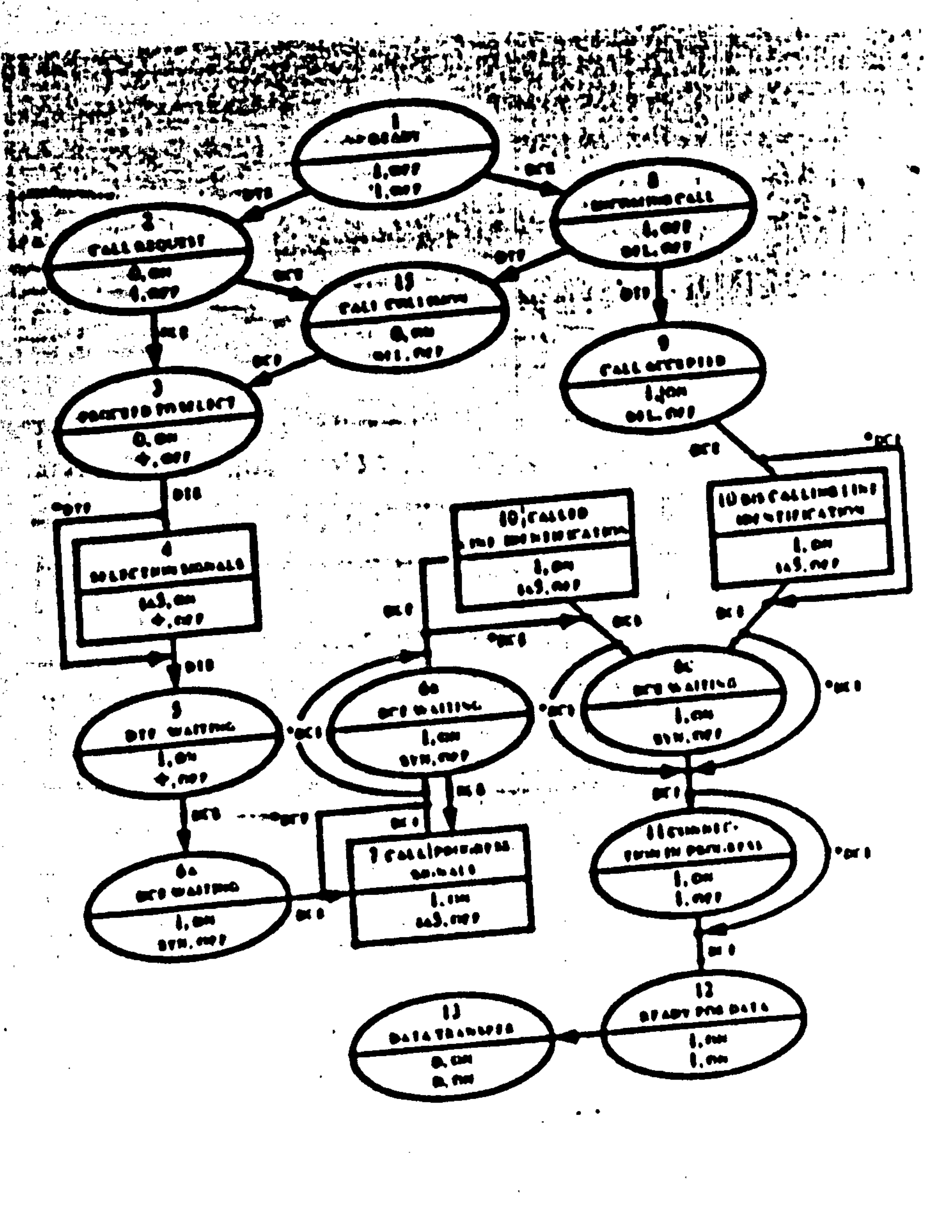
This example is based on Wulf, Shaw, Hilfinger & Flon [6] who go on to consider the implementation and verification of such a system.

2.3 A communications protocol - CCITT X 21.

The X 21 interface protocol for digital telecommunications systems is defined in a complex document which includes the 'dynamic' specification of the interaction between two important components as a finite state machine.

The two systems are called : a *data terminal equipment* (DTE)
and a *data circuit termination equipment* (DCE).

The state diagram describes the logical relationships between events at the interface between the two systems. Each state is identified by four parameters, two for each system and the overall system is described in terms of these states. In the diagram the ellipses describe the states with the 4 parameters together with identification names and numbers. The rectangles denote sequences of states that for these purposes can be identified with one another. This demonstrates the facility that state machines have for information hiding and hierarchical design practices. The article [7] by West & Zafiropulo gives more details.


1
READY
2
CALL REQUEST
3
PROCEED TO SELECT
4
SELECTION SIGNALS
5
DTE WAITING
6a
DTE WAITING
7 CALL PROGRESS SIGNALS
8
INCOMING CALL
9
CALL ACCEPTED
10 CALLED LINE IDENTIFICATION
10 DCE CALLING LINE IDENTIFICATION
11 CONNECTION IN PROGRESS
12
READY FOR DATA
13
DATA TRANSFER
15
CALL COLLISION
DTE
DCE

2.4 Specifying user interfaces

One fundamental problem in software engineering is the design of user interfaces. It seems that formal methods enthusiasts are interested in all aspects of system design except the part that probably matters most! There are a few groups working on this problem and it seems to be something of a neglected 'branch line'. Perhaps the real problem is that it brings us face to face with psychology and somehow this is alien to our formalising instincts.

We will examine some simple ideas for the formal specification and analysis of human-computer interfaces. Examples will be taken from a variety of systems examined by some of my students.

Interfaces today are often 'modal' in that they are designed around a hierarchical set of menus. Whether the menu choices are made by hitting a key or by positioning and clicking a mouse is not too important ; that is more to do with ergonomics. The Xerox-Macintosh style window management systems are widely regarded as the ultimate style of interface but this is a trivialisation of the subject. Users of window systems can still get lost if the underlying structure of the interface is unsupportive or obstructive. The problem behind all this is of course the meaning of the term 'usability'. People are all very different and this is the trouble when looking for general design principles and methodologies.

Our first example is of a hypothetical information system which store record in a file structure and allows the user to *enter* new records, *delete* existing records, *list* related records and *show* a specific record. Each one of these functions will require a specific function space which is designed to prevent improper activity which may cause problems and to allow a simple conceptual model of the system to be acquired by the user.

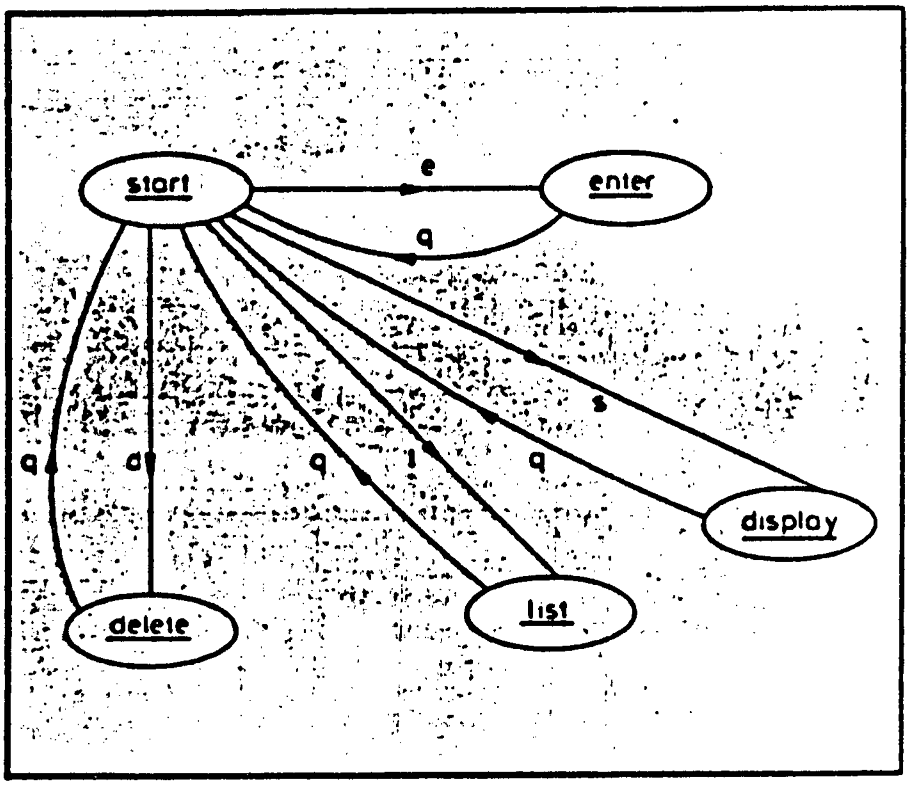
The design process again consists of a series of more and more detailed state diagrams that are arranged in a hierarchical manner.

First we have to define the data types associated with the functions; the functions must also be formally specified and we use *Z* for this purpose here.

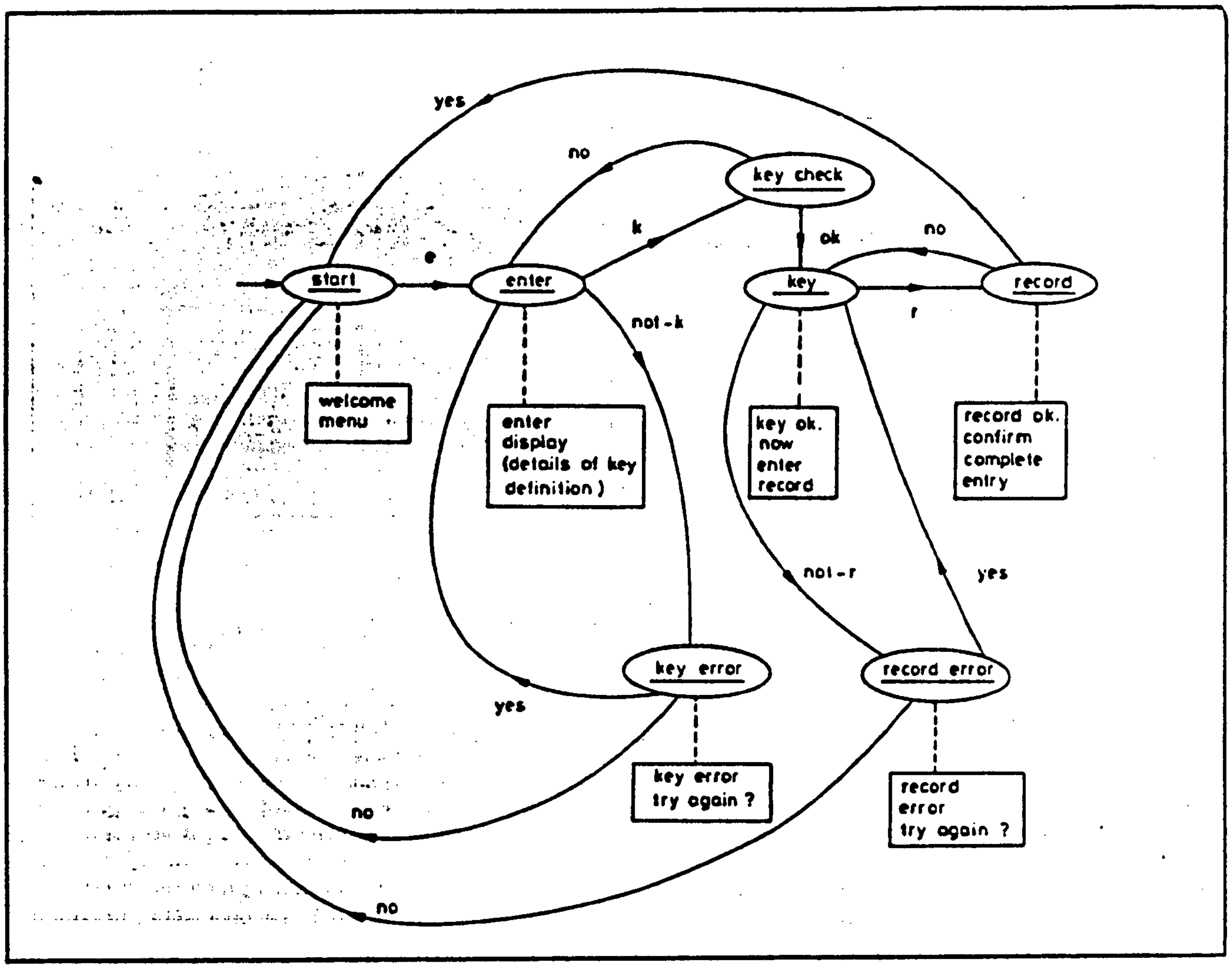
The file will be reharded as a partial function from a set of keys to a set of records which are products of the field data types. In this case the keys will be author name, year, (number of work in that year) and this is described in the *Z* schemas below.

The functions *enter* etc are also specified.

The first diagram is the top level machine which allows for the choice of the desired function. The boxes contain the appropriate screen displays.



The basic state diagram
e, d, l and s denote enter, delete, list and display commands, respectively



The first machine model for enter

At the next level we introduce the details of the *enter* function space using the symbol *k* for an arbitrary key element and *r* for a record element.
Note that the diagram is still hiding information since we have not used the specific syntactic nature of *k* or *r* yet. This is done at the next stage where the design is extended to the level of key-stroke detail. At each state, as we are analysing the design and preparing for the next, more detailed stage, we consider the needs of the user and support in the form of error trapping, screen displays and other measures.
Different types of errors, both syntactic and semantic can be dealt with here.

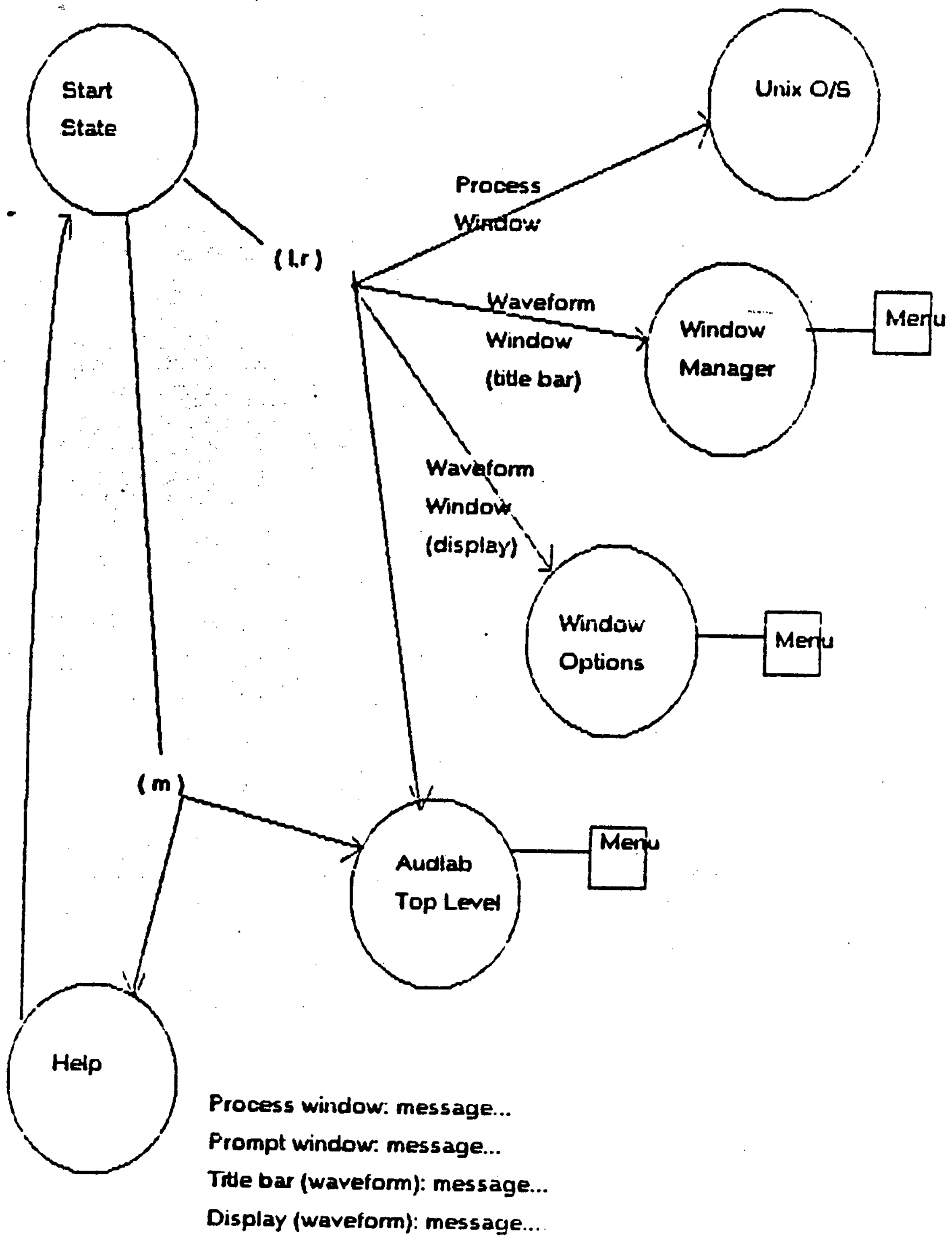
The use of a combination of formal specifications in Z of the data types and the use of state machine diagrams for the dynamic aspects of the interface design has proved a very usable way of specifying interfaces. Several students have successfully specified a variety of systems for example the top level of MS-DOS, Unix, and a popular word processor package. These studies demonstrate that the 'average' designer can deal with the methodology. This is not always the case with formal methods.

We conclude this section with an example taken from another of my postgraduate students, Martin Combs, who has specified and analysed in some detail some (appalling) speech processing packages used by the linguist fraternity. ([8])

Some of these packages are driven by a window system and some are traditional menu-based systems that rely on function key selection.

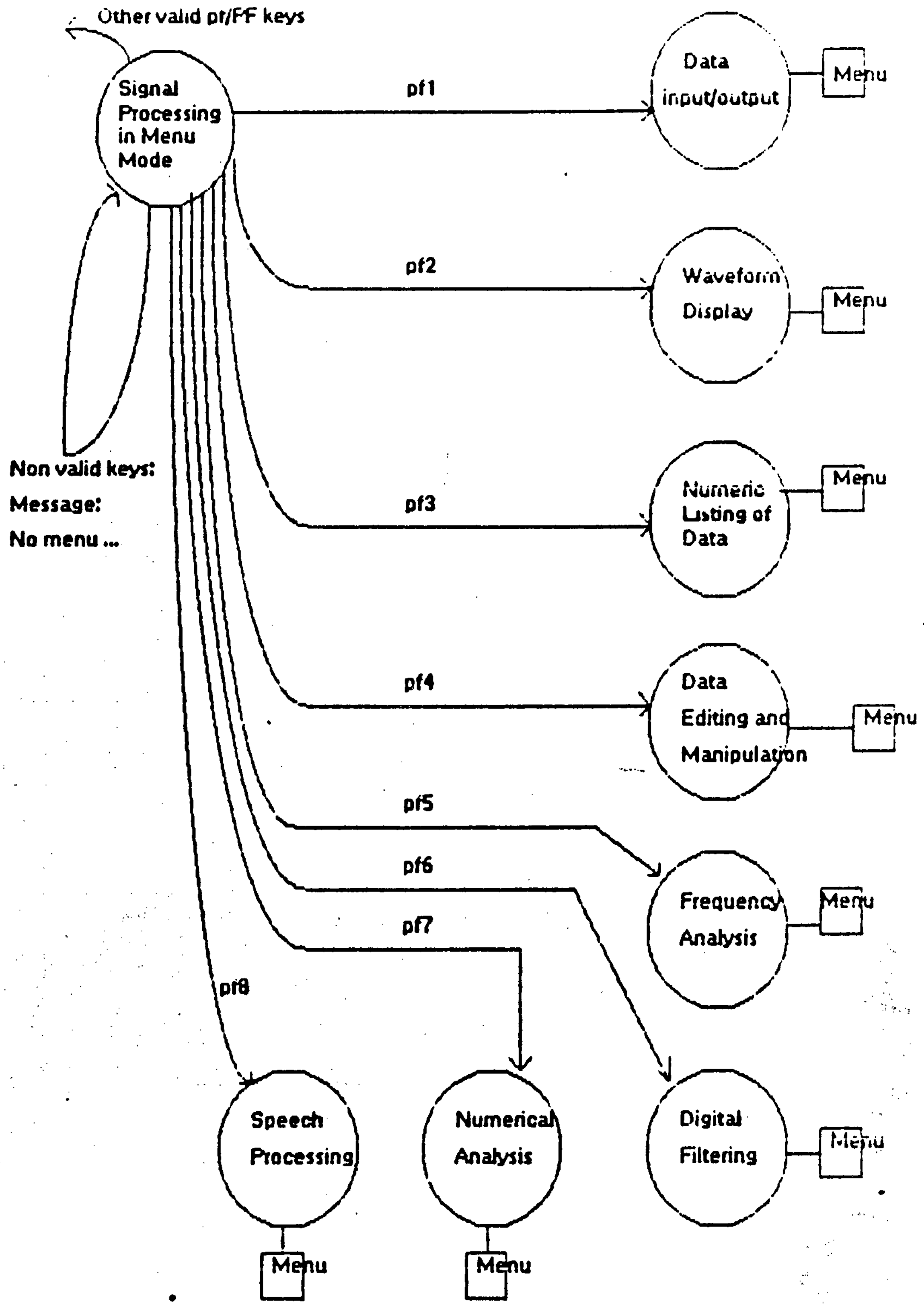
The two diagrams are taken from his dissertation. He later went on to design a much more friendly system using our techniques. The third diagram illustrates a small part of this.

Audlab

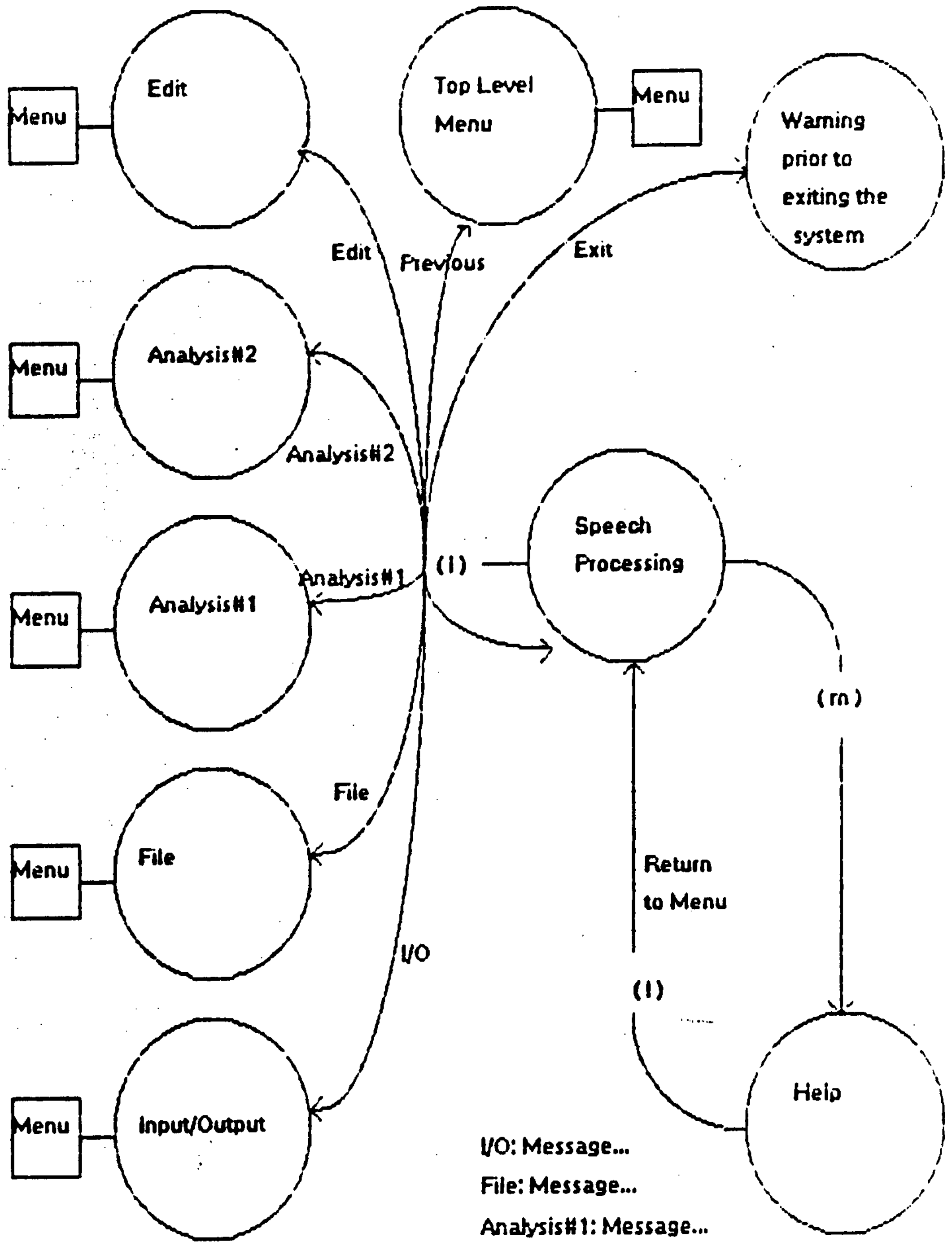

Start State
Unix O/S
Process Window
(l,r)
Waveform Window (title bar)
Window Manager
Menu
Waveform Window (display)
Window Options
Menu
(m)
Audlab Top Level
Menu
Help
Process window: message...
Prompt window: message...
Title bar (waveform): message...
Display (waveform): message...

Audlab Menu System: level 1: Start State

ILS



Signal Processing Menu State



I/O: Message...
File: Message...
Analysis#1: Message...
Analysis#2: Message...
Edit: Message...
Previous: Message...
Exit: Message...

Speech Processing Screen State

You may notice that the diagrams we are now dealing with are no longer straight finite state machines. The arrows are often labelled by functions that are defined for various data types. These are X-machines and they can easily be formally defined in a complete way incorporating the data type specifications.

The use of X-machines for the specification of user interfaces also allows us to try and define some very important and difficult matters. One thing that is fundamental to good interface design is the identification of a user's processing *goals*. These can often be defined in terms of a set of <u>functions</u> from the users conceived input data type to the perceived output type.
The design problem for interfaces is then the construction of an X-machine that will reproduce these functions. To do this the designer will generally try to decompose the goals into a sequence of simpler *tasks* which will form the basis of a detailed design. The information given to the user is then carefully planned out to reinforce the user's conceptual model of the system and to try and make it match well with the designers. This relationship between the system and the user's model of the system must be the basis for designing *expert* interfaces which can learn about the user on the basis of the user's history of behaviour.
More in [2],[3].

2.5 Automata theoretic testing

Since some software systems can be modelled as finite state machines, or generalisations of them it is possible to use these models as a basis for system testing. We describe very briefly the work of Chow [9].

Essentially machines will describe the control structure of the system that describes how the system operations are sequenced as a response to environmental stimuli. The usual method is to assume that the machine model is complete, reachable and minimal.

The testing set is defined from the model and is based on ideas of automata equivalence, i.e. the automaton 'implied' by the implementation is tested against the automaton of the design to see if they are equivalent as machines.

Chow's method consist of

(i) estimating the size of the implied implementation machine,
(ii) constructing a suitable set of test sequences,
(iii) examining the responses when test sequences are applied.

To construct the test sequences he first constructs a test tree from the machine diagram, this describes how each state can be reached from the initial state.

The set P of input sequences is defined to be the

minimal set of sequences such that if $qi \xrightarrow{x} qj$ then $p \in P$

such that $px \in P$ and $qo \xrightarrow{p^{*}} qi$.

The next step is the construction of a 'characterization' set, this is a, W, set of input sequences that can distinguish between the behaviour of every pair of states in a minimal automaton. Then by forming

$$Z = W \cup \Sigma.W \cup \ldots\ldots \cup \Sigma^{m-n}.W$$

where m is the estimated number of states in the implementation and n is the size of the design machine we have a method for generating test sequences.

Chow proves the theorem that two automata are equivalent if f they are P.Z-equivalent. (Thus we can test equivalence of two machines by applying this small test set to the machines started in their initial states and checking the corresponding outputs.)

This type of testing has been successfully applied to graphics command languages, a real-time process control system and a telephone switching system. For these applications several subtle errors were found in the systems with fairly small numbers of test sequences (i.e. a few hundred).

Part 3 Some structure theory

3.1. Machine models of parallelism and neural processing

3.1.1. Cellular automata

This, one of the classical models of parallel systems, has been extensively studied.

Consider a set of identical finite state machines $A = (Q,\Sigma,F)$ (with no output). These are distributed in a regular fashion across a 2 or 3 dimensional grid:-

$$\begin{tikzcd}
{} \arrow[r, dash] & A_{11} \arrow[r, dash] \arrow[d, dash] & A_{12} \arrow[r, dash] \arrow[d, dash] & A_{13} \arrow[r, dash] \arrow[d, dash] & A_{14} \arrow[r, dash] \arrow[d, dash] & {} \\
{} \arrow[r, dash] & A_{21} \arrow[r, dash] \arrow[d, dash] & A_{22} \arrow[r, dash] \arrow[d, dash] & A_{23} \arrow[r, dash] \arrow[d, dash] & A_{24} \arrow[r, dash] \arrow[d, dash] & {} \\
{} \arrow[r, dash] & A_{31} \arrow[r, dash] \arrow[d, dash] & A_{32} \arrow[r, dash] \arrow[d, dash] & A_{33} \arrow[r, dash] \arrow[d, dash] & A_{34} \arrow[r, dash] \arrow[d, dash] & {} \\
& \vdots & \vdots & \vdots & \vdots &
\end{tikzcd}$$

Each machine Aij is in some state and receives as input the states of its immediate neighbours, thus

A_{23} receives the states of A_{13}, A_{24}, A_{33} & A_{22}.

The complete network is synchronized and at the next 'clock tick' the state of A_{23} changes in accordance with the values of its neighbour states.

Thus the next state function is of the form

$F: Q \times (Q^4) \dashrightarrow Q$

for this 2-dimensional example.

There are various modifications of this model which include, for example, cellular output functions and intracellular communication alphabets (so that we can get away from using the states of neighbouring cells as inputs) etc. Some useful results have emerged from these concepts and they have led to applications in the modelling of biology, in computer graphics and in simple analysis of parallel systems and the complexity of parallel algorithms.

There are several disadvantages, however, in using cellular automata as a vehicle for reasoning about neural networks. The physical arrangements in neural networks in animals is rather different and some of the tasks that animals can perform seem to beyond the capabilities of cellular automata.

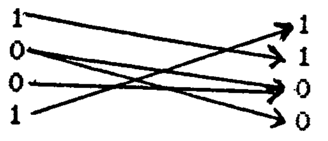
Some useful papers are to be found in [14] & [15].

3.1.2. Bus automata

These were introduced by Rothstein [**] and have been shown to be very powerful models.

To define a bus automaton we first introduce the concept of a <u>conduction function</u> (or C-function).

Consider a directed bipartite graph with k input and k output nodes eg.



Edges with origin 1 also have target 1

This can be represented by a kxk Boolean matrix.

$$G = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

which describes the connections (1 for yes, 0 for no) between nodes

Then

$$G. \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$

Now let B_k be the set of kx1 binary column vectors and define functions of the form

$$G : (B_k)^m \dashrightarrow B_k \quad \text{where}$$

$$G (v_1,.,v_n) = M_1 v_1 + \ldots + M_n v_n$$

where M_i are k x k Boolean matrices. These are called C-functions of dimension n.

Let C_n be the set of all C-functions of dimension n.

Now for the definition of a bus automaton.

We choose a standard cellular automaton constructed from standard individual automata, with state sets Q. Let $a = 3^d - 1$ where d is the spatial dimension of the system.

Define $g : Q \dashrightarrow B_k$ to be the local output function of the cell.

For each $q \in Q$, define a conduction function

$$Gq : (B_k)^a \times B_k \dashrightarrow B_n$$

and finally consider an output function

$h : (B_k)^* \times Q \dashrightarrow B_k$

Such that

$h(x,q) = G_q(\ x, g\ _{(}q_{)})$ for $x \in (B_k)^*$, $q \in Q$.

The vector x represents output of neighbours to a particular cell, g(q) represents signals originating at the cell and G_q is the transformation applied to all of these signals to produce the total cell output. Thus G_q will define conduction channels between cells dependent on the states of the surrounding cells.

Essentially if the array of input cells receives inputs while each are in specific states, the conduction function and the next state functions determine the resultant next state and the channels through which the state information can be sent. Thus bus automata are essentially cellular automata with the extra dimension of the ability to use communication channels to directly communicate with distant cells controlled by a local switching network. It is possible for cells to be sources, sinks or links for communications purposes as well as information processors.

The recognition power of such systems is quite impressive, here we have an initial state specified at one boundary of the system which receives inputs and consider a computation to have resulted in the recognition of the input string if a given cell or cells enters certain prescribed final states.

<u>Immediate</u> languages are languages L such that a fixed constant K exists which acts as an upper bound for the number of steps required to recognize any member of L.

Some results of Rothstein and workers.

3.1.2 (i) Regular languages are immediate with respect to one-dimensional bus automata.

3.1.2 (ii) Parikh languages (e.g. $L = \{a^p : p$ is prime$\}$) are immediate bus automata languages.

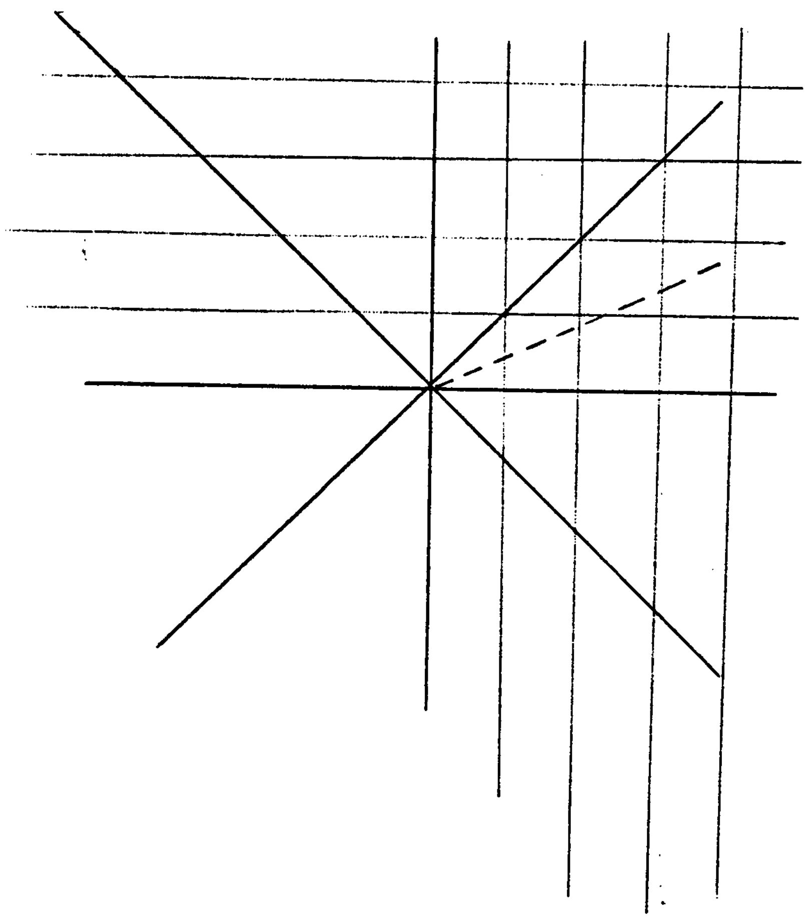
3.1.2 (iii) Recursive languages exist which are not immediate with respect to polynomial propagation time.

See [10]

<u>3.2 Applications to vision</u>

The origins of bus automata are to be found in questions concerned with the recognition of geometric objects and patterns by cellular arrays of machines, these machines correspond to the neural-retinal system in animals. The 'outer layer' of cells will receive 'on-off' input symbols when we consider light rays from a 2-dimensional monochrome collection of straight lines or other simple geometric objects. The effect of these inputs on the whole cellular array must be to generate various state changes and communications that essentially involve the 'recognition' of these patterns in a machine theoretic way.

Consider the problem of recognizing straight lines through some central axial point.



Divide the plane into 8 octants and superimpose a 2-dimensional grid of cells upon the diagram. A line through the origin in an octant can be coded up into a binary string according to the rule:-

if the line crosses opposite sides of a cell we write a 0

if the line crosses adjacent sides of a cell we write a 1.

The line indicated encodes to 0011...

Thus the recognition of lines can be encoded into problems of recognizing and generating binary strings. To do this effectively the idea of a bus automaton has been developed.

Other types of geometric objects can be quickly recognized by bus automata. e.g. parabolas etc. See [11].

3.3 Minimization of machines

One basic process that can make the analysis or implementation of a machine easier is the reduction of the state set of a machine by removing any redundant states. We cannot do this simply by deletion without disturbing the behaviour.

Consider an X-machine

$$\mathcal{M} = (X, \Phi, Q, F, Y, Z, \alpha, \beta, I, T)$$

and let

$$\blacksquare = (Q, \Phi, F)$$

We call $\blacksquare$ a <u>general</u> <u>machine</u> and we will now indicate how it can be minimized. Thses general machines contain all of the essential information required for the reduction of the state space of the machine.

Let a, b be states of Q we define ;

$\mathcal{L}(a,b)$ to be the set of all path labels in $\blacksquare$ from a to b,

$\mathcal{L}^-(a)$ the set of all path labels into a

and

$\mathcal{L}^+(a)$ the set of all path labels leaving a.

A relation $\equiv$ is defined on Q as follows :-

for $a,b \in Q$, $a \equiv b$ iff (i) $\mathcal{L}(a,b) = \mathcal{L}(b,a)$

and (ii) either $\mathcal{L}^-(a) = \mathcal{L}^-(b)$;

or $\mathcal{L}^+(a) = \mathcal{L}^+(b)$;

or $\mathcal{L}^-(a) \subseteq \mathcal{L}^-(b)$ &

$\mathcal{L}^+(a) \subseteq \mathcal{L}^+(b)$;

or $\quad L^{-}(a) \subseteq L^{-}(b)$ &
$L^{+}(a) \subseteq L^{+}(b)$.

The transitive closure, ≈, of this relation is then the basis for the minimization process. We factor ▪ out with respect to ≈ and continue this process until we find the situation stabilizes. The stable general machine is then minimal with respect to the behaviour of the original machine. That is the functionality of the minimal machine and of the original machine are the same. See [12]

3.4 Decomposition Theories

Most decomposition theories involce the replacement of a machine by another machine with the same (or greater) processing functionality but which is constructed from 'simpler' machines connected together in standard ways.

3.4.1 We can ignore outputs using some results from automata theory.

Let us consider the finite state machine case. If $U=(Q,\Sigma,H,F,G)$ and $U^1=(Q^1,\Sigma^1,H^1,F^1,G^1)$ are state machines then we define the concept of U <u>covering</u> U^1 as follows:-

Let $a:\Sigma\to\Sigma^1$, $b:H\to H^1$ be functions and suppose that $\emptyset:Q\to Q^1$ exists such that

whenever $x\in\Sigma^*$ and $q\in Q$ with x applicable to state q (i.e. the machine can completely process x) and a(x) is applicable to state $\emptyset(q)$ then the outputs from the two machines do not differ when specified.

It is possible to show that if U is any machine and we consider the underlying automation without outputs, u, then if $u\le u'$ for some other output free automation then we can define outputs on u' to produce a machine U' such that U' covers U.

Thus we need only consider the decomposition of output-free automata.

Two of these are related under the relation ≤ mentioned above if a function $a:\Sigma\to\Sigma'$ exists together with a negative partial function $b:Q'\to Q$ such that

$$b(q')F_x \subseteq b(q'F'_{a(x)}) \text{ for } q'\in Q',\ x\in\Sigma^*$$

We can now replace each output-free machine (Q,Σ,F) by a transformation semigroup (Q,S)
where S is the semigroup generated by the next-state functions in Σ^*.
Then we can use the Holonomy decomposition theorem
(Eilenberg, improved by Holcombe) to obtain:-

$$(Q,S) \le A_1 \circ A_2 \circ \ldots \circ A_n$$

where each A_i are products of holonomy transformation semigroups generated by a height function defined by (Q,S).
Each A_i is of a particularly simple form and can be generated by finite simple groups and aperiodic semigroups.

The product $\circ$ of two transformation semigroups is defined in a standard algebraic way,

e.g. $(Q,S)\circ(Q',S') = (Q\times Q', S^{Q'}\times S')$

where S^{Q^1} is the set of all mappings from Q^1 to S, the action of $S^{Q^1}\times S^1$ on $Q\times Q^1$ is defined canonically that is

$(q,q')(f,s') = (qf(q'),q's')$

where

$q\in Q$, $q'\in Q'$, $f\in S^{Q'}$, $s'\in S'$.

Thus we can infer that for finite state machines, each can be generated from simple machines connected together using parallel and sequential connections. See [13] with more on products in [14].

3.5 A design methodology based on X-machines.

The initial model or specification can be developed in terms of a specification of a basic data type and functions in *Z* together with a machine description of processing and control states which allow us to define a simple X-machine model of the simplified system. This model can then be extended and refined to provide a series of more detailed and powerful designs again described in terkms of X-machines. These models are successively verified and tested at each stage. Eventually we obtain a series of validated designs culminating in the ultimate detailed design which can then be implemented directly according to the circumstances.

Such methods have been discussed in a variety of places for example Milne [5]

The types of development steps that are appropriate for an X-machine development method include the following.

Extension : The state set Q of the X-machine is enlarged and extra labelled arrows inserted in the state space. The next state function is extended in this process.

Refinement : The state set remains unchanged but more labelled arrows are inserted in the state space.

Generalization : Here the data type X is changed into a semantically related data type X' which is then the basis for an X'-machine which is generated from the original X-machine. The input and output relations α and β will usually have to be adjusted.

Enlargement : This involves keeping both X and Q the same but increasing the number of labelled arrows in the state space. This will also increase the behaviour of the machine since more paths will be available for processing.

Other controlled developments of X-machines can be considered including, of course, combinations of the above. Some of the examples considered earlier demonstrate these processes, eg the lexical analyzer example and the user interface example.

The use of generalizations of the verification and testing methods developed for state machines will, hopefully, extend to X-machines as the minimization procedure did. The decomposition of X-machines into products of simpler machines is more problematical, however, although there may be a prospect of some useful approaches to modelling parallelism using products of machines.

References.

[1] M.Holcombe, "X-machines as a basis for dynamic system specification." Software Engineering Journal March 1988 69-76.

[2] " "Formal methods in the specification of the human-machine interface." Int. CIS Journal. 1(1).1987.24-34.

[3] " "Goal-directed task analysis and formal interface specifications." Int. CIS Journal. 1(4).1987.14-22.

[4] S.Eilenberg. "*Automata, languages and machines*". Academic Press, 1974.

[5] G.Milne. "Towards verifiably correct VLSI design." in Formal aspects of VLSI design. North-Holland 1986.

[6] W.A.Wulf, M.Shaw, P.N.Hilfinger & L.Flon. "*Fundamental structures of computer Science.*" Addison-Wesley. 1981.

[7] C.H.West & P.Zafiropulo. "Automated validation of a communications protocol.." IBM Jour.Res.& Dev. 22, 1978, 60-71.

[8] M.R.Combs. "An analysis of the menu-driven human-computer interfaces of two digital speech processing packages." M.Sc. dissertation. University of Sheffield. 1988.

[9]T.S.Chow. "Testing software design modeled by finite-state machines." IEEE Trans. Soft. Eng. SE-4 1978, 178-187.

[10] J.M.Moshell & J.Rothstein. "Bus automata and immediate languages." Inf. & Control. 40.1979,88-121.

[11] J.Rothstein & A.Davis. "Parallel recognition of parabolic and conic patterns by bus automata." Proc.Int. Conf. Parallel Processing 1979 (IEEE 79CH1433-2C) 288-297.

[12] M.Holcombe & M.Stannett. "General machines." Dept. Report CS-87-2 University of Sheffield.

[13] M.Holcombe."*Algebraic automata theory.*" CUP. 1982.

[14] F.;Gecseg. "*Products od automata.*" EATCS Mono. 7,1986.

[15] C.Choffrut(ed.) "*Automata networks.*" LNCS 316. 1988.

[16] T.Legendi(ed.) "*Parallel processing by cellular automata and arrays.*" North-Holland 1987.