AN ASSOCIATIVE AND
IMPRESSIBLE COMPUTER

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Doctor of Philosophy
in the
University of Canterbury

by
John G Cleary

University of Canterbury
1980
<table>
<thead>
<tr>
<th>CONTENTS</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>1</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>3</td>
</tr>
</tbody>
</table>

**CHAPTER I**

**AN INTRODUCTION TO THE FLM** 8

1. THE FLM 9
   1.1. Storage 10
   1.2. Retrieving 12
   1.3. Computing 15
   1.4. Creation and Alteration of Programs 17

2. AN EVALUATION OF THE FLM 20
   2.1. Production Systems 21
   2.2. Programming Power 22
   2.3. Ease of Programming 24
   2.4. Integrated System 26

3. EFFICIENCY OF IMPLEMENTATION 27
   3.1. Speed and Memory Space 27
   3.2. Forgetting 28

4. USES OF PRODUCTION SYSTEMS 29
   4.1. Human Problem Solving 30
   4.2. Learning of Heuristics 30

APPENDIX 32

A.1. FORMAL DESCRIPTION OF FLM 32
   A.1.1. Buffer-update 32
   A.1.2. Update 33

A.2. PREDICTION 33

A.3. FLM COMPUTER 33

**CHAPTER II**

**PROGRAMMING THE FLM** 35

1. THE POWER OF THE FLM COMPUTER 35
   1.1. Disjunction 36
   1.2. Conjunction 39
   1.3. Negation 42
   1.4. Subroutines 43
   1.5. Summary 47

2. COMPLEXITY 47

3. FINITE AUTOMATA 49
   3.1. Simulation of Automata 49
   3.2. Programming Automata 53
   3.3. Efficiency of Programming 55
   3.4. Psychological Speculation 58

4. EXECUTION TIME FOR AUTOMATON SYNTHESIS 59

5. REPRESENTING THE FLM 66

6. SIDE EFFECTS OF PROGRAMMING 67
   6.1. Program Mode 67
   6.2. Deleting Traces 70

7. SUMMARY 71

APPENDIX 73

A.1. Invariance of Memory 73
A.2. Stability in Run-mode 74
   A.2.1. Invariance of Deterministic Predictions 75
# Table of Contents

3. NETWORKS ............................................. 164
   3.1. Network Representation ............................ 165
   3.2. Nets Over Trees .................................. 168
   3.3. Update Algorithm .................................. 168
   3.4. Deletion .......................................... 170

4. FELIX .................................................. 171
   4.1. Memory Structure .................................. 171
      4.1.1. Event Tree .................................... 174
      4.1.2. Branches ...................................... 175
      4.1.3. Leaves and Nets ................................ 176
   4.2. Storage Usage ..................................... 177
   4.3. Execution Times .................................... 178
      4.3.1. Updating ....................................... 179
      4.3.2. Forgetting ..................................... 180
      4.3.3. Prediction ..................................... 181

5. HASHING ............................................... 182
   5.1. The Basic Algorithm ............................... 182
   5.2. Best Match of Traces ............................... 185
   5.3. Variable Length Hash Keys ........................ 186
   5.4. Storage Requirements .............................. 188
   5.5. Execution Time .................................... 189
   5.6. Calculation of t(T) ............................... 190
   5.7. Deletion and Forgetting ......................... 193

6. AN ANALYSIS OF MEW-PUSS .............................. 195
   6.1. The Mew-PUSS Algorithm ......................... 196
   6.2. Expected Error .................................... 196
   6.3. Storing Events More Than Once ................. 197
      6.3.1. Algorithms ..................................... 198
      6.3.2. Expected Error ................................ 198
   6.4. Comparison of Expected Error ................... 200

7. SIMPLE ................................................ 202
   7.1. The Simple Algorithm ............................ 202
      7.1.1. Update ....................................... 205
      7.1.2. Bufferupdate .................................. 207
      7.1.3. Getspace ...................................... 207
      7.1.4. Predict ........................................ 208
   7.2. Hardware Implementation of Simple ............. 209
      7.2.1. SPM ........................................... 209
      7.2.2. Instructions .................................. 211
      7.2.3. Notation ...................................... 212
      7.2.4. Simple Routines Using the SPM .............. 213
      7.2.5. Execution Time ................................ 214
      7.2.6. Storage ....................................... 214

8. COMPARISON OF FLM IMPLEMENTATIONS .................. 221
   8.1. Storage Usage .................................... 223
   8.2. Execution Times ................................... 227
      8.2.1. Central Store Resident ...................... 227
      8.2.2. Disc Resident .................................. 228
   8.3. Conclusion ....................................... 228
   8.4. Example Memories ................................. 229

9. NOTATION .............................................. 231
APPENDIX. ................................................................. 232
  1. Forgetting Results ............................................. 232
      1.1. Two Population Model ................................... 232
          1.1.1. Ladder .............................................. 232
          1.1.2. Random Forgetting ................................. 236
          1.1.3. Best Case ......................................... 237
      1.2. Cyclic Model ............................................ 238
          1.2.1. Random Forgetting ................................ 238
          1.2.2. Best Case ......................................... 238
  2. Memory Usage in Simple ....................................... 239
      2.1. All Action Inputs ....................................... 239
      2.2. Alternating Actions and Patterns ..................... 244
          2.2.1. N Odd ............................................. 244
          2.2.2. N Even ............................................ 245
  3. LISTING OF B6700 VERSION FELIX ............................. 248
      3.1. Memory Structure as Defined by Macros ............ 248
      3.2. Memory Access Routines ............................... 250
      3.3. Setfree ............................................... 251
      3.4. Get .................................................. 251
      3.5. ENSURE ............................................... 252
      3.6. Initialize ............................................ 256
      3.7. Update ............................................... 257
      3.8. Predict ............................................... 262

CHAPTER V
BEST MATCHING ALTERNATIVES .................................... 264

CHAPTER VI
RELAXATION ALGORITHM FOR EVALUATING NETWORKS ............... 266
  1. STATEMENT OF PROBLEM ...................................... 268
      1.1. Preliminary Definitions ............................... 271
          1.1.1. Equation Set .................................... 271
          1.1.2. Evaluation ...................................... 271
          1.1.3. Consistent ...................................... 271
      1.2. Solution ............................................. 272
  2. PRELIMINARY RESULTS ....................................... 274
      2.1. Definitions .......................................... 274
          2.1.1. D ................................................. 274
          2.1.2. l ................................................. 274
          2.1.3. G ................................................. 274
          2.1.4. S* ............................................... 275
          2.1.5. Cyc ............................................... 275
      2.2. Examples from Definitions ............................ 275
      2.3. Semi-Reflexive Sets .................................. 277
  3. EVALUATION MAPS ............................................. 281
  4. CASCADES .................................................. 284
      4.1. Definition : Cascade, Length of Cascade ............ 285
      4.2. Construction of Cascades ............................. 287
  5. RELAXATION ALGORITHM ...................................... 290
      5.1. Algorithm ............................................ 290
      5.2. Theorem - Algorithm Terminates Correctly .......... 291
  6. ALTERNATIVE EVALUATION MAPS ................................ 294
  7. CONVERSE PROBLEM ........................................... 296
8. EXAMPLES ........................................ 296
   8.1. Cascades of Length 1 ................. 296
   8.2. Further Examples of Cascades ....... 300

APPENDIX. ......................................... 303
   A.1. Example of Operation of Relaxation Algorithm .... 303

CHAPTER VII
CONCLUSION AND FURTHER RESEARCH .......... 306
  1. IMPLEMENTATION OF A.I. SYSTEMS ......... 306
  2. STABILITY .................................... 307
  3. PROGRAMMING ................................. 309
  4. IMPLEMENTATION .............................. 310
  5. FORGETTING ................................. 311
  6. NETWORKS .................................... 312
  7. IMPORTANCE OF THE FLM .................... 313

ACKNOWLEDGEMENTS ............................... 315

REFERENCES ....................................... 316
ABSTRACT

A new process for storing strings of objects on a best match basis is defined. This process is used to construct a simple computer system and more complex systems compounded from this. These systems are compared with some models of psychological processes. It is shown that some elementary programming constructs can be duplicated by these systems, that others are impossible, and that even the simplest system can simulate any given Turing machine.

Particular consideration is given to the fact that the memory of the system is continually changing and being updated. In an analysis which has not been undertaken elsewhere it is shown that a fixed algorithm can be encoded even within this changing memory. This result is particularly important as it strongly constrains the possible structures for the multiple systems. General principles underlying both natural and artificial adaptive systems are indicated by these results.

A number of ways of implementing the system on both serial and parallel digital computers are compared in terms of storage efficiency and execution times. An actual implementation is exhibited and used to support these theoretical estimates. It is concluded that execution times almost independent of memory size can be obtained in practice.

An alternative best matching algorithm which finds nearest neighbours in Euclidean space is examined. A new analysis of this algorithm is given which shows that its execution time is independent of the number of points being matched but that it increases exponentially with the dimension of the Euclidean space.

A new algorithm is presented for solving some systems of
simultaneous equations over finite domains. This algorithm can be used for extracting additional information from a best match memory. However, it is cast in a very general form and should be applicable elsewhere.
INTRODUCTION

Both the brain and a digital computer are capable of performing large amounts of computation. However, they acquire their behaviour in strikingly different ways. A digital computer requires at some stage a complex and carefully prepared input in some computer language in order to be programmed. In contrast, the human and some animal brains attain very complex behaviour, for example language, as the result of a heterogeneous mixture of experience, teaching and self motivated behaviour such as exploration and imitation.

This dichotomy has generated two inter-related lines of endeavour amongst psychologists and computer scientists. One is to achieve an understanding of the common phenomena underlying both the brain and computers; the other is to program digital computers so that they can emulate the abilities of the brain.

So far, however, the application of computing concepts to the brain has been almost totally unproductive, see for example the discussion in (Crick, 1979). Traditional computing concepts such as computer languages and compilers do not seem to have any identifiable analogies in the brain.

The efforts to construct an 'artificial intelligence' have relied on ever more complex and carefully prepared programs. While useful in themselves, these programs shed no light on how the brain achieves its abilities without such strict programming.

This thesis proposes a computing system which is simple enough to be theoretically understood. Because it operates in a way which is similar to the brain it may be able to shed light on its workings. All
sensory inputs to the brain seem to be able to modify its later behaviour. Similarly, the computer system to be described has a memory which is being continually updated and modified by its inputs (hence the 'impressible' in the title of this thesis). Also, the behaviour of the brain is at all times readily influenced by external patterns. Similarly, the computer system decides what is to be done next by using the pattern of its recent inputs and actions to recall information from its memory (hence the 'associative' in the title of this thesis). The computer system is referred to as the FLM from Finite context Learning Machine which refers to the fixed length pattern or context used to update and retrieve information from the computer's memory. The major part of this thesis is concerned with obtaining a theoretical understanding of the FLM and with techniques for implementing it.

The underlying associative mechanism of the FLM was first proposed in 1972 and implemented in the form of an early version of PURR-PUSS. Results from this system have been published in (Andreae and Cleary,1976). It was realized later that this associative mechanism could be implemented so that its execution time was independent of memory size. The Felix system, which includes this fast algorithm, was proposed in (Cleary,1973) and a later version is described fully in Chapter IV.

The form of the FLM used in this thesis arose from the need for a very simple and well-defined system which could be used to make precise the results in Chapters II and III on how the FLM can be programmed. These results arose from practical experience with PURR-PUSS and Felix during the period 1972 to 1975.

One important result is that it is possible to describe the future behaviour of the FLM very closely even though the contents of
its memory are continually being modified. This result is particularly important, for without it, it would be impossible to say that a particular piece of behaviour (i.e., an algorithm) had been permanently acquired; for it might later be modified in such a way that the behaviour could not be reproduced. It also assumes importance in Chapter III where multiple systems compounded of more than one FLM are described. It is shown that the structure of such a system is severely constrained if the system's future behaviour is to be sufficiently well described.

If one places no importance on how long a calculation takes, then it is known that any possible calculation can be accomplished by a very simple computer called a Turing machine. This may take a very long time but eventually it will complete any calculation which can possibly be completed. It is shown in Chapter II that an FLM shares this ability by showing that it can do everything that a Turing machine can do.

Digital computers achieve some of their speed and economy of storage by being able to use programming techniques such as subroutines. A subroutine is a single piece of program which is written once and which can be used in many different places without having to be written out again in full. It is shown that the FLM can effectively make use of subroutines and a number of similar programming techniques. However, it is also shown that the simple FLM and any multiple FLM systems can only execute an action when something positively happens. That is, they are unable to execute an action when a condition is absent without specifically listing all the other conditions which may occur.

If the FLM is to be used to construct an artificial intelligence system or if it is to be used as part of a simulation, say of a
psychological model, then it is important that it should run quickly and use a minimum of storage. Chapter IV considers a number of possible implementations of the FLM both for serial digital computers and, in one instance, for parallel hardware. It is shown how the FLM memory can be arranged using a tree structure and using hash coding techniques. A new algorithm is described for constructing variable length hash keys. This allows some saving of space when used to construct an FLM memory. A comparison is made of these implementations in terms of their storage usage and execution times. These comparisons draw on a number of theoretical results and on experience with an actual implementation of the FLM. It is shown that very large memories holding millions of 'program steps' are practical and that the time to store and retrieve information is nearly independent of memory size.

The best match or associative scheme which is introduced in Chapter I is not the only possible matching algorithm which might be used. However, few theoretical results are available about the speed of other algorithms. Chapter V provides a complete analysis of an algorithm which uses as its measure of similarity the distance between two points in Euclidean space. (Normal space is a three dimensional Euclidean space and a flat plane is a two dimensional Euclidean space). The analysis shows the algorithm to be the most efficient yet proposed for this form of matching. It also provides a lower bound on the execution time of the algorithm which is independent of the number of points being matched, but which increases exponentially with the dimension of the space. The contents of Chapter V were completed in 1975 and have been published in (Cleary, 1979).

The memory of the FLM potentially holds more information than is actually used by the best match retrieval of the FLM. In particular
predictions a number of steps into the future can be made. An attempt to set up a number of consistent schemes to do this ran into difficulties when it was found that they required the solution of a large number of simultaneous equations over variables which could take on only a small number of discrete values. Chapter VI presents a solution to this problem. An algorithm is given which works for certain classes of problems where it is known that all possible simultaneous equations must have a solution. It is cast in a very general form and should be applicable elsewhere.
CHAPTER I
AN INTRODUCTION TO THE FLM

In this and the following two chapters I will propose an architecture for a simple computer based on an associative memory. A novel feature of the computer is that the 'programs' in its memory are continually being updated while it is operating. I will place particular emphasis on the use of this computer to achieve a theoretical understanding of such adaptive systems and on efficient implementations of it using serial digital computers.

It will be seen that the computing system, referred to from now on as the FLM(Finite context Learning Machine), is similar to the stimulus response(SR) mechanism described in (Suppes, 1969a) and the production systems of (Newell and Simon, 1972). SR theory is an explanation of learning in the human brain at an elementary level. Newell and Simon's production systems provide a description of human problem solving of the type required to play chess, for example. This thesis, however, is not intended to have direct psychological content except for the possibilities inherent in a novel computing system.

The FLM is also similar to the 'teachable' system PURR-PUSS, (Andreae and Cleary, 1976) and (Andreae, 1977). The aims for the design of PURR-PUSS are to provide an open system capable of being taught and of acting in the world. The FLM has been introduced to provide a more theoretically founded basis for the development of systems like PURR-PUSS. It has been made as simple and theoretically 'clean' as possible. Many practical details such as the interaction with the human operator have been simplified in the interests of clarity.

The FLM also forms an example of a 'length-k model' as discussed
in (Witten, 1979). In the terminology of this article the FLM is just a record of the 'length-k vocabulary' of its inputs. The FLM, however, makes no attempt to reduce the number of states in the model as Witten does.

I.1. THE FLM

On a serial digital computer the basic cycle of operation is: fetch an instruction, execute the instruction, increment the program counter and repeat the cycle. Apart from specifically programmed changes to the program counter (jumps), instructions are fetched in a strict linear sequence.

By contrast in the FLM computer, each 'instruction' is fetched associatively from a memory using as a key a buffer which holds a list of the most recent instructions executed by the computer and the results of their execution. Also on each cycle a copy of the buffer is stored into the memory and may be associatively recalled again in the future. In this way the FLM is continually being updated and modified by its inputs.

The rest of this section describes the FLM and, together with the Appendix at the end of this chapter, makes precise its operation. Three algorithms will be described; the first stores information into the memory of the FLM, the second retrieves information from this memory on a 'best fit' basis and the third combines these two to form a primitive computing system.
I.1.1. Storage

The FLM stores its 'programs' and other information in its memory which is a set of traces. Each trace is a sequence of $N + 1$ events, consisting of a tail of $N$ events and a single event which is its head. $N$ is finite and fixed for a particular FLM and is referred to as the order of the FLM. The events of which the traces are composed are simple discrete values with no internal structure available to the FLM. (It may help some readers to note that the stimuli and responses of SR theory will both be treated as events.)

An FLM 'sees' one event at a time in a strict sequence. Each time an event is seen, a trace may be stored in the memory. The trace, if stored, consists of the last $N + 1$ events seen, with the most recently seen event forming the head. When information is to be retrieved from the memory the last $N$ events seen are matched against the tails of the traces in the memory. The heads of the best matching traces then form a prediction of the next event to be seen.

This storage process is illustrated in Fig.1. There, the events seen are the letters of the alphabet. As each letter(event) is seen it is pushed onto the head of a special trace — the buffer trace. The oldest event in this buffer trace is lost and the whole trace is placed in the memory. The buffer trace also provides the context used by the prediction algorithm to fetch information from the memory.

As the FLM has only a finite memory, then some exceptional action must be taken when the memory becomes full if it is to continue running. Chapter IV evaluates techniques for deleting traces from memory to allow addition of new information. This process will be referred to as forgetting.
most recent event

...A B C D E F G

A B C D
B C D E
C D E F

D E F G

| MEMORY |

head

Sequence of events seen by FLM.

Successive traces stored in memory.

Most recent trace held as buffer trace.

There are 3 events held in the tail of the buffer. This is the order of the FLM.

Fig. 1 Storing Traces in an FLM.
1.1.2. Retrieving

Consider how the information in an FLM memory might be retrieved. The simplest technique might be to search the memory for traces whose tails are equal to the tail of the buffer trace. The heads of these traces would then form a prediction of the next event to be seen. It is possible to do this very efficiently using hash coding techniques. (The mew-PUSS of (Andreae, 1977) is an FLM like system which uses this approach). However if the current buffer does not equal the tail of any trace in the memory then no prediction will be made. The FLM uses a best match algorithm which allows a prediction to be made even when there is no exact match in the memory. It will be shown in Chapter IV that this prediction process can be implemented on a serial digital computer so that its execution time is small and independent of the number of traces in the memory.

The measure of similarity used in making best match predictions is obtained as follows:

1. Consider the first two traces which are to be matched;
2. Align them side by side, the first event of one trace aligned with the first event of the other and so on to the N'th event;
3. Examine the two tails proceeding from the first event to the N'th event;
4. Then the matching length is the number of events examined before the corresponding events are unequal.
Example:

<table>
<thead>
<tr>
<th>Tail</th>
<th>Head</th>
<th>Tail</th>
<th>Head</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B C D X</td>
<td>A B C D X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A Y C D Z</td>
<td>A B C Y Z</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

matching length = 2

The events predicted by an FLM are the heads of those traces in the memory which match the buffer trace over the maximal length. So if there are traces in the memory matching over lengths 1, 2 and 3 only the heads of traces matching over a length of 3 will be predicted.

Fig.2. gives an example of the operation of the prediction algorithm.

A special case arises when the buffer trace matches no traces in the memory. It is convenient for our later analysis if the prediction is then considered to be 'any possible event'.
Traces in memory

Z B C D
Y B C U
Q R C S

Buffer trace

A B C

Z B C D and Y B C U match over length 2.
Q R C S matches over length 1.
So the events predicted are \{D, U\}.

Fig. 2 Best Match Prediction from an FLM.

Fig. 3 FLM Computer
I.1.3. Computing

The storage and prediction algorithms for the FLM have been introduced and it will now be shown how they can be combined into a simple computing system. To do this, two classes of events, (i) patterns and (ii) actions, will be distinguished. (These correspond respectively to the notions of stimulus and response. It will be noted that the distinction is in the way these events are used rather than any distinction involving storage and retrieval.)

The actions are predicted and chosen by the FLM. If more than one prediction is available then an external agency, perhaps a programmer, is needed to select an action. The patterns are received from the world external to the FLM as a consequence of the actions. The algorithm then for the FLM to act as a computer is:

Predict the set of events which will occur next.

Choose an action from this set (possibly with the help of a programmer).

This event is the next seen by the FLM and a trace will be stored. The action will also be transmitted from the FLM to the outside world.

Obtain a pattern from the outside world as the next event to be seen by the FLM, but do not store the trace predicting the pattern.

Repeat this sequence from the start.

Fig. 4 illustrates the operation of this computing system with a simple example. The external world of the FLM might be that of a very simple robot with two actions L(eft) and R(ight) which can see two patterns b(right), and d(ark). (The convention that actions are upper
case letters and patterns lower case will be used throughout the thesis). Fig. 4 shows the successive contents of the buffer as the FLM chooses actions and sees the resulting patterns. It should be noted that when the prediction of the next action is made the buffer trace has already been shifted down one position (see Appendix for details) by the storage algorithm. So the head of the buffer trace contains no meaningful event. This is indicated by a dash -. The shifting is done in this way so that the tail of the buffer contains the last N events seen.
I.1.4. **Creation and Alteration of Programs**

The mechanism described above is not complete. As yet I have not shown how the program within the FLM can be entered or altered. There are a number of ways this might be done. Some alternatives are considered in (Andreae and Cleary, 1976) and in (Andreae, 1977). A very simple mechanism is explained below.

The programmer can put the FLM into two modes, the **run mode** and the **program mode**. In the run mode the programmer is able to supply an action only if no traces match in the FLM memory. (This will happen if and only if the last pattern has never occurred previously as an input, whereupon all possible actions are predicted.) In the program mode the programmer must make a positive response each time an action is chosen. If he enters a null input then one of the predicted actions will be chosen. Otherwise he may specify the action to be chosen, which will then be executed and the appropriate trace placed in the memory.

The bottom of Fig. 4 shows a programming example in which the FLM is programmed so that whenever the sequence "bLb" occurs then the next action will be "R". This is done as follows: the programmer puts the FLM into program mode. The next time the sequence "bLb" occurs it is as part of the sequence "RbLb". Because of the match with the earlier trace ("RdLbL") the action "L" is predicted and the programmer overrides this by entering "R". The trace "RbLbR" is then stored.

The FLM is switched back to run mode and continues executing correctly, executing "R" whenever "bLb" occurs. It should be noted, however, that the program recorded in the memory does not remain static but that new traces will continue to be added to the memory, even if the programmer is not called upon to supply an action. For example, if
the sequence "LbLb" were to occur, then "R" would be predicted because of the match with the trace "RbLbR", but a new trace "LbLbR" would still be stored. I will show in the next chapter that these extra traces 'don't matter'; they only provide a redundant specification of what has already been programmed.
Order = 4  Actions = \{L, R\}  Patterns = \{b, d\}

<table>
<thead>
<tr>
<th>Actions</th>
<th>Action</th>
<th>Pattern</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>L b R d</td>
<td>{L, R}</td>
<td>L</td>
<td>Head of trace is unspecified when prediction is made.</td>
</tr>
<tr>
<td>L b R d L</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b R d L b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R d L b</td>
<td>Ev</td>
<td>L</td>
<td>No traces match in memory and programmer chooses action.</td>
</tr>
<tr>
<td>R d L b L</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d L b L d</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Program Mode

<table>
<thead>
<tr>
<th>Actions</th>
<th>Action</th>
<th>Pattern</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>R b L b</td>
<td>{L}</td>
<td>R</td>
<td>Programmer overrides prediction.</td>
</tr>
<tr>
<td>R b L b L</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Run Mode

<table>
<thead>
<tr>
<th>Actions</th>
<th>Action</th>
<th>Pattern</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>L b L b</td>
<td>{R}</td>
<td>R</td>
<td>Best match from trace above.</td>
</tr>
<tr>
<td>L b L b R</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 4 Execution by FLM Computer
I.2.  AN EVALUATION OF THE FLM

It will be shown in (II.4) that this FLM computer can simulate any finite automaton. So given a suitable external world on which its actions can operate (for example a Turing tape) then it can simulate any Turing machine and, by Church's thesis, any effective computing device. In this very formal sense the FLM is equivalent to a human being with a pencil and sufficient paper, the CPU of a Von Neumann computer given a random access memory, or to a programming language such as FORTRAN. This shows, however, how limited this criterion of effective computability is; it would be a strange person who could tell no difference between his best friend and a FORTRAN program! Therefore, I must examine other properties of the FLM if I am to show that it offers an alternative to traditional computing systems.

I will demonstrate various programming techniques for the FLM and a more powerful FLM computer. I will argue that these techniques and actual programming experience reported elsewhere (Andreea, 1977) show that the FLM offers substantial and concise programming power. I will point out that the FLM provides a very simple interface for the programmer. It will be shown (Chapter II) that the programs in the FLM memory can be updated and modified so that there is no interference with other programs already present in the memory. I will argue that these features make the FLM particularly suitable for use in artificial intelligence applications where the inputs to the FLM do not come from a human programmer but from some automatic process. I will point to another FLM like system, PURR-PUSS (described in (Andreea and Cleary, 1976) and (Andreea, 1977)), in support of this argument.

Finally it will be shown (Chapter IV) that the FLM can be
implemented so that its execution times are nearly independent of the size of the memory and are very short compared with say human response times. The memory requirements are such that systems with millions of traces are practical with current (1979) mini-computer technology.

These arguments are summarized in the following sections and supported in detail elsewhere in the thesis.

I.2.1. Production Systems

The contents of an FLM memory are statements in a computer language (i.e. the 'FLM Language'), with the FLM computer being a mechanism for altering and interpreting statements in this language. Such a view invites comparison between the 'FLM language' and other similar computer languages. The FLM is a member of a class of languages known as 'production systems'. A production system consists of a set of productions; each production consists of a condition part and an action part. Whenever the condition part of a production is true then its corresponding action is executed. (This must be modified slightly if it is possible that more than one condition is true. It is usual then to order the productions so that the first true one is 'actioned'). A single production will be written in the form:

\[ \text{condition} \rightarrow \text{action} . \]

In the FLM each trace corresponds to a production. The condition part is supplied by the tail of the trace and the action part by the head or prediction of the trace.

Production systems have been used in both artificial intelligence and psychology (see section 4). In each discipline there have been particular reasons for their use but common to both are the
following:

(i) a production system is very simple;
(ii) it is possible to encode an arbitrarily complex calculation using a production system.

Because of the simplicity of production systems workers in artificial intelligence have found it very easy to write software to create, manipulate and interpret them. It is also more plausible to a psychologist to imagine such a simple mechanism occurring in the brain and to test its correctness.

I.2.2. **Programming Power**

While it is important that the FLM can execute any calculation, it is possible that the programs for some calculations are stilted and artificially complex. For example, few people would seriously advocate using a Turing machine as a practical computer. It is important, then, to look in more detail at what can be done concisely and easily using the FLM; that is, to see how powerful it is as a programming language.

There is a difficulty in doing this because the individual actions of the FLM might be very powerful. For example, a single action might calculate the square root of a number if the FLM were connected to a calculator in its external world. So a complete evaluation of the power of the FLM is not possible without specifying what the individual actions of the FLM can do.

In what follows I will consider in particular the ability of the FLM to synthesize the actions it has available into larger units. Only the very simplest external actions will be presumed.

Chapter II considers, as examples, some facilities available in
other languages to see whether they can be duplicated in the FLM. Sections 1.1 to 1.3 consider the formation of compound conditions from simpler conditions (where 'condition' is being used in the sense that a trace is a production with a condition and action part). Three ways of synthesizing conditions are considered:

1. if condition1 or condition2 then action;
2. if condition1 and condition2 then action;
3. if not condition then action.

If two traces, both predicting the same action, are present in the FLM memory then the effect is that of an or of their conditions. If two sequences of events are concatenated in the same trace then the predicted action will be executed only if both sequences occur, effectively forming an and of the conditions. Such concatenation is restricted, however, by the finite length of the FLM buffer. It will be found that the formation of a negation is impossible without reducing it to some positive form, that is by enumerating all the possible alternative cases.

Section 1.4 of Chapter II shows that, using the FLM, fragments of programs can be programmed and then reused many times without re-programming the entire fragment each time. In one sense this is a trivial ability of the FLM as a trace will occur in the memory only once no matter how often it is used. However, this does not fully capture the idea of a subroutine or procedure as used in many languages. A crucial feature of a subroutine is that the flow of the program after completing the subroutine is dependent on the conditions before starting it, not on the internal details of the subroutine itself. This is only possible in the FLM if all paths through the subroutine are less than one trace in length, clearly limiting their
usefulness.

A technique is available to solve these problems and considerably increase the power of the FLM. This utilizes echo actions which have only one effect; to return a pattern equal to themselves! By using echo actions as markers to retain information within the buffer it is possible to extend the effective memory of the FLM back over many buffer lengths. It is by the use of echo actions that the FLM is able to simulate any finite automaton.

The power of echo actions is further enhanced in the Dual-FLM and other systems using more than one FLM. These systems, which are introduced in Chapter III, provide rather more powerful methods for forming the conjunction of conditions, subroutines and a simple form of the negation problem, although the general form of the negation problem is found to be intractable.

The reader who finds these purely theoretical studies of a computer language dry and uninteresting is invited to consult (Andreae and Cleary, 1976), (Andreae, 1977) and the series of reports in the series (Man-Machine Studies,UC-DSE/1 to UC-DSE/15). These give extensive examples of programming for the FLM-like system PURR-PUSS, including application of programming methods similar to those outlined above.

1.2.3. Ease of Programming

The only restrictions on a programmer supplying input to the FLM are that he supplies a valid action, and that it is supplied only when the FLM is ready to receive an action. This very simple 'syntax' may be contrasted with the complex syntax of current high level languages.
The minimal syntax of the FLM and the ease with which new programs can be added make it suitable for artificial intelligence applications where the programming inputs are supplied automatically; see for example (MacDonald, 1979) and II.3.

The FLM can be used to program many (but not all) finite automata merely by presenting the FLM with sequences of inputs and outputs from the automaton. The programmer in this case is very simple, consisting of the finite automaton to be simulated and the logic to transfer the inputs and outputs of the FLM as patterns and actions.

This programmer establishes the bare possibility of doing automatic programming using the FLM. A much richer experience has been obtained with PURR-PUSS, where both automatic programming and human programming are integrated into the same system. Of particular interest is the 'novelty' mechanism (Andreae, 1977, Chapter 4) which notes traces which have occurred just once. PURR-PUSS attempts to repeat such novel experiences by choosing actions so as to lead back to the same situations again. Once a novel experience has been repeated it is no longer novel. This provides a neatly self-extinguishing system whereby PURR-PUSS tends to explore and become familiar with the external world around her. This self-motivated (from PURR-PUSS's viewpoint) behaviour can later be used to good effect by the human programmer as it accumulates experience of the environment without his intervention.
I.2.4. Integrated System

In the preceding analysis of an FLM as a programmable system, I have tried to make statements which could be substantiated either by example or formal proof. However, there are less tangible properties of the FLM which make it an appropriate object to be implemented and studied. The FLM is a closely integrated system. Functions such as input/output, programming, compilation and execution which are separate and distinct in most computing systems are subsumed into a single mechanism. For example, to perform an output from the FLM it is not necessary to execute special output statements; presentation of the output as an action is sufficient.

The FLM is easily controlled either by its own internal processing or by the inputs it receives from its external environment. This point is also made about the production system in (Newell and Simon, 1972):

"A production system offers a nice balance between stimulus-bound activity and stimulus-independent activity. The production system itself is stimulus bound if by stimulus one means the contents of the dynamic working memory [buffer trace]...

If the majority of the productions executed are reactions to internally produced symbols then the system will appear not to be stimulus-bound. On the other hand, if almost all productions take as part of their condition an external symbol, then the system will appear to be very stimulus-bound."

Interactive languages such as LISP or BASIC gain much of their immediacy and usefulness by being interpretive and not requiring a
separate compilation step when altering the source program. The FLM, by making execution integral with the process of program entry makes the consequences of new programming immediately apparent.

I.3. EFFICIENCY OF IMPLEMENTATION

I.3.1. Speed and Memory Space

I have presented some arguments as to why one might want to use an FLM as a computing device. To complete this argument it is necessary to show that the FLM can be implemented efficiently on a serial digital computer. (Other hardware is possible, but is not generally available. Chapter IV proposes a parallel hardware version of the FLM).

The FLM will be expected to absorb large numbers of traces in some applications. For example, the navigation problem which will be referred to in Chapter IV generated several thousand traces. There is currently some pessimism about the possibility of a device like the FLM being able to handle large amounts of data. Uhr rejects out of hand learning devices which store examples of all the patterns they see:

"...every time it sees a variant instance of a pattern 'learn' it in the sense of defining a new template [trace] in the dictionary memory of templates. But this is ridiculous in practice when we have any interesting amount of variation to our patterns. For such a program would build up an enormously large and unwieldy list of templates". (Uhr, 1973).

Minsky and Papert further conjecture that it will take a large amount of computing to retrieve information on a 'best fit' basis from such a memory:
"...we conjecture that .. for large data sets with long word lengths there are no practical alternatives to large searches that inspect large parts of the memory". (Minsky and Papert, 1969).

Chapter IV proposes and analyses a number of FLM implementations. An actual implementation in Algol for the Burroughs B6700 is reported. It is shown both theoretically and in practice that the time used to update and extract predictions from the FLM memory is small (10 milli-secs of CPU time on the B6700) and independent of the total number of traces in the memory. At worst it is linearly dependent on the order of the FLM.

By coding the memory as a tree structure it is possible to avoid holding each trace in its entirety. It is found in practice that the storage per trace can be kept constant at around 100 bits per trace, although there is a logarithmic dependence on the total size of memory available. With current technology, say a mini-computer with 10 megabytes of disk storage, FLM memories of up to a million traces could be implemented.

I.3.2. Forgetting

Despite the large size possible for an FLM memory it is always possible that the space available will be filled. At this point the FLM might do a number of things;

i) cease operation entirely

ii) continue but not add any more traces to its memory

iii) delete traces from its memory as space is needed for new traces.
i) and ii) are trivial to implement but their acceptability will depend on circumstances. The third possibility is rather more interesting. If the FLM is receiving inputs from an external world which is changing and uncertain then the deletion of old traces may be reasonable as it merely alters the time span over which experience has been accumulated not the total amount accumulated. Indeed in a changing environment it may be important that old traces which no longer occur should be replaced by new ones.

I will term the process of deleting traces to make way for new ones 'forgetting'. It consists of two parts; selecting traces to be forgotten, and then the actual deletion of the chosen traces from the memory structure.

In Chapter IV I will show that the deletion of traces from the memory can be accomplished in times comparable to those required for updating the memory. I will also investigate some possible algorithms for choosing traces to be forgotten. These investigations indicate that it is reasonable to choose traces at random. More sophisticated techniques lead to little improvement and as they usually require more storage per trace they are in fact counter-productive.

I.4. USES OF PRODUCTION SYSTEMS

It was noted in 2. that the traces in an FLM can be considered to form a production system. In this section I will make reference to two uses of production systems in artificial intelligence and psychology and to the reasons why a production system was used as the programming medium in each case.
I.4.1. Human Problem Solving

In (Newell and Simon, 1972) an investigation is made of the algorithms used by humans in solving small arithmetic puzzles and problems. Production systems are used to represent the algorithms used by the different subjects studied. Reasons for using production systems as opposed to any other computer-like language are given on pp 804-805 (Ibid). Some of these reasons are specifically psychological (related to the author's model of human short term memory for example) but others of general interest are:

- a production system can express an arbitrary calculation;
- each production is relatively independent of others and so it is easy to grow a production system incrementally;
- the environment external to the system is able to influence ongoing calculations easily and under the control of the production system;
- the structure of a production system is homogeneous and does not make arbitrary distinctions between code to control the flow of a program, and the actual execution of actions.

I.4.2. Learning of Heuristics

An example of the use of production systems in artificial intelligence is the poker playing program reported in (Waterman, 1970). A set of heuristics, expressed as a production system, was used to determine whether to bet and how much to bet. The heuristics were automatically programmed from experience with a large number of actual games.
The condition part of each production was a statement about the current state of play, and the action part was a restriction on how to bet. A paraphrased example of one such production is:

"If your hand is good and the opponents last bet was not large" -> "bet low".

Reasons given in the paper for using production systems were as follows:

Individual productions were easily manipulated. New rules could be inserted and old ones modified without prohibitive amounts of computing.

Individual productions formed easily isolated units which could be evaluated for their effect on play.

It was possible to code complex heuristics.

In support of the last reason it was noted that techniques used in other adaptive game playing programs, such as weighted polynomials on various features of the game, were unable to make the variety of discriminations needed for this particular application.
I.A.1. FORMAL DESCRIPTION OF FLM

Given a set of events, Ev.

A trace T, of length N, is a sequence of \( N + 1 \) events \( T(N), \ldots, T(1), T(0) \).

The tail of T is the sequence \( T(N), \ldots, T(1) \).

The head of T is the event \( T(0) \).

An FLM is a triple \( (Ev, Buf, Mem) \)

where: \( Ev \) is the set of events;

\( Buf \) is a trace, the buffer trace;

\( Mem \) is the memory, a set of traces.

The length of Buf is the order of the FLM.

The following two algorithms update an FLM memory. The first accepts an event and inserts it into the tail of the buffer trace. The second accepts an event and puts it into the head of the buffer trace which is then stored in the memory.

I.A.1.1. Buffer-update

Given an event \( e \):

\[ \text{[Shift tail of Buffer]} \]

\[ \text{For } i := N \text{ downto 2 do} \]
\[ \text{Buf}(i) := \text{Buf}(i-1); \]
\[ \text{[Put new event into tail]} \]
\[ \text{Buf}(1) := e; \]
\[ \text{[Leave head of buffer undefined]} \]
\[ \text{Buf}(0) := 0; \]
I.A.1.2.  Update

Given an event e:
{Put into head of Buffer, leave tail undisturbed}
Buf(0) := e;
if there is insufficient room to store a new trace in Mem
then Mem := some subset of Mem ; {forgetting}
Mem := Mem V {Buf};

I.A.2.  PREDICTION

PREDICT(S, Mem, l) =
{T: T ∈ Mem and length(T) ≥ 1 and length(S) ≥ 1 and
T(i) = S(1), 1 ≤ i ≤ l}
Note that if length(S) < 1 then PREDICT(S, Mem, 1) = Ø.

Predict(S, Mem, l) =
if l=Ø then Ev
else {T(0): T ∈ PREDICT(S, Mem, l)}.

Bm(S, Mem) =
maximum l, 0 ≤ l ≤ length(S)
such that Predict(S, Mem, l) ≠ Ø.

PRED(S, Mem) = PREDICT(S, Mem, Bm(S, Mem)).
Pred(S, Mem) = Predict(S, Mem, Bm(S, Mem)).

For an FLM F = (Ev, Buf, Mem)
Pr( F ) = Pred(Buf, Mem).

I.A.3.  FLM COMPUTER

Let Ev = Ac V Pa,
where Ac is a non-empty set of actions
Pa is a non-empty set of patterns

Presume that the three following procedures exist external to
the FLM:
Execute (a): (a \in A_c)
executes an "action" in the external world outside the FLM
and as a result returns a pattern.

Accept (A):
obtains a non-empty set of events which is a subset of A
from some source external to the FLM (this might be a human
programmer).

Choose (A):
returns as a result some action chosen from the set A. Any
information including a dialogue with the programmer can be
used.

The following program defines the operation of an FLM computer based on
an FLM F = (Ev, Buf, Mem). It may be in either "program" or "run" mode.

```
computer-loop:
{Ch is the set from which the next action will be chosen}
    Ch := Fr(F);
    if Ch \supseteq A_c or in-program-mode
    then Ch := Accept (Ac);
    {a is the action to be executed}
    a := Choose (Ch);
    {p is the pattern which results}
    p := Execute (a);
    Update (a); {store a trace}
    Buffer-update (a); {recall action in buffer}
    Buffer-update (p); {recall pattern in buffer}
    goto computer-loop; {repeat it all again}
```
This chapter is concerned with analyzing the abilities of the FLM as a computer. It starts by investigating various programming techniques taken from high level languages and shows which of these are possible on the FLM. For example, it is shown that subroutines can be used provided they are sufficiently short.

It is then shown that the FLM can simulate any finite automaton and also that it can be programmed to simulate many finite automata merely by observing their external behaviour. The low execution time of the FLM is then contrasted with an algorithm proposed elsewhere which is capable of reconstructing any finite automaton given only its external behaviour. The execution time of this algorithm is shown to be more than exponential in the number of states in the automaton.

The chapter concludes by showing the very important result that it is possible to speak meaningfully of a program in the FLM memory even though the contents of the memory are continually changing.

II.1. THE POWER OF THE FLM COMPUTER

One important facet of a computer language is its power or conciseness; that is, its ability to express an algorithm in a few well chosen symbols. The power of a language will vary depending on the type of algorithm which it is expressing. So one language might contain facilities for doing arithmetic on complex numbers, and another for manipulating strings of text. To a large extent these variations in
power depend on the 'atomic' operations of the language, so a language for complex arithmetic will allow a statement such as "A + B" to indicate the addition of the complex numbers A and B.

In precisely this way, the power of the FLM will depend on the atomic 'actions' available to it. For example, one might provide the FLM with a calculator and read its display. Such possibilities prevent a direct assessment of the FLM's power without reference to the external facilities available to it.

The ability to synthesize longer units from atomic operations is an important part of any language. This section examines the FLM's ability to synthesize actions by considering how effects available in other languages can be duplicated by it.

I will consider now how compound conditions can be built up from simple ones. Composites of the following three types will be considered:

\[(\text{condition 1 and condition 2}) \rightarrow \text{action};\]
\[(\text{condition 1 or condition 2}) \rightarrow \text{action};\]
\[(\text{not condition 1}) \rightarrow \text{action}.\]

(The arrow, \(\rightarrow\), was introduced in I.2.1 and indicates that if the condition is true then the action should be executed.)

II.1.1. Disjunction

In high-level languages one can achieve an effect similar to the production:

\[(\text{condition 1 or condition 2}) \rightarrow \text{action}.\]

If either condition 1 or condition 2 is true then 'action' is executed. The logical effect of this statement is easily obtained by
splitting it into two separate productions:

condition 1 \(\rightarrow\) action

condition 2 \(\rightarrow\) action

Two traces, say

\[ C \ c \ D \ d \ A \]

\[ E \ e \ F \ f \ A \]

form the disjunction of the two 'conditions' \(C \ c \ D \ d\) and \(E \ e \ F \ f\).

In many languages 'action' itself might well be made up from smaller units. So one of the effects in these languages of (condition 1 or condition 2) \(\rightarrow\) action is that the code for 'action' need only be written once, not twice as is necessary when it is split into two productions. A similar situation occurs in the FLM when A is the start of a longer sequence of actions. When the second trace is entered into the FLM it is not necessary for the programmer to explicitly reprogram the sequence following A. To demonstrate this I will consider an example.

Presume that the sequences to be entered are:

\[ \ldots C \ c \ D \ d \ A \ a \ R \ r \ S \ s \ T \ t \ldots \]

and \[ \ldots E \ e \ F \ f \ A \ a \ R \ r \ S \ s \ T \ t \ldots \]

and that the first sequence of events has already been programmed. What happens when the second sequence of events is entered in program mode is illustrated in Fig. 1. The actions E, F and A must be chosen explicitly by the programmer. The following actions R, S and T will be selected by the FLM and the programmer need only acquiesce to this selection, not actually enter the action. Only the first action A of the sequence A a R r S s T t... needs to be entered for the entire sequence to be repeated.
Sequence of events entered previously:

\[ \text{...C e D d A a R r S s T t...} \]

Buffer Trace (Order=4)  How Action Chosen
                      (In program mode)

\[ \begin{align*}
\text{............E} & \quad \text{Programmer} \\
\text{......E e F} & \quad \text{Programmer} \\
\text{E e F f A} & \quad \text{Programmer} \\
\text{F f A a R} & \quad \text{FLM} \\
\text{A a R r S} & \quad \text{FLM} \\
\text{R r S s T} & \quad \text{FLM}
\end{align*} \]

Fig. 1. Example of Common Sequence of Events Triggered by an Initial Action
II.1.2. Conjunction

The second construct I will consider is

(condition 1 and condition 2) -> action;

where "action" is executed only if both the conditions are true. An
effect similar to this can be achieved in the FLM by concatenating
events into a trace. So the trace

C c D d A

can be viewed as the production

("C c" occurs and then "D d" occurs) -> A

Concatenation of events in this way is restricted by the limited
length of the FLM buffer. For example the two traces:

C c D d E e A
C b D d E e B

cannot be distinguished in an FLM of order 4 as the tail D d E e will
predict both A and B.

This problem can be solved by introducing a special class of
actions, echo actions. (So called because of their similarity to
hearing an echo of a word just spoken.) The effect of an echo action is
to return a pattern equal to the action itself. It has no other effect
on the external world. (I will use events prefixed with a hash (E.G.
"#1") to denote echo actions and their resultant patterns.)

An echo can be used as a 'place keeper' to maintain information
within the buffer. The first of the two traces above can now be
re-expressed in the sequence:

...C c D d #1 #1 E e A...

which generates the traces

C c D d #1
Similarly the second trace can be recoded as:

...C b D d #2 #2 E e B...

which generates the traces

C b D d #2
D d #2 #2 E
#2 #2 E e B

The two sets of traces are disjoint, because of the presence of the two different echo actions and so a unique prediction of the actions A and B is possible. This re-coding may be simplified as the second echo (#2) is not essential. The sequence

...C b D d E e B...

generates the traces

C b D d E
D d E e B

both of which are distinguishable from those incorporating #1 above.

Provided sufficient echo actions are available, an arbitrarily long sequence of events can be used to control an action. For example the two sequences,

...C b D d E e F f X...
...C c D d E e F f Y...

can be distinguished by recoding the first as:

...C b D d #1 #1 E e #2 #2 F f X...

In an FLM containing large numbers of traces it might be that large numbers of echo actions would be required to maintain sufficient distinction between the traces. For example it would be quite unsatisfactory if the number of echo actions was comparable to the
total number of traces. However, it is only necessary that an entire trace containing an echo action predict a unique action, not that a different echo action be used in each case. A particular echo action might be used in many different sequences of events and the traces generated from them. For example it is sufficient to code the sequence above as:

...C b D d #1 #1 E e #1 #1 F f X ...

using the echo #1 twice rather than the two different echoes #1, #2. Also each of the three automata in Section 3 and Fig.5, which cannot be simulated without using echo actions, require only one echo action for a successful simulation.

Note

Techniques similar to the echo action have been proposed independently by a number of authors. The form used here and the term 'echo action' are taken from (Andreea, 1977).

In the production system in (Newell and Simon, 1972) the action part of a production can be a sequence of primitive actions. It is suggested (p 805, ibid) that each such production containing more than one primitive action could be broken down into a number of productions, each linked to the next by a unique symbol.

In (Suppes, 1969a) it is shown that an SR table can simulate any finite automaton. The technique uses a compound stimulus including part of the last response to record the state of the automaton being simulated. In this way each response is partly echoed in the next stimulus.
II.1.3. **Negation**

The final construct to be considered is negation:

\[
\text{(not condition)} \rightarrow \text{action}
\]

where action is executed only if the condition is not satisfied.

It is impossible to duplicate this effect using our simple FLM computer. To see the difficulties involved consider the following example:

Following the sequence C c D any one of the patterns a, b, c, ...
... z can occur. It is intended that after "a" the action A should be
executed and after any other letter that Z should be executed. This
roughly corresponds to the productions

\[
\begin{align*}
\text{("C c D" and then not "a") } & \rightarrow \text{ Z} \\
\text{("C c D a") } & \rightarrow \text{ A}
\end{align*}
\]

These can be programmed in the FLM by the traces

\[
C \ c \ D \ a \ A \\
C \ c \ D \ b \ Z \\
" \\
" \\
C \ c \ D \ z \ Z
\]

In this case there are 26 alternatives to be programmed. However, in many real cases it will be impossible to enumerate all the
alternatives. An enumeration in this way certainly does not capture the
spirit of the not construct. If one of the cases say "C c D z" has not
been programmed then either the FLM will stop and ask the programmer
for an action, or in run mode it might choose an inappropriate action
because "D z" matched some other 'irrelevant' trace.

There is a simpler version of this problem; that is, to execute
an action independent of the last pattern which occurred. So if "Z" is
to be executed after "a" in the problem above then the problem can be
reduced to:

"After the sequence C c D execute Z independent of the pattern
following D."

Even this simpler problem is not soluble without programming each case
separately. However, a solution to this using a multiple FLM system
will be given in the next Chapter(III.4). III.4.3 also briefly mentions
a storage mechanism, similar to that of the FLM, which might allow the
general problem to be solved.

II.1.4. Subroutines

An important feature of many languages is the ability to write a
fragment of a program and then have it used many times without it
having to be reprogrammed each time. In one sense the FLM is
continually doing this, for no matter how often a trace occurs it is
still stored only once. Consider for example the two sequences of
events used in Fig. 1 to illustrate the formation of a disjunction of
conditions. It was only necessary for the programmer to enter the
action "A" the second time for the entire following sequence to be
recapitulated. In this way "A" acted as a unique label identifying the
following sequence or 'subroutine'.

Sharing code in this way does not capture fully the idea of
subroutines (or procedures) as implemented in many other computer
languages. A crucial feature of a subroutine is that after it has been
executed the program should be controlled by the situation which held
before the subroutine was entered. A subroutine can be visualised as a
black box with an unknown and arbitrarily complex p interior. There are many paths into the box but after passing through each splits again and goes its own way, independent of what happened inside the box.

Using an FLM the only way that the situation before entering the subroutine can affect that on leaving is for every path within the subroutine to be less than one trace in length. Then the last events prior to entering the subroutine can have some effect on those just after leaving it. This limits the usefulness of such short pieces of code.

There is another difficulty best illustrated by an example. Consider the short subroutine in Fig. 2 which consists of two different possible sequences and which is shared by two separate paths. Provided the FLM is of order 6 it is possible to successfully program this. However, all four possible combinations of events must occur before the program is complete. Fig. 2 shows each of these cases as they might occur while the program is being entered. Also shown (by underlining) are those actions which would need to be explicitly entered by the programmer in program mode. In the second sequence all the actions A, B and C will be predicted, from the initial sequence, but the T and E need to be explicitly entered. Note that in all four cases the first action upon leaving the subroutine (E or Z) must be entered by the programer. In general, if there are p paths using the subroutine and q
different sequences within it, then all $p$ times $q$ possible combinations must be programmed with at least one action being specified explicitly by the programmer. For a true subroutine the amount of programming would be proportional to $p$ plus $q$. So although the FLM helps reduce the programming effort (actions C, D and T are predicted on lines 2, 3 and 4) larger savings might be hoped for, especially for more complex subroutines.
Shared Code
(Subroutine)

Actions not correctly predicted by the FLM are underlined (assuming an FLM of order 6, and that the sequences are seen by the FLM in the order given.)

... A a B b C c D d E e ...
... A a B b C s t t E e ...
... X x Y y C c D d Z z ...
... X x Y y C s t t Z z ...

Fig. 2 Example of a Short Subroutine and the Sequences Required to Program It.
II.1.5. **Summary**

I have shown that there are limitations on the power of the FLM computer imposed by the finite length of the buffer and its inability to ignore the last pattern seen. These restrictions are matters of efficiency, not of absolute ability, because an FLM with echo actions can simulate any finite automaton. Nevertheless the utility of the FLM will be greatly improved if the restrictions can be overcome. It is shown how this can be done in the next chapter using the Dual FLM and other multiple FLM systems.

II.2. **COMPLEXITY**

The abstract definition of a program's complexity is the size of the smallest description of the program. In practice, such a precise measure can be obtained only for the simplest cases. For actual programs the complexity is usually estimated from the size of the source code. It can be seen from the description of the FLM in the appendix to Chapter I and from the source of the implementations of Chapter IV that the FLM itself is very simple.

There is another aspect of the FLMs complexity which is of importance; that is the complexity of the inputs required to program it. This is of particular relevance if the programming inputs to the FLM are provided by some automatic process and not by a human programmer. The symbols or actions transmitted from the programmer to the FLM constitute a language with its own rules of composition. Any automatic mechanism which generates statements in such a language must
obey these composition rules. For example, the system mentioned in Chapter I, 4.2 which generates heuristics for poker must obey the rules of the subset of LISP which it uses. If the rules of composition are very complex then the generating mechanism must in turn be complex to successfully obey all of them. If they are simple then the mechanism may itself be simple (although other considerations may increase its complexity.)

The FLM has the advantage that the rules for input to it are very simple. There are only two; that the programmer supply a valid action; and in run mode that an action be supplied only when asked for. The syntax of most currently available languages is much more complex than the FLM. Even if a subset of a language such as FORTRAN or Algol is used then a great deal of complexity must still be retained in order that the subset have sufficient power to be useful. This greater complexity makes such languages unsuitable for automatically generating instructions in the language.

It was seen in previous sections how echo actions could be used to increase the power of the FLM. The use of echo actions, however, makes programming the FLM more difficult. The extra difficulty lies not in their presentation to the FLM, which is identical to any other action, but in their correct usage. For example, when an echo is used (as in 1.2) to separate into two an otherwise ambiguous trace then it is necessary to anticipate, at the beginning of the trace before the ambiguity arises, that an echo action is to be used. It should be noted that neither the automatic programmer of Section 3 nor PURR-PUSS can construct echo actions without intervention from a human programmer. However, in PURR-PUSS echo actions once programmed (by a human) can be integrated into later automatically generated sequences.
II.3. FINITE AUTOMATA

This section shows first that by suitable programming an FLM can simulate any finite automaton, and second that in some cases the external behaviour of the automaton is sufficient to effect this programming. Finite automata are of interest because any algorithm can be executed using a finite automaton coupled with a suitable memory device such as a Turing tape. It would be a severe limitation on the FLM if there were some class of algorithms accessible to finite automata but not to it. Whether it is practical to use an FLM for a particular calculation depends also on other considerations such as speed and efficiency.

It will be shown first that using sufficient echo actions (one for each automaton state) any finite automaton can be simulated by an FLM. This is contrived and inefficient; however, it will be seen that for many automata no echo actions at all are necessary for their programming. This allows these automata to be programmed by showing the FLM a sufficient sequence of the external behaviour of the automaton. Finally, some examples will be given of automata for which echo actions are essential if the automata are to be programmed. In each case one echo action is sufficient to achieve the programming.

II.3.1. Simulation of Automata

This section restricts its attention to transition output automata. In such automata the transitions from one state to another are labelled with a single input and output symbol. Whenever a particular input occurs, the transition from the current state for that
input is selected (it is unique), the output attached becomes the next output of the automaton, and the state on which the transition ends becomes the next state of the automaton.

Fig. 3 illustrates how the FLM computer can be used to simulate such an automaton. The automaton inputs are treated by the FLM as patterns and the automaton outputs are simulated by the non-echo actions chosen by the FLM. (The echo actions, if any, used by the FLM remain private to it.) To construct an FLM program for a particular automaton a unique echo action is assigned to each state (say #1 for state 1 and so on). The FLM, after receiving an automaton input as a pattern, chooses as its next action the echo corresponding to the state at the end of the appropriate transition. It then selects as its next action the automaton output for that transition. Recalling that every echo action occurs twice, once as an action and once as a pattern, the events received by the FLM will occur in the following cycle:

...#state #state output input #state #state output input....

To be more precise, if the automaton executes the following sequence:

state-1, input 1 then output-1 and transition to state-2
state-2, input 2 then output-2 and transition to state-3

then the sequence of events received by the FLM will be,

...input-1 #state-2 #state-2 output-1
input-2 #state-3 #state-3 output-2.

(This is not necessarily the most compact representation of the automaton.)

For the automaton to be simulated correctly each action in the sequence must be predicted uniquely, whether it is an echo action specifying the next state or a non-echo action specifying the next automaton output. The last state and the last input uniquely predict
the next state so all traces of the form

\#state output input \#state \#state : output
must also be unique. The FLM will simulate the automaton correctly if
it contains at least the set of program traces generated by all traces
of the two forms above.

Fig. 4 gives an example of a simple automaton drawn both as a
network and as a table, giving the output and new state for all
combinations of initial status and inputs. The figure also gives two
FLM programs for simulating the automaton. The first of these is
constructed as explained above, the second will be explained in the
next sub-section. A short example is also given of the operation of the
two programs.

In (Andreae and Cleary, 1976) an example is given of the
PURR-PUSS system programmed to behave as a Universal Turing Machine.
This programming was done using essentially the techniques described
above.
Fig. 3 Simulation of an Automaton by the FLM
II.3.2. Programming Automata

To program an automaton on the FLM in this way requires that the programmer have a knowledge of the internal states of the automaton being programmed. However, there exist automata where the current state can be determined uniquely just by examining the most recent inputs to and outputs from the automaton. In this case, no echo actions are needed for an FLM program as the current state will be implicit in the events currently in the buffer trace.

To be more precise, if the current state of the automaton can be determined by the last \( k \) events (automaton inputs and outputs) then it can be simulated by an FLM of order \( k \) (or greater) without using echo actions.

To enter such a program it is sufficient to present the FLM with actual sequences of the automaton's input and output behaviour. That is, merely by observing the external behaviour of the automaton the FLM is eventually able to mimic it.

An arrangement for doing this is illustrated in Fig. 3. The FLM is connected to the automaton to be simulated so that it sees both its inputs and outputs. In program mode, the outputs of the automaton are automatically chosen as the next action by the FLM. Then, provided a sufficient variety of inputs is received, the FLM will eventually collect all the traces necessary to simulate the automaton. It can then be switched to run mode and each action predicted and chosen by the FLM will equal the output of the automaton.
Specification of Automaton to be simulated as a transition diagram. Each pair, say pA, indicates an input p and output A for that transition.

\[
\begin{array}{c}
\text{pA} \\
1 \\
qA \\
2 \\
qB \\
pA
\end{array}
\]

\[-\#1\ A \\
q \#2 \ #2 \ B \\
p \ #2 \ #2 \ A \\
\#1 \ A \ p \ #1 \\
\#2 \ A \ p \ #2 \\
- \ B \ p \ #2 \\
\#1 \ A \ q \ #2 \\
\#2 \ A \ q \ #1 \\
- \ B \ q \ #1
\]

FLM program generated as specified in 3.1. Shorter FLM program generated as specified in 3.2.

These programs are given as a minimal list of traces needed for correct execution. A - indicates that any event may appear there, and will in practice correspond to a number of traces.

<table>
<thead>
<tr>
<th>FLM Events (for program 1)</th>
<th>Automaton</th>
<th>FLM Events (for program 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actions</td>
<td>Patterns</td>
<td>Outputs</td>
</tr>
<tr>
<td>B</td>
<td>#2</td>
<td>p</td>
</tr>
<tr>
<td>A</td>
<td>#2</td>
<td>p</td>
</tr>
<tr>
<td>A</td>
<td>q</td>
<td>A</td>
</tr>
<tr>
<td>#1</td>
<td>#1</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>#1</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>q</td>
<td>A</td>
</tr>
<tr>
<td>#2</td>
<td>#2</td>
<td>B</td>
</tr>
</tbody>
</table>

Fig. 4 Example of Automaton Simulation

54
There exist automata, however, which cannot be programmed in this simple fashion no matter how long the FLM buffer. One such automaton is specified in Fig. 4. In it, arbitrarily long sequences of the form \( p A q, p A p A q, p A p A p A q \ldots \) and so on can occur in either state 1 or state 2. (Following my normal convention, lower case letters \((p)\) indicate patterns and in this case automaton inputs, and upper case letters \((A)\) indicate actions and in this case automaton outputs.) If such a sequence longer than the order of the FLM should occur it would be impossible to determine whether the next output should be \( A \) in state 1 or \( B \) in state 2. It is essential, then, to use echo actions to program this automaton. However, it is not necessary to use all the states as echoes as was done in the last section. Only the two possible versions of the sequence \( \ldots p A p A p A q \) need to be distinguished. This can be done by echoing, say state 1. The sequence \( \ldots p A p A q \) then becomes \( \ldots p \#1 \#1 A p \#1 \#1 A q \) in state 1 and \( \ldots p A p A q \) in state 2. These two sequences are distinguishable with a buffer of three or more events.

Fig. 4 shows the program which results from this more economical use of echo actions. This may be compared with a similar reduction when forming the conjunction of two conditions in 1.2.

II.3.3. **Efficiency of Programming**

I have shown above that not all states need be echoed to program an automaton. To gain some more general idea of the extent to which automata need to be supplemented by echo actions I will consider programming a set of automata proposed elsewhere as a test for an automaton simulation algorithm.
(Bierman, 1972) gives an algorithm for synthesizing automata from samples of their input/output behaviour. It is shown there that any finite automaton can be reconstructed by this algorithm, given a sufficiently representative sample of its behaviour. So, for example, the automaton of Fig. 4 could be reconstructed by the algorithm. The reference gives a list of nine automata which were successfully reconstructed. These are reproduced in Fig. 5 in transition output form. Each transition is tagged with a triple (E.G. AXL) where the first (A) is the automaton input. The last two (XL) specify the automaton output when the transition is traversed. The automata are designed to form part of a Turing machine so the output consists of a symbol to be written to the Turing tape (X) and an action to be performed on the tape (L), (L means move left, R move right).

Of these nine automata, six can be programmed on the FLM without being augmented by any echo actions. (This can be checked by enumerating all sequences up to a fixed length leading to each state and then verifying that no sequence leads to more than one state.) The three remaining automata, 5, 7 and 8 of Fig. 5, each require only one state to be echoed for a successful FLM program. For example, in automaton 5 the arbitrarily long sequence ... B B R B R B B R A can occur in either state 1 or 2. Echoing either one of #1 or #2 removes the ambiguity. Similarly, in automaton 7 states 3 and 4 and in automaton 8 states 2 and 3 need to be separated by one of them being echoed.
Fig. 6 Automata Taken from (Bierman, 1972, Fig. 3). (see 4.2)
II.3.4. Psychological Speculation

I will now speculate upon the psychological significance of these results. It is possible that there are classes of learnt human behaviour which are remembered and coded using the principles of the FLM. Simple sequences of motor actions seem likely candidates although there is no apparent reason why, at the simplest level, all human action should not be organized on this basis. Such actions will split into two classes according to the ease with which they can be learnt. In the first class there will be those sequences where at every moment there is sufficient input from the environment (and immediately preceding inputs) to predict the next action uniquely. These sequences will be readily learnt by the simple process of experiencing the sequence of actions involved. In the second class will be those where the inputs from the environment are ambiguous. These correspond with automata which require echo actions for their implementation. These will be more difficult to learn as some 'echoing' process must be learnt to support the actions.

Echo actions might take many forms in the human body, from gross and obvious phenomena such as talking to oneself or tapping one's foot in rhythm to completely internal and invisible processes. How these echo actions might be learnt (or taught) is beyond the scope of this thesis.
II.4. EXECUTION TIME FOR AUTOMATON SYNTHESIS

An automata synthesis algorithm given in (Bierman, 1972) is capable of reconstructing any automaton without recourse to such stratagems as echo actions. However it is not a viable alternative to the FLM. This section shows that the computation time of the algorithm rises very rapidly with the size (number of states) of the automaton being constructed. This increase is particularly serious when the automaton being modelled is non-deterministic and the number of states required to model it increases unboundedly with the number of inputs and outputs observed.

A lower bound on the computation time will be obtained, the dominant term of which is more than exponential in the product of the number of states and the size of the input alphabet.

The details of the algorithm are given in (Bierman, 1972) together with some worked examples. The description given here is sufficient to enable the derivation of the execution time. However, anyone unfamiliar with the technique should check the reference.

The algorithm is to construct a finite automaton to model some given sequence of inputs to and outputs from an unknown finite deterministic automaton. I will consider the case where the model is a transition-output automaton. That is, each transition is determined by its initial state and an input symbol and specifies a single output symbol and state on which it terminates (see Fig. 5 for some examples).

The algorithm starts with an empty model (no transitions specified) and as well there is specified an upper limit on the number of states which may be used. The usual procedure is to set this to some small initial value, and if the algorithm fails to construct a model
with this limit to increment it and start again.

The algorithm samples each input/output pair in succession calling itself recursively each time a new transition is added to the model. At the start of each step the simulated automaton has some set of transitions specified and is in one of its states. The next input symbol is examined. If there is a transition specified in the model for the current state and the input symbol, then a check is made that the actual output symbol agrees with that predicted. If it does, then the state is moved to that at the end of the transition and the next input sampled. If it does not then the algorithm backs up one level of its recursive calls to where it last inserted a new transition.

If there is no transition specified in the model then a new one will be inserted. There are a number of ways this can be done and each is tried; if it fails to produce a consistent model then one of the other possibilities is tried until, if all fail, the algorithm backs up one recursive level. So that this can be done efficiently a record is kept of the states to which at least one transition has been specified so far. Each of these is selected in turn. A new transition is inserted in the model from the current state to the selected state. The input and output symbols associated with the new transition are those currently being sampled. A recursive call is then made to the algorithm. If this fails (that is, it backs up because the model and the input output sequence are inconsistent) then this new transition is removed and another inserted to some other used state. If all these fail and less than the maximum allowed number of states has been used so far, then a transition is inserted to some previously unused state. If this fails then the algorithm backs up one level.

In this formulation of the algorithm it will continue forever if
it can find a satisfactory model. However, we are interested in the case where there is no \( n \)-state deterministic automaton which can generate the behaviour, where \( n \) is the maximum allowed number of states. Let \( T(n) \) be the execution time of the algorithm if it fails, that is it eventually backs up to the start. Then even if there is some \( n \) state model it will still take at least \( T(n-1) \) time to find this representation.

Consider the algorithm in more detail. Let \( n \) be the maximum allowed number of states. Number each state in the model from 1 to \( n \) (inclusive). Let \( \text{input}(t), \text{output}(t) \) be the \( t \)'th input and output symbols respectively of the sequence being modelled (where \( \text{output}(t) \) immediately follows \( \text{input}(t) \)).

Let \( \text{check}(t, s, m) \) be the recursive procedure. It has three parameters:

- \( t \) the next input/output pair to be sampled
- \( s \) the current state of the model
- \( m \) the number of states used so far

To construct an \( n \) state modelling automaton a call is made to \( \text{check}(0, 1, 1) \).
Check (t, s, m):

Start: if s has a transition assigned for input(t)
then
1. Begin
   if output(t) equals the output for the transition
   then
   1.1 Begin
      t := t + 1;
      s := terminating state of the transition;
      goto start;
   End;
   End else
   Begin {insert a new transition in the model}
   For i := 1 upto m do
   2.1 Begin
      insert a new transition from s to i with input(t),
      output(t);
      check (t + 1, i, m);
      remove the new transition;
      {use a virgin state if available}
      if m < n then
   2.2 Begin
      insert a new transition from s to m + 1 with
      input(t), output(t);
      check (t + 1, m + 1, m + 1);
      remove the new transition;
   End;
   End;

T will be estimated by counting the number of calls made to check. This will be found to depend on n, the maximum number of states allowed and a, the number of symbols in the input alphabet of the automaton being modelled. T(n,a) then will be the expected number of calls to check when the algorithm fails to find any n-state automaton to model a given input output sequence. T(n,a) will be obtained from the values of c(k,m), the number of further calls to check made when check is entered with k transitions already assigned to the model and m states used so far. (c(k,m) depends on n and a, but these are not explicitly noted to ease the notation.) Note that 0 ≤ k ≤ am and 0 ≤ m ≤ n. As the model is empty when check is first called,

\[ T(n,a) = c(0,1) \]  
(1)
Now step 2 of check occurs with probability \((1-k/\text{am})\) \((k\text{ out of a possible }\text{am transitions have been assigned})\). Assuming each call to check fails then step 2.1 generates an expected \((1-k/\text{am})\).\text{m}.c(k+1,m)\) calls \((k+1\text{ because }1\text{ new transition is added to the model})\). Step 2.2 occurs only if \(m < n\) and generates an expected \((1-k/\text{am})\).c(k+1,m+1)\) calls. Combining these one obtains:

\[
c(k,m) = 1 + (1-k/\text{am}).(\text{m}.c(k+1,m)+c(k+1,m+1)) \quad 1 \leq m \leq n \quad (2a)
\]
\[
c(k,n) = 1 + (1-k/\text{am}).\text{m}.c(k+1,m) . \quad m = n \quad (2b)
\]

From this it is possible to derive a closed expression for a lower bound to \(T(n,a)\). Consider the following inequalities derived from (2):

\[
T(n,a) = c(k,m) \geq (1-k/\text{am})\cdot c(k+1,m+1) \quad 1 \leq m \leq n \quad (3a)
\]
\[
c(k,n) \geq (1-k/\text{an})\cdot n\cdot c(k+1,n) . \quad (3b)
\]

Using these relationships bounds can be obtained for \(c(1,1)\) in terms of \(c(n,n)\) and for \(c(n,n)\) in terms of \(c(an,n)\).

From (2b) \(c(an,n) = 1\) . \quad (4)

From (3b) \(c(n,n) \geq (an-n)/a\cdot (an-n-1)/a\cdots 1/a\cdot c(an,n)
\]
\[\geq (an-n)! . \quad (an-n)\] then using (4) \(\geq a \cdot (an-n)! . \quad (5)
\]

From (3a) \(c(o,1) \geq 1\cdot (1-1/a.1)(1-2/a.2)\cdots
\]
\[\cdots (1-(n-1)/a\cdot (n-1)).c(n,n)
\]
\[\geq (an-n)! . \quad (an-n)\]

then using (5) \(\geq (1-1/a)\cdot a \cdot (an-n)! . \quad (an-n)\)

Rearranging and using Stirling's approximation for the factorial,

\[
\frac{1}{2} \quad \frac{a-1}{2} \quad 1
\]

\[
T(n,a) > /e \cdot (an-n) [(n(a-1)/ea) \cdot (1-1/a)] . \quad (6)
\]

Ignore the initial constant and square root, to obtain

\[
T(n,a) \sim \left(n\cdot \exp\left(-a/(a-1)\right)\right) . \quad n(a-1)
\]

and for large \(a\) \(\sim \left(n/e\right)\).
This bound is weak but still sufficient to illustrate the more than exponential increase in computing time with the number of states. It has been shown in (Gaines, 1976) that if the output symbols are generated at random then the expected number of states needed in the smallest automaton compatible with a sequence of inputs and outputs of length N is $N - \log_2(N)$. So not only will the execution time increase more than exponentially with the number of states but also with the length of the example sequence. This may be contrasted with the FLM whose execution time is proportional only to the length of the input sequence. The FLM might be practical when used with long input sequences, the algorithm above is certainly not. Table 1 gives some values of $T(n,a)$ calculated from (2) and (6).
Fig. 6 Summary of notation used in deriving execution time

Table 1 Values of \( T(n,a) \)

<table>
<thead>
<tr>
<th>( n ) / ( a )</th>
<th>( 2 )</th>
<th>( 3 )</th>
<th>( 4 )</th>
<th>( 10 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3.75</td>
<td>6.77</td>
<td>11.8</td>
<td>221</td>
</tr>
<tr>
<td>3</td>
<td>5.25</td>
<td>22</td>
<td>109</td>
<td>6.69x10^5</td>
</tr>
<tr>
<td>4</td>
<td>7.56</td>
<td>136</td>
<td>2.87x10^3</td>
<td>4.34x10^6</td>
</tr>
<tr>
<td>10</td>
<td>1.52x10^3</td>
<td>3.66x10^6</td>
<td>2.04x10^7</td>
<td>1.21x10^55</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( n ) / ( a )</th>
<th>( 2 )</th>
<th>( 3 )</th>
<th>( 4 )</th>
<th>( 10 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3.12x10^{-2}</td>
<td>3.36x10^{-2}</td>
<td>2.54x10^{-2}</td>
<td>1.34x10^{-3}</td>
</tr>
<tr>
<td>3</td>
<td>2.37x10^{-2}</td>
<td>7.51x10^{-2}</td>
<td>.151</td>
<td>2.05</td>
</tr>
<tr>
<td>4</td>
<td>2.38x10^{-2}</td>
<td>.313</td>
<td>2.33</td>
<td>6.33x10^4</td>
</tr>
<tr>
<td>10</td>
<td>.893</td>
<td>3.13x10^6</td>
<td>3.36x10^{12}</td>
<td>1.35x10^{47}</td>
</tr>
</tbody>
</table>
II.5. REPRESENTING THE FLM

When dealing with the formal properties of a computer language it is usual to do so in terms of fixed statements in the language. One might take some set of statements and show that as a result of them, say, the square root of an input parameter is correctly calculated or, as in the last section, an automaton is simulated. The statements which specify the square root calculation are not themselves modified by executing the calculation. They remain the same afterwards and can be used again to calculate another square root.

It was noted in I.1.4 that even when operating in run mode, without any intervention being required from the programmer, new traces can be added to the FLM memory. There is thus some doubt as to whether a set of traces which would successfully complete a particular calculation would behave the same way again after the calculation was performed.

This problem would be resolved by a description of the FLM memory contents which satisfied two conditions:

i) that the description held true before and after the calculation;

ii) that the description was sufficient to specify the course of the calculation.

Note that it is not necessary for the description to completely describe the FLM's behaviour as the description need only be sufficient to ensure that the calculation is carried out.

The description sought can be provided by the FLM itself! Let the FLM memory be frozen and consider two versions running side by side, the one frozen and not being updated, the other being updated
normally. The frozen version of the FLM will be able to predict much of
what the running FLM predicts. To be precise, the frozen FLMs
predictions will be superset of the running FLMs predictions. (This is
proven in Theorem A.1).

If the predictions of the frozen FLM are deterministic (there is
only one action predicted) for a particular trace then the prediction
of the running FLM will also be deterministic. So any algorithm which
is deterministic and present in the frozen FLMs memory will also be
present and still deterministic in later versions of the running FLMs
memory.

The notion that a computer system must be able to have its
programs represented by some static structure will become very
important in the next chapter. To ensure that this is possible
stringent constraints have to be placed on the multiple FLM systems
considered there.

II.6. SIDE EFFECTS OF PROGRAMMING

II.6.1. Program Mode

When the FLM computer is running in program mode any action may
be chosen by the programmer; he is not restricted to choosing one which
is predicted by the FLM. If he chooses an action which is not predicted
then the overall predictions of the memory are disturbed rather more
than in run mode (see 5. and A.1). However, something can still be said
about what happens. Two cases that need to be considered are where the
current buffer matches some trace in the memory over its full length
(say N), and where the longest match is over some intermediate length
(1 to N-1).

If the buffer has a match over the full length of a trace then whenever this buffer occurs in future it will match this trace as well as the new trace added when the action was chosen. That is; the buffer will predict two (or more) actions whenever it occurs again.

The danger of this is that the programmer may be unaware of all the circumstances under which this buffer will arise and the newly added action, while appropriate when chosen, may cause unexpected side effects in other situations. To alleviate this type of side effect a restriction might well be written into the FLM itself so that actions are only accepted from the programmer, in program mode, if the match is over less than a full buffer length. Such a restriction is implicit in PURR-PUSS where the programmer or 'teacher' is able to enter an action only when PURR-PUSS is unable to select one herself.

The second case to be considered occurs when the buffer does not match any trace in the memory over its full length. Fig. 7 illustrates what can happen in this circumstance.

Any traces which have occurred previously will continue to yield the same predictions after the new trace is added. This gives some protection against unexpected side effects, as any sequence of events which has occurred previously will be executed again correctly in the future. The potential predictions for sequences which have not occurred before may be changed, however. This is illustrated in Fig. 7 by the before and after predictions of some traces which do not occur in the memory.
<table>
<thead>
<tr>
<th>Trace</th>
<th>Original Prediction</th>
<th>New Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Memory</td>
<td>A a B b C c D</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>Z z B b C c D</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>M m N n C c D</td>
<td>D</td>
</tr>
<tr>
<td>New Trace</td>
<td>U u V v C c X</td>
<td>D</td>
</tr>
<tr>
<td>Test Traces</td>
<td>R r S s C c -</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>H h B b C c -</td>
<td>D</td>
</tr>
<tr>
<td></td>
<td>T t V v C c -</td>
<td>D</td>
</tr>
</tbody>
</table>

When a new trace is stored in an FLM memory in program mode, the predictions of traces already present in the memory remain unchanged. However, the potential predictions of traces never encountered previously may be altered (see 'Test' traces above).

Fig. 7. Alterations to Predictions by Traces Stored in Program Mode
II.6.2. Deleting Traces

The FLM mechanism makes no provision for the deletion of a trace once it has been added to the memory. This 'problem' can be ignored, and when it is necessary to alter the action predicted by a particular trace an entirely new set of traces can be programmed to circumvent it. (Those who find this completely unreasonable should consider the difficulty experienced when attempting to re-train an animal (or human) after it has once learnt to perform a task. It is perhaps not too absurd to think that the brain shares this property that once a program is stored it cannot be deleted, only circumvented). All that this requires is sufficient variety in the environment and available actions so that similar effects can be achieved in a number of ways. The larger and more varied the external world outside the FLM, the easier it will be to do such reprogramming. If necessary, echo actions could be used to distinguish the new traces from the old faulty ones.

The problem can be solved more directly by providing a mechanism for deleting a trace. This can be done either by marking the trace and leaving it in the memory, or by actually removing it from the memory. The easier of these two to implement is the former, and examples of implementations are described in (Andreae and Cleary, 1976) and (Andreae, 1977). An example of an implementation of the latter possibility is given in Ch. IV where it is shown, as part of the forgetting mechanism, that it is possible to remove a trace in times comparable to those for updating and prediction. 

Marking or deleting a trace in this way will have side effects similar to those caused by altering an action predicted by a full buffer match. Because of this possibility of side effects and the
uncertainty about the necessity of deletion, I have not made it an integral part of the FLM. I remain content to describe its possibility and potential difficulties until more evidence is available as to its utility.

II.7. SUMMARY

This chapter has analysed both what the FLM can be programmed to do and what cannot be easily and conveniently programmed. These analyses are underpinned by the result of section 5 and Theorem A.1 that it is possible to talk meaningfully about a program in the FLM even though the contents of its memory are continually being altered. Without such a result all the programming techniques and examples would have no substance.

Some standard programming techniques have been shown to be possible. These can reduce the amount of interaction necessary to encode an algorithm. For example, simple subroutines can be developed with the aid of echo actions, although their length is limited by the length of the FLM's buffer. However the negation construct where an action is executed in the absence of a positive condition is effectively impossible.

The basic capability of the FLM is proven by its ability to simulate any finite automaton. This is of importance, not so much for what it says the FLM can do, as for the fact that it would have been most unfortunate had the FLM not been able to simulate some finite automaton. The ability of the FLM to readily absorb information from its surroundings is shown by the fact that it can be programmed to
simulate many finite automata merely by observing the outputs from and inputs to the automaton.
II.A.1. Invariance of Memory

If, when a trace is added to a memory, its head is predicted by the memory then the predictions of all traces from the updated memory will be equal to or a subset of the predictions from the old memory.

Theorem

For a trace Buf and set of traces Mem, Buf(0) ∈ Pred(Buf, Mem) implies for all traces S

\[ \text{Pred}(S, \text{Mem}) \supseteq \text{Pred}(S, \text{Mem} \cup \{\text{Buf}\}) \]

Proof

The proof will consider three cases where the match of S with Buf is less, equal to or greater than its longest match with a trace in Mem.

Let \( b = \text{Bm}(S, \{\text{Buf}\}) \)
\[ m = \text{Bm}(S, \text{Mem}) \]

Consider now the three (exhaustive) cases, \( b < m \), \( b = m \), \( b > m \).

1) \( b < m \)

The traces which match S over length m in Mem will continue to do so, as a result:

\[ \text{Bm}(S, \text{Mem}) = \text{Bm}(S, \text{Mem} \cup \{\text{Buf}\}) \]

and

\[ \text{Pred}(S, \text{Mem}) = \text{Pred}(S, \text{Mem} \cup \{\text{Buf}\}) \]. \hspace{1cm} (1)

2) \( b = m \)

Every trace matching Buf over length m will also match S over length m (as they are identical over this length) so,
\text{Pred}(S, \text{Mem}) = \text{Pred}(\text{Buf}, \text{Mem}) .

Also, Buf provides another matching trace in Mem \cup \{Buf\}
so \; \text{Pred}(S, \text{Mem} \cup \{\text{Buf}\}) = \text{Pred}(S, \text{Mem}) \cup \{\text{Buf}(0)\} .

Combining these two equations with the premise that
\text{Buf}(0) \in \text{Pred}(\text{Buf}, \text{Mem})
gives \; \text{Pred}(S, \text{Mem} \cup \{\text{Buf}\}) = \text{Pred}(S, \text{Mem}) . \quad (2)

3) \; b > m

Bm(\text{Buf}, \text{Mem}) = m, \text{ for if not, the longer traces matching Buf
would also match } S, \text{ but this implies }
\text{Pred}(S, \text{Mem}) = \text{Pred}(\text{Buf}, \text{Mem}) .

Also, Buf will be the single longest matching trace of S in Mem \cup \{Buf\}
so,
\text{Pred}(S, \text{Mem} \cup \{\text{Buf}\}) = \{\text{Buf}(0)\} .

Combining these two equations with the premise that
\text{Buf}(0) \in \text{Pred}(\text{Buf}, \text{Mem})
implies
\text{Pred}(S, \text{Mem}) \supseteq \text{Pred}(S, \text{Mem} \cup \{\text{Buf}\}) = \{\text{Buf}(0)\} . \quad (3)

The three results (1), (2) and (3) constitute a proof of the theorem.

\textit{End of Proof}

\textit{II.A.2. Stability in Run-mode.}

When an FLM computer is executing in run-mode then the predictions for any particular trace will either remain unchanged or be a subset of its earlier predictions.
Corollary

Let Mem be an FLM memory at the start of one iteration of algorithm (I.A.3) in run mode and let Mem' be the memory at the start of the next iteration then for any trace S,

\[ \text{Pred}(S, \text{Mem}) \supseteq \text{Pred}(S, \text{Mem}') \]

Proof

The algorithm implies that the action chosen and stored as Buf(0) satisfies the condition of the last theorem that

\[ \text{Buf}(0) \in \text{Pred}(\text{Buf}, \text{Mem}) \]

or in the notation of the algorithm

\[ \text{Buf}(0) = a \in \text{Ch} \subseteq \text{Pred}(\text{Buf}, \text{Mem}) . \]

End of Proof

II.A.3. Invariance of Deterministic Predictions

If a trace predicts a single action then it will continue predicting that action so long as the memory is updated by predicted actions.

Corollary

For a trace Buf and set of traces Mem if

\[ \text{Buf}(0) \in \text{Pred}(\text{Buf}, \text{Mem}) \]

then for all traces S

\[ \text{Pred}(S, \text{Mem}) = \{a\} \]

implies

\[ \text{Pred}(S, \text{Mem} \cup \{\text{Buf}\}) = \{a\} . \]

Proof

\[ \text{Pred}(S, \text{Mem} \cup \{\text{Buf}\}) \neq \emptyset \text{ from the definition of Pred} \]

and

\[ \text{Pred}(S, \text{Mem} \cup \{\text{Buf}\}) \subseteq \text{Pred}(S, \text{Mem}) = \{a\} \]
so the only possible value for \text{Pred}(s, \text{Mem} \cup \{\text{Buf}\}) \text{ is } \{a\}.

\textbf{End of Proof}
Traces in FLM memory of order 2

A a P
B a Q
C a R
D a R New trace added (Buf)

The prediction of the buffer Buf is \{P,Q,R\} and the action chosen is R. (that is a=R).

<table>
<thead>
<tr>
<th>Case</th>
<th>S</th>
<th>b</th>
<th>m</th>
<th>Pred(S,Mem)</th>
<th>Pred(S,Mem ∪ {Buf})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) b&lt;m</td>
<td>A</td>
<td>a</td>
<td>1</td>
<td>2</td>
<td>{P}</td>
</tr>
<tr>
<td>2) b=m</td>
<td>B</td>
<td>a</td>
<td>1</td>
<td>1</td>
<td>{P,Q,R}</td>
</tr>
<tr>
<td>3) b&gt;m</td>
<td>D</td>
<td>a</td>
<td>2</td>
<td>1</td>
<td>{P,Q,R}</td>
</tr>
</tbody>
</table>

Fig. 8 Illustration of Cases Occurring in Theorem A.1
CHAPTER III
MULTIPLE FLM SYSTEMS

Systems combining more than one FLM have proved in practice to be more powerful than the single FLM computer described in Chapter I. For example, PURR-PUSS (Andreae, 1977) employs five different FLM-like memories. The tasks for which PURR-PUSS has been programmed would be impossible in practice without the greater flexibility supplied by multiple memories.

This chapter investigates multiple FLM systems in three steps. Firstly, a 'Dual-FLM' will be described and it will be used to provide solutions to the programming problems of 'coding a subroutine' and the 'conjunction of conditions' from the last chapter. Secondly, an abstract description of a class of multiple FLM systems will be given. It will be shown that this class includes the earlier Dual-FLM, that it is meaningful to consider fixed programs, and that none of these systems can solve the 'negation problem' (II.1.3) in its most general form. However, a multiple FLM computer will be exhibited which is capable of solving a simple form of the problem. Thirdly and finally, some variations on the MFLM will be considered which allow it to choose actions in a way which is closer to that used by PURR-PUSS.
Fig. 1 Structure of the Dual-FLM Computer
III.1. DESCRIPTION OF THE DUAL-FLM

The Dual-FLM is a computer which makes use of two FLMs with separate buffers and memories. These are referred to as the Main FLM and the Echo FLM. (See Fig. 1). The two FLMs receive as inputs different events. It is the independent predictions which the two FLMs can make from their different buffers that together make them a powerful system.

The way an event is seen by the two FLMs is determined by whether it is a pattern or whether it is one of three types of actions, namely ordinary, echo and mark actions. (Ordinary and echo actions are the same as those introduced in Chapters I and II).

Ordinary actions affect the external world and are predicted by the Main FLM. Echo and mark actions have the same external effect as the echoes introduced in the last chapter; they all return a pattern equal to themselves. Echoes are predicted only by the Main FLM but contribute to both the Main and Echo buffers. Marks are predicted only by the Echo FLM and, like echoes, contribute to both the Main and Echo buffers. However, marks are also predicted by the Main FLM in an indirect and special way. A special action, "x", is stored in the Main memory whenever a mark action occurs. It indicates that a mark action has been stored but leaves the Echo memory to remember which particular action it is.

In what follows, echoes will be distinguished by an initial # (E.G. #a, #b) and marks by an initial * (E.G. *m, *n).

The cycle of operations for the Dual-FLM computer is:

Choose the next action by using the predictions of the Echo and Main memories and possibly by accepting help from the
programmer;  
Transmit the chosen action to the external world and receive a pattern in return;  
Update the Main memory with the action and pattern;  
Update the Echo memory if the action was an echo or mark;  
Repeat the whole sequence from the beginning.

This cycle is very similar to that of the single memory computer. Only the details of the choice and update steps are more complex, to accommodate the two memories.

III.1.1. Updating

The updating part of the cycle is, in more detail:

Update and store a new trace in Main with the action just executed, except that if the action was a mark then use * rather than the actual action;

Update the Main buffer with the resultant pattern;

If the action executed was a mark then update the Echo buffer with the mark and store a trace;

If the action executed was an echo, update the Echo buffer.

As a result of this, the traces in the Main memory will consist of alternating actions and patterns (this is identical to the single FLM computer). The Echo memory traces will contain echoes and marks in any order, and are headed only by marks. Fig. 2 gives an example of a short sequence of events entering the Dual-FLM and the resultant traces stored in the Main and Echo memories.
<table>
<thead>
<tr>
<th>Events Received by Main FLM</th>
<th>Events Received by Echo FLM</th>
<th>Traces Stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>Action</td>
<td>Pattern</td>
<td>Events</td>
</tr>
<tr>
<td>A</td>
<td>a</td>
<td>#a</td>
</tr>
<tr>
<td>X</td>
<td>x</td>
<td>X x *</td>
</tr>
<tr>
<td>*</td>
<td>*m</td>
<td>X x *</td>
</tr>
<tr>
<td>A</td>
<td>b</td>
<td>*m A</td>
</tr>
<tr>
<td>#b</td>
<td>#b</td>
<td>A b #b</td>
</tr>
<tr>
<td>X</td>
<td>x</td>
<td>#b #b X</td>
</tr>
<tr>
<td>*</td>
<td>*n</td>
<td>X x *</td>
</tr>
<tr>
<td>B</td>
<td>a</td>
<td>* *n B</td>
</tr>
</tbody>
</table>

Fig. 2 Example of Updating of Dual FLM
III.1.2. Choosing an Action

The action chosen for execution by the Dual-FLM will either be taken from the combined prediction of the Echo and Main memories (see below) or accepted from the programmer. As in the single FLM the programmer is asked for an action in run-mode only if the current prediction includes all possible actions. In program mode the programmer may choose any action he wishes.

The prediction is made as follows:

If the prediction set of the Main FLM includes the special action * then the dual prediction is the union of the Echo prediction and the Main prediction less the *.

If the prediction set of the Main FLM does not include * then it forms the dual prediction itself.

Finally, the Dual-FLM chooses an action from amongst those predicted.

Fig. 3 gives examples of the operation of this rule.

<table>
<thead>
<tr>
<th>Individual FLM Predictions</th>
<th>Dual-FLM Predictions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main</td>
<td>Echo</td>
</tr>
<tr>
<td>{[*]}</td>
<td>{[*m]}</td>
</tr>
<tr>
<td>{[*],#a}</td>
<td>{[*m]}</td>
</tr>
<tr>
<td>{[*]}</td>
<td>Ma</td>
</tr>
<tr>
<td>[#a,A}</td>
<td>{[*m]}</td>
</tr>
<tr>
<td>{*[}</td>
<td>{[*m,*n]}</td>
</tr>
<tr>
<td>{A,B}</td>
<td>Ma</td>
</tr>
</tbody>
</table>

Note: Ma is the set of all possible marks.

**Fig. 3 Examples of Prediction by Dual-FLM**
III.2. PROGRAMMING

Having described the operation of the Dual-FLM, I will now demonstrate some programming techniques for it. I will first introduce a technique for forming the conjunction of conditions. This same technique can be used to form subroutines. I will illustrate how this technique can avoid the restriction of II.1.4 where the sequences of events within a subroutine had to be less than one buffer in length. The excessive amount of programming needed with this earlier technique is also avoided.

III.2.1. Conjunction

I will now return to the problem of forming the conjunction of two conditions which was solved using echo actions in (II.1.2). In the FLM the problem of forming a conjunction reduces to that of separating sequences of events such as:

\[
\ldots C \ c \ D \ d \ E \ e \ A \ldots \\
\ldots C \ b \ D \ d \ E \ e \ B \ldots
\]

The problem becomes non-trivial when the buffer-length is insufficiently long to bridge the common segment in the middle and the FLM is unable to remember c or b when A or B is to be chosen. These two sequences can be recoded in the Dual-FLM as follows:

\[
\ldots C \ c \ #c \ #c \ D \ d \ E \ e \ *m \ #m \ A \ldots \\
\ldots C \ b \ #b \ #b \ D \ d \ E \ e \ *n \ #n \ B \ldots
\]

To see how this enables the Dual-FLM to separate the two cases see Fig. 4. This lists the traces generated by the two sequences and, as well, shows the predictions and decisions of the Dual-FLM when the
sequences occur in run mode. The echoes $c$ and $b$ record in the echo buffer which pattern (c or b) has occurred after the action C. When the critical choice is to be made between the actions A and B the echo FLM is able to provide a unique and appropriate mark ($m$ or $n$) because its buffer contains only the last echo ($c$ or $b$) which has been preserved, undisturbed by the intervening actions. The mark then appears in the Main FLM buffer which permits the action A or B to be chosen unambiguously.

When compared with the single FLM technique of (Ch. II.1.2) this method is less efficient as two extra actions, a mark and an echo instead of one echo, are required for each of the sequences. However, in the Dual-FLM case, only two echoes are needed, independent of the number of intervening actions whereas for the single FLM the echo must be rehearsed at least once every buffer length to maintain the needed information.
(a) Sequences to be Distinguished.

\[ \ldots C \ c \ D \ d \ E \ e \ A \ \ldots \]
\[ \ldots C \ b \ D \ d \ E \ e \ B \ \ldots \]

(b) Dual-FLM Solution.

\[ \ldots C \ c \ #c \ D \ d \ E \ e \ *m \ *m \ A \ \ldots \]
\[ \ldots C \ b \ #b \ D \ d \ E \ e \ *n \ *n \ B \ \ldots \]

(c) Traces Generated by Programming.

<table>
<thead>
<tr>
<th>Main (order=4)</th>
<th>Echo (order=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-) C c #c</td>
<td>#c #m</td>
</tr>
<tr>
<td>C c #c #c D</td>
<td></td>
</tr>
<tr>
<td>#c #c D d E</td>
<td>#b *n</td>
</tr>
<tr>
<td>D d E e *</td>
<td></td>
</tr>
<tr>
<td>E e *m *m A</td>
<td></td>
</tr>
<tr>
<td>- - C b #b</td>
<td></td>
</tr>
<tr>
<td>C b #b #b D</td>
<td></td>
</tr>
<tr>
<td>#b #b D d E</td>
<td></td>
</tr>
<tr>
<td>D d E e *</td>
<td></td>
</tr>
<tr>
<td>E e *n *n B</td>
<td></td>
</tr>
</tbody>
</table>

(d) Execution in Run-Mode.

(\(-\) indicates the value of the prediction does not matter)

<table>
<thead>
<tr>
<th>Events</th>
<th>Prediction</th>
<th>Action Pattern</th>
<th>Action Chosen</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>#c</td>
<td>c #c</td>
<td>#c</td>
</tr>
<tr>
<td>#c</td>
<td>#c</td>
<td>D #m</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>d</td>
<td>E #m</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>e</td>
<td>*m #m</td>
<td>*m</td>
</tr>
<tr>
<td>--&gt;*m</td>
<td>*m</td>
<td>A #m</td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>#b</td>
<td>b #b</td>
<td>#b</td>
</tr>
<tr>
<td>#b</td>
<td>#b</td>
<td>D #n</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>d</td>
<td>E #n</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>e</td>
<td>*n #n</td>
<td>*n</td>
</tr>
<tr>
<td>--&gt;*n</td>
<td>*n</td>
<td>B #n</td>
<td>B</td>
</tr>
</tbody>
</table>

Fig. 4 Coding of Conjunction using Dual-FLM
Fig. 5 compares the generated sequences of events for the two techniques when a long gap separates the initial condition and the final action. As can be seen, for longer sequences the single FLM uses many more echoes.

It should be noted also that the Dual-FLM is not debarred from using the same techniques as the single FLM, and can rely on the main buffer alone to provide the appropriate context. So both techniques might be used depending on the number of events to be bridged.

Sequences to be Separated

..... C b D d E e F f G g H h .... Z z B ....
..... C c D d E e F f G g H h .... Z z A ....

Dual-FLM Solution

..... C b #b #b E e F f G g H h .... Z z ^p ^p B ....
..... C c #a #a E e F f G g H h .... Z z ^m ^m A ....

Single FLM Solution (Order=4)

...C b E e #1 #1 F f #1 #1 G g #1 #1 H h #1 #1 ... #1 #1 Z z B ....
...C c E e #2 #2 F f #2 #2 G g #2 #2 H h #2 #2 ... #2 #2 Z z A ....

Fig. 5 Coding of Conjunction Using Dual-FLM Compared With Single Memory FLM
III.2.2. **Subroutines**

I will now extend the technique of the last section to include the coding of subroutines. Fig. 6 shows the same subroutine as that considered in Fig.II.2 and in II.1.4. It shows the two sequences needed by the Dual-FLM to program the subroutine and the two calls to it. It also shows the traces generated as a result of this programming. The bottom of the figure shows two paths through the subroutine and the predictions of the Dual-FLM at each step. Neither of these combinations of internal and external paths has occurred during the programming and yet both are correctly executed.

As in the last section, an echo is used to separate the different ways of entering the subroutine (different calls to it). Then at the exit of the subroutine this echo is present in the echo buffer, and enables a different mark to be chosen for each route out of the subroutine. These separate marks then provide sufficient context in the main buffer for each different route to be resumed.

This technique has two main advantages over the earlier solution. The first advantage is that a path within the subroutine can be of any length; that is, it is not restricted by the length of the Main buffer. The second advantage is the reduction in the amount of programming needed to record the subroutine and the calls to it.

It is easily seen that any number of actions other than marks and echoes can occur between entering and leaving the subroutine, as during any such sequence the Echo buffer remains undisturbed. If echoes or marks are used within the subroutine, however, a little more care is needed as the information within the Echo buffer may be destroyed. If the Echo buffer is of length N, then at most N-1 echoes can be used
before leaving the subroutine.

In particular this limits the total number of calls to subroutines which can be made from within a subroutine and by implication the level of nesting which can be achieved.

It will be recalled from Ch.II.1.4 that in coding a subroutine with \( p \) paths into and out of it and \( q \) possible sequences within it, \( p \) times \( q \) different passes through the subroutine would be necessary to program it. In the example of Fig. 6 that would be four (two times two) sequences. But, as can be seen, using the Dual-FLM only two of these possible combinations are necessary to completely code the subroutine. The other two combinations of internal and external paths form the example at the bottom of Fig. 6, and, as can be seen, they are executed correctly. In general it is possible to completely code a subroutine when the number of passes through it is limited to the maximum of \( p \) and \( q \). (This is ensured by traversing a different sequence within the subroutine each time a different 'call' to the subroutine is programmed.)
(a) Effect to be Achieved

\[ \begin{array}{cccccc}
A & a & B & b & C & D & d \\
& & & & C & s & T \\
& & & & E & e & \end{array} \]

\[ \begin{array}{cccccc}
X & x & Y & y & & \\
Z & z & & & & \end{array} \]

(b) Dual-FLM Solution

\[ \begin{array}{cccccc}
A & a & B & b & #a & #a \\
& & & & C & D & d \\
& & & & C & s & T \\
& & & & *m & *w & E & e \\
X & x & Y & y & #x & #x & *x & Z & z \end{array} \]

(c) Two Possible Sequences to Program Subroutine

... A a B b #a #a C c D d *m *m E e ...
...
X x Y y #x #x C s T t *x *x Z z ...

(d) Execution in Run mode of Two Paths Through Subroutine
(Note: The paths below have not explicitly occurred in the programming above)

<table>
<thead>
<tr>
<th>Action</th>
<th>Events Pattern</th>
<th>Echo</th>
<th>Predictions Main</th>
<th>Echo</th>
<th>Action Chosen</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>a</td>
<td></td>
<td>[B]</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>b</td>
<td></td>
<td>[#a]</td>
<td></td>
<td>#a</td>
</tr>
<tr>
<td>#a</td>
<td>#a</td>
<td>#a</td>
<td>[C]</td>
<td>[*a]</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>s</td>
<td></td>
<td>[T]</td>
<td>[*a]</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>t</td>
<td></td>
<td>[*]</td>
<td>[*a]</td>
<td>*m</td>
</tr>
<tr>
<td>*m&lt;----</td>
<td></td>
<td>*m</td>
<td>[E]</td>
<td></td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>x</td>
<td>x</td>
<td>[Y]</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>Y</td>
<td>y</td>
<td></td>
<td>[#x]</td>
<td></td>
<td>#x</td>
</tr>
<tr>
<td>#x</td>
<td>#x</td>
<td>#x</td>
<td>[C]</td>
<td>[*x]</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>c</td>
<td></td>
<td>[D]</td>
<td>[*x]</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>d</td>
<td></td>
<td>[*]</td>
<td>[*x]</td>
<td>*x</td>
</tr>
<tr>
<td>*x&lt;----</td>
<td></td>
<td>*x</td>
<td>[Z]</td>
<td></td>
<td>Z</td>
</tr>
</tbody>
</table>

Fig. 6 Programming a Subroutine

90
III.3. MULTIPLE FLMs

This section informally introduces the MFLMs, a class of computers involving Multiple FLMs which includes the Dual-FLM. A precise and formal description is given in A.1.

The individual FLMs in an MFLM store traces and make predictions using the same algorithms as for the single FLM. To tie them together into an integrated system it is necessary to decide which FLMs will receive which events and also how to combine the individual predictions of the FLMs into a single composite prediction from which an action can be chosen.

III.3.1. Update Functions

The mechanism for deciding whether an event will be seen by a particular FLM is provided by three partial functions, associated with that FLM, called the update functions. (A partial function is one which may be undefined for certain of its possible arguments. It yields a single well defined result for all other possible arguments.) Each of the three update functions may have as an argument any event which can be seen by the MFLM. If the result they return is undefined then the particular event is not seen by that FLM (although it may be seen by other FLMs in the MFLM). Otherwise the result of the function will be an event seen by the FLM. The three functions are written PR (prediction), AC (action) and PA (pattern) with appropriate subscripts to distinguish them for particular FLMs.

The PR function is applied only to actions and determines whether they will be predicted (that is placed at the head of traces).
or not. Consider, for example, an ordinary action, say A, seen by the Dual-FLM. The main memory is updated by a trace with A as its head. So the value of PRm(A) (the prediction update function for the main memory) is just A. However, the ordinary action is ignored by the echo memory so PRe(A) (the prediction update function for the Echo memory) is undefined. For a mark action, say *m, the values are:

\[
\begin{align*}
PRm(*m) &= * \\
PRe(*m) &= *m
\end{align*}
\]

That is; main is updated by a trace with * at its head and echo by one with *m at its head.

The AC update function is similarly applied only to actions and determines whether they will be included in the tail of the buffer trace or not. For example ACe is well defined for echo actions whereas PRe remains undefined.

The PA function is applied only to incoming patterns and determines whether or not they will be included in the tail of the FLM buffer.

By judicious use of these functions one can achieve all the types of parallel context mentioned at the end of Chapter 3 of (Andreea, 1977).

### III.3.2. Global Prediction

To obtain a single global prediction from all the constituent FLMs, a function P is defined. This takes the sets of events predicted by each FLM and gives as a result a single set of events. For the dual FLM the function P is given in Fig. 7. P(M,E) includes the echo prediction E only if the Main prediction, M, includes the mark "*".
(See A.2 for a formal description of the Dual-FLM as an MFLM.)

Update Functions

<table>
<thead>
<tr>
<th>e</th>
<th></th>
<th>Main FLM</th>
<th></th>
<th>Echo FLM</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>*a</td>
<td>*</td>
<td>*</td>
<td></td>
<td>*a</td>
<td>*a</td>
</tr>
<tr>
<td>#a</td>
<td>#a</td>
<td>#a</td>
<td></td>
<td>-</td>
<td>#a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th>PAm</th>
<th></th>
<th></th>
<th>PAe</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>*a</td>
<td>*a</td>
<td>-</td>
<td></td>
<td>*a</td>
<td></td>
</tr>
<tr>
<td>#a</td>
<td>#a</td>
<td></td>
<td></td>
<td>#a</td>
<td></td>
</tr>
</tbody>
</table>

Note: - indicates function undefined.

Global Prediction Function

\[ P(M,E) = \begin{cases} * \in M & \text{then } M - \{*\} \cup E \\ \text{else } M \end{cases} \]

where: M is the set predicted by Main FLM; E is the set predicted by Echo FLM.

**Fig. 7 Description of Dual-FLM as an MFLM**
III.3.3. MFLM Computer Algorithm

The algorithm which regulates the execution of an MFLM has the same basic structure as that for the single FLM computer and the Dual-FLM. Informally the algorithm is:

Obtain predictions from each of the individual FLMs and then calculate a global prediction using P;

Choose the next action using the global prediction and possibly by accepting help from the programmer;

Transmit the chosen action to the external world and receive a pattern in return;

For each FLM in the MFLM:
   if PR is defined for the action then update the buffer-head of the FLM with PR(action) and store the trace;
   if AC is defined for the action then update the buffer of the FLM with AC(action);
   if PA is defined for the pattern then update the buffer of the FLM with PA(pattern).

Repeat the whole sequence from the beginning.

III.3.4. Conditions on MFLM

In order that this algorithm should work correctly it is necessary to impose three conditions on the prediction function P.

i) For an action to be chosen in the second step of the algorithm above it is necessary that P always delivers a non-empty set. (This is similar to the condition on the single FLM memory where it yields a prediction of all possible events when no trace in the memory matches.) I will refer to this as the condition that P be non-empty.

ii) The next two conditions must be applied so that the MFLM will be sufficiently well behaved that a static representation of it can
be constructed. To do this it is necessary to ensure, in run mode, that later predictions are subsets of earlier ones just as was done for the single FLM. If this is true of the individual FLMs then it must still be true of the overall prediction. Therefore P must yield a prediction which is a subset or a superset depending on whether the individual predictions are subsets or supersets. For example, in an MFLM with two constituent FLMs the condition is:

\[ \text{if } A \supseteq A' \text{ and } B \supseteq B' \text{ then } P(A,B) \supseteq P(A',B') \]

I will refer to this as the condition that P be monotonic.

iii) To ensure that an individual FLM behaves correctly as mentioned above it must be updated with a new action only when the action is a part of the current prediction by that FLM. The FLM might be updated with spurious actions if it had been independently predicted by a different FLM. Also the function P might be 'pathological' and predict actions totally unrelated to those predicted by the individual FLMs. To forestall these possibilities any action a which is predicted by P must also be predicted by any FLM which will be updated with a. That is; if PR(a) is well defined for an FLM then PR(a) must occur in the prediction of the FLM. I will refer to this as the condition that P be reversible (it must correctly reverse the PR functions).

A.2 includes a proof that the Dual-FLM satisfies these three conditions.
III.3.5. Execution Time

The execution time for each iteration of the MFLM algorithm will be the sum of the update and prediction times of the individual FLMs together with any manipulation required for constructing the global prediction and choosing an action. As a result an MFLM will be able to take advantage of the low execution times inherent in the FLM design.

III.3.6. Static Representation of an MFLM

In the last chapter I showed that it was meaningful to speak of a fixed program in the FLM even though the actual contents of the memory were changing. This was done by showing that a frozen version of the FLM memory acted as a model of the following behaviour of the FLM, in particular it completely specified any deterministic steps.

The MFLM memories may also change with time and so it is necessary to find some static model of their behaviour. This can be done, as for the FLM, by freezing the contents of all the memories in the MFLM and using this as a model. That this model is valid is shown in Theorem A.3. This proves that if, when a prediction is obtained from the MFLM the same traces are matched against the earlier frozen version, then the prediction from the current MFLM will be a subset of that from the frozen MFLM.

The theorem is proven in two steps. It is first shown, using the condition of reversibility, that a trace will be added to the individual FLM memories, making up the MFLM, only when the head of the trace is predicted by the FLM. This implies that Theorem A.1 of Chapter II applies, so the current predictions of the individual FLMs will be
subsets of the predictions of the individual frozen FLMs. Using the condition of monotonicity this immediately implies the result for the global MFLM prediction.

III.4. NEGATION REVISITED

In this section I will show that the general form of the negation problem cannot be solved by any FLM system. However, a simpler form of the problem can be solved and an MFLM will be exhibited which does just this.

III.4.1. General Negation Problem

To recapitulate (from II.1.3), the problem to be solved is:

if the sequence of events \( \ldots C \circ D \) occurs then \( A \) is to be executed;

if any pattern other than \( a \) occurs after the \( D \) then \( Z \) is to be executed;

To provide a realistic problem for the MFLM it will be assumed, for any two patterns, \( a \) and \( x \) say, occurring after \( D \), that for each \( PA_i, 1 \leq i \leq m \)

\[
\begin{align*}
\text{either } PA_i(x) & \text{ and } PA_i(a) \text{ are both undefined} \\
& \text{ or } PA_i(x) = PA_i(a) \\
& \text{ or } a \neq x.
\end{align*}
\]

These conditions imply that the MFLM cannot have an inbuilt mechanism which knows about the difference between "a" and "x" without any programming. For example it cannot see a pattern which is present precisely when "a" is absent.
Consider first a situation where \( x \), which has never occurred previously, occurs for the first time following \( D \). The prediction by \( P \) will be just \( \{Z\} \). Let the FLM buffers for this case be \( \text{Buf}_1, \ldots, \text{Buf}_m \).

In the second situation, \( a \) occurs and \( \{A\} \) will be the prediction by \( P \). Let the FLM buffers for this case be \( \text{Buf}_1', \ldots, \text{Buf}_m' \). Assume that apart from a occurring in place of \( x \) that the buffers are identical to the first case.

Taking each FLM in turn

either \( \text{PAi}(x) \) and \( \text{PAi}(a) \) are both undefined, so they will make no contribution to the buffer, so \( \text{Buf}_i = \text{Buf}_i' \);

or \( \text{PAi}(x) = \text{PAi}(a) \) so \( \text{Buf}_i = \text{Buf}_i' \) anyway;

or \( \text{PAi}(x) \neq \text{PAi}(a) \) and because \( \text{Buf}_i'(1) = \text{PAi}(x) \) and \( \text{Buf}_i'(1) = \text{PAi}(a) \)

\( \text{Buf}_i \neq \text{Buf}_i' \).

In this third situation \( x \) (and by implication \( \text{PAi}(x) \)) will never have occurred previously, and so the buffer will match nothing in the memory. So all possible events will be predicted, that is

\( \text{Pred}(\text{Buf}_i,\text{Mem}_i) = \text{Ev}_i \).

It will be trivially true then that

\( \text{Pred}(\text{Buf}_i',\text{Mem}_i') \subseteq \text{Pred}(\text{Buf}_i,\text{Mem}_i) \)

This will be true then for all the FLMs (as in the two other cases \( \text{Buf}_i' = \text{Buf}_i \)). Also, \( P \) is monotonic so

\( P(\text{Pred}(\text{Buf}_1',\text{Mem}_1'),\ldots,\text{Pred}(\text{Buf}_m',\text{Mem}_m')) \subseteq P(\text{Pred}(\text{Buf}_1,\text{Mem}_1),\ldots,\text{Pred}(\text{Buf}_m,\text{Mem}_m)) = \{Z\} \)

So the only prediction possible when a occurs is \( \{Z\} \)! It is impossible for the MFILM to distinguish \( a \) as a separate case.
III.4.2. Simple Negation Problem

The analysis above does not exclude the possibility of programming the simpler form of the negation problem where the pattern following D is ignored and Z is always predicted as the next action. An MFLM containing two FLMs will now be constructed, and a solution to this problem obtained using it.

Fig. 8 gives the update functions and global prediction function for the two FLM systems to be used. The events for the system include echo actions (E) ordinary actions (A) and their resultant patterns (P). The two FLMs are the Action FLM and the Main FLM. The Action FLM sees only echo actions and ordinary actions, no patterns at all. (It is this lack of patterns that will enable it to ignore the pattern following D and predict Z.) The Main FLM sees all actions (echo and ordinary) as well as all patterns. To permit the detailed selection of ordinary actions by the Main FLM, the Action FLM stores ordinary actions as "!". Only when "!" is predicted by the Action FLM is the prediction by the Main FLM included in the global prediction. In this way the Action FLM regulates when an ordinary action is permissible, but not which one will, in fact, be predicted. ("!" is used here in much the same way that "*" is in the Dual-FLM.)

Note that the patterns returned by echo actions are ignored, as they provide no additional information not already provided by the action. Also, the full form of ordinary actions is recorded in the Action FLM buffer rather than the special action "!", thus providing more specific contexts.

Fig. 9 shows a sequence in program mode where the computer is programmed to ignore the pattern following D and to execute a Z. This
is done by introducing two echo actions around the D. These carry the system past any pattern after D using the Action FLM buffer. The programming is verified by a sequence in run-mode where the unseen pattern x occurs after D and the action Z is correctly selected.

It may be noted that the function P is very similar to that for the Dual-FLM. It is easily verified that it obeys the three conditions of being non-empty, monotonic and reversible.
### Update Functions

<table>
<thead>
<tr>
<th></th>
<th>Action FLM</th>
<th>Main FLM</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>PRA</td>
<td>PRm</td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>#a</td>
<td>#a</td>
<td>#a</td>
</tr>
<tr>
<td>P</td>
<td>AA</td>
<td>-</td>
</tr>
<tr>
<td>#a</td>
<td>-</td>
<td>P</td>
</tr>
</tbody>
</table>

### Prediction Function

\[
P(A,M) = \begin{cases} 
\text{if } \not\in A \text{ then } (A-\{!\}) \cup M \\
\text{else } A 
\end{cases}
\]

where: \(A\) is the prediction set of the Action FLM.

\(M\) is the prediction set of the Main FLM.

**Fig. 8 MFLM for Solution of Simple Negation Problem**
<table>
<thead>
<tr>
<th>Events</th>
<th>Action FLM(order=2)</th>
<th>Main FLM(order=2)</th>
<th>Global Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Action Pattern</td>
<td>Prediction</td>
<td>Trace</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>program mode</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>A B !</td>
<td>B b C</td>
</tr>
<tr>
<td>B</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#a</td>
<td>#a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#b</td>
<td>#b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>run mode</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>b</td>
<td>{!}</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>c</td>
<td>(#a)</td>
<td></td>
</tr>
<tr>
<td>#a</td>
<td>#a</td>
<td>{!}</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#b</td>
<td>#b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

Fig. 9 Solution to Simple Negation Problem
III.4.3. Alternative Storage Scheme for Negation

It may be better appreciated why the MFLMs cannot solve the general negation problem by considering an FLM like system which, at least superficially, can solve the problem.

Consider a system containing two FLMs which receive the same events but which have buffers of order 1 and 2 respectively. Their composite prediction is taken from the order 2 FLM if it has a full match and from the order 1 FLM otherwise. Now if the memories contain only the following two traces which match #b:

\[ a \quad #b \quad A \]

then whenever "a" occurs after "D" (using the problem illustrated in Fig. 9) "A" will be predicted and executed and whenever anything else occurs "Z" will be executed. This would effectively solve the negation problem. However, the FLM cannot generate such memories because whenever a #b A is stored #b A must also be stored in the order 1 FLM thus destroying the distinction between the two situations.

It is not clear whether it is possible to setup a learning algorithm which could reliably generate and maintain such pairs of traces. However, the suggestion in 5.4 that a memory only be updated when its prediction agrees with the action actually chosen might provide such a mechanism.
III.5. CONSEQUENCES OF RESTRICTIONS ON MFLM

III.5.1. Example of Invalid MFLM

An intuitively appealing and simple strategy for choosing actions in an MFLM is to choose first those which are predicted by all the FLMs or, if there are none, those which are predicted by all the FLMs bar one and so on. This type of procedure is used for example in PURR-PUSS, see (Andreae, 1977, Chapter 5).

Fig. 10 defines a simple MFLM incorporating two FLMs; an Action FLM and a Pattern FLM. Both predict all actions; however, the buffer of the Action FLM contains only actions, and the buffer of the Pattern FLM only patterns. The global prediction function (as suggested above) first predicts any actions which are predicted by both the action and pattern FLMs, or, if there are none of these, those predicted by either of them.

The global prediction function as specified is neither monotonic nor reversible. It is not monotonic because
\[ P([A,B], [B,C]) = \{B\} \]
and
\[ P([A], [C]) = \{A,C\} \]
whereas the second expression should be a subset of the first.

It is not reversible because,
\[ C \in \{A,C\} = P([A], [C]) \]
but \( P[Ra(C)] = C \notin \{A\} \)
that is, \( C \) should be predicted by the Action FLM but is not.

It might be that these conditions on the prediction function are too harsh and in practice no difficulty is encountered. Fig. 11 shows, however, the difficulties that can arise with this particular MFLM.
This shows how the sequence W w X x which was originally programmed to be followed only by A can, after a period in run mode, be followed by the action C. (The two crucial steps are arrowed.) This inadvertent reprogramming is a direct result of the lack of monotonicity and reversibility of P.

### Update Functions

<table>
<thead>
<tr>
<th>e</th>
<th>Action FLM</th>
<th>Pattern FLM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PRa</td>
<td>PAa</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>p</td>
<td>-</td>
<td>PAp</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Prediction Function

\[
P(A,B) = \begin{cases} 
A \cap B \neq \emptyset & \text{then } A \cap B \\
\text{else } A \cup B; 
\end{cases}
\]

where: A is the prediction set of the Action FLM; B is the prediction set of the Pattern FLM.

**Fig. 10 Example of an Invalid MFLM**
<table>
<thead>
<tr>
<th>Action Pattern</th>
<th>Action FLM Prediction</th>
<th>Pattern FLM Prediction</th>
<th>Global Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>program mode</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>x</td>
<td>Ev(_a)</td>
<td>Ev(_p)</td>
</tr>
<tr>
<td>A</td>
<td>W X A</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>in run mode</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>x</td>
<td>[C]</td>
<td>[A]</td>
</tr>
<tr>
<td>C</td>
<td>A B C</td>
<td>w x C</td>
<td>{A,C}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>b</td>
<td>[A]</td>
<td>[C]</td>
</tr>
<tr>
<td>C</td>
<td>W X C</td>
<td>a b C</td>
<td>{A,C}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>x</td>
<td>{A,C}</td>
<td>{A,C}</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>{A,C}</td>
</tr>
</tbody>
</table>

--- critical steps in sequence
Ev\(_a\) set of all events seen by Action FLM.
Ev\(_p\) set of all events seen by Pattern FLM.

**Fig. 11** Example of 'Re-programming' in Run-mode
III.5.2. Possible Alternative

In formulating the design of the MFLM I have been guided by the need for a system which can be formally described and analyzed. This has led to a family of systems which can solve some simple programming problems but which cannot use some of the techniques used by PURR-PUSS. The design of PURR-PUSS has proceeded, however, on much more pragmatic grounds than those described here (see (Andreea, 1977, Chapter 5)), emphasising the 'teachableness' of the system, its tendency not to enter endless loops, and its ability to 'reach' any stored experience again. At the very least, comparisons of PURR-PUSS with the more rigid and formal MFLM should lead to a better understanding of the extent to which PURR-PUSS must diverge from a clean formal scheme to attain her design goals. More satisfying from the theoretician's point of view would be to construct a system including both the MFLMs good formal behaviour and PURR-PUSS's pragmatic design goals.

To some, the least obvious of the three constraints on the MFLM will be that of reversibility, which ensures that any action chosen for execution will have been predicted by each of the FLMs which are updated by the action. It is possible to remove this restriction by only updating an individual FLM if the action chosen has in fact been predicted by it. Using the MFLM of Fig. 10 the action A can be chosen when Fa predicts {A} and Fp predicts {B}. However, using the alternate update only Fa and not Fp will have a trace stored.

Making this alteration to the MFLM algorithm and removing the constraint of reversibility leads to a curious possibility. The prediction algorithms of Fig. 10 can be replaced by

\[ P'(A, B) = A \lor B, \]

107
which is non-empty and monotonic. Now the algorithm for the MFLM (and
indeed for the single FLM computer) specifies that after the prediction
of a possible action has been made, then an actual action must be
chosen from this prediction set. No restriction is placed on the means
by which this choosing is done. Any information can be used, including
whether or not the action has been predicted by both FLMs. That is, an
action can be chosen from the actions calculated by a 'choice'
function, say C, where:

\[
C(A,B) = \begin{cases} 
A \cap B \neq \emptyset & \text{then } A \cap B, \\
\text{else } A \cup B;
\end{cases}
\]

As \( C(A,B) \subseteq P'(A,B) \) such a choice would be consistent with the
algorithm. (Fig. 12 shows how this modified MFLM would handle the
sequence of Fig. 11. Differences from Fig. 11 are underlined.)

This seems somewhat paradoxical as a function has been used for
choosing actions which cannot be used for making predictions. The
resolution of the paradox lies in the fact that the prediction function
is used in the analysis of what has been programmed into the FLM. The
choice function can at most have pragmatic worth as any behaviour or
action selection which arises as a result of it alone cannot be
guaranteed to be repeatable. In practice, it may still be useful,
however, to include such a choice function even if it cannot be
guaranteed.

It may be possible, then, to build a PURR-PUSS like system using
both a valid global prediction function and a more effective choice
function. Using the prediction functions one can analyze what one is
sure has been correctly and repeatably programmed. The choice function
would then provide practically useful, if not guaranteed, behaviour.
<table>
<thead>
<tr>
<th>Action Pattern</th>
<th>Action FLM</th>
<th>Pattern FLM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>program mode</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>w</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>x</td>
<td>Ev&lt;sub&gt;a&lt;/sub&gt;</td>
</tr>
<tr>
<td>A</td>
<td>W X A</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>in run mode</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>w</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>x</td>
<td>{C}</td>
</tr>
<tr>
<td>C</td>
<td>A B C</td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>a</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>b</td>
<td>{A}</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>a b C</td>
</tr>
<tr>
<td>W</td>
<td>w</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>x</td>
<td>{A}</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: ___ indicates a difference from Fig. 11.

**Fig. 12 Example Using Modified MFPM Algorithm.**
III.6. SUMMARY

This chapter has provided an initial theory of computers containing many FLMs. The extra power obtained by using multiple FLMs has been demonstrated by two different systems which have provided solutions to some of the programming problems raised in Chapter II.

Building on the theoretical base provided by Chapter II, constraints on the way in which predictions may be extracted from such systems have been developed. A condition was imposed that once a program had been entered into the computer it should continue executing correctly. One way of meeting this condition was to ensure that the global prediction function for the whole MFLM was non-empty, monotonic and reversible. These conditions, however, exclude an intuitively appealing technique used with some success in FURR-PUSS. This technique can be reinstated, however, if the condition of reversibility is lifted and it is simultaneously ensured that an individual FLM only stores a new trace when it agrees with the FLM's current prediction.

The importance of this chapter rests not so much on the particular form and abilities of the MFLMs, as on the fact that a beginning has been made on extracting some order from the chaos of possibilities which present themselves when designing multiple FLM systems.
III.A.1. Definition of an MFLM

III.A.1.1. Constituents

This section gives a formal definition of the structure of an MFLM. An MFLM contains the following elements:

For some \( m, m \geq 1 \), and each \( i, 1 \leq i \leq m \):
Fi an FLM where \( Fi = (Ev_i, Buf_i, Mem_i) \), (see I.A.1);

Three partial functions (the *updating* functions)
\[ PR_i : Ev \rightarrow Ev_i \]
\[ AC_i : Ev \rightarrow Ev_i \]
\[ PA_i : Ev \rightarrow Ev_i \]

a non-empty set of *events* \( Ev \);
a non-empty set of *actions* \( Ac \);
a non-empty set of *patterns* \( Pa \), where \( Ev = Pa \cup Ac \).

and a global prediction function \( P \) where:
\[ Ev, Ev_1, Ev_2, \ldots Ev_n \rightarrow Ev \]
\[ P : 2 \times 2 \times \ldots \times 2 \rightarrow 2. \]

The following three conditions apply to \( P \):

i) non-empty
\[ A_i^0, 1 \leq i \leq m \]
implies \( P(A_1, \ldots A_m) \neq 0. \)

ii) reversible
If \( e \in P(A_1, \ldots A_m) \)
and \( PR_i(e) \) is well defined
then \( PR_i(e) \in A_i, 1 \leq i \leq m. \)

iii) monotonic
\[ A_i \geq A_i', 1 \leq i \leq m \]
implies \( P(A_1, \ldots A_m) \geq P(A_1', \ldots A_m'). \)
III.A.1.2. MFLM Computer Algorithm

This section defines the computer algorithm obeyed by the MFLM. This assumes that the following functions and routines are defined as in I.A:

- Execute
- Accept
- Choose
- Buffer-update
- Update

The computer can operate in either program mode or run mode.

computer-loop:
\[
\begin{align*}
\text{For } i & := 1 \text{ to } m \text{ do} \\
\text{Pr}[i] & := \text{Pred(Buf}^i, \text{ Mem}) ; \quad \{\text{obtain individual predictions}\} \\
\text{Ch} & := P(\text{Pr}[1],\text{Pr}[2], \ldots ,\text{Pr}[m]) ; \quad \{\text{obtain global prediction}\} \\
\text{if} \text{ in-program-mode or } \text{Ch} \supseteq \text{Ac} & \quad \{\text{choose an action and}\} \\
\text{then} \text{ Accept (Ch)} ; & \\
\text{A} & := \text{Choose (Ch)} ; \\
p & := \text{Execute (A)} ; & \quad \{\text{obtain result}\} \\
\text{update memories and buffers} & \\
\text{For } i & := 1 \text{ to } m \text{ do} \\
\text{Begin} & \\
\text{At} & := \text{PRI (A)} ; \quad \{\text{action prediction}\} \\
\text{if} \text{ At} & \neq \text{null} \\
\text{then} \text{ Update (Buf}^i, \text{ Mem}^i, \text{ At)} ; & \\
\text{At} & := \text{ACi (A)} ; \quad \{\text{action in buffer}\} \\
\text{if} \text{ At} & \neq \text{null} \\
\text{then} \text{ Buffer-Update (Buf}^i, \text{ At)} ; & \\
\text{pt} & := \text{PAi (p)} ; \quad \{\text{pattern}\} \\
\text{if} \text{ pt} & \neq \text{null} \\
\text{then} \text{ Buffer-Update (Buf}^i, \text{ pt)} ; & \\
\text{End} ; \\
\text{goto computer-loop} ;
\end{align*}
\]

Note: The special value null has been used to indicate that the value of a function is undefined.
III.A.2. Dual-FLM

III.A.2.1. Construction

This section gives a formal construction of the Dual-FLM as a multiple FLM system.

The set of actions (Ac) is split into three non-empty disjoint sets, Ma (marks), Ec (echoes), Or (ordinary actions) that is:

\[ Ac = Ma \cup Ec \cup Or \]

The system comprises two FLMs designated Fm(main), Fe(echo), where:

\[ Fm = (Pa \cup Ec \cup Or \cup \{^*\}, Buf_m, Mem_m) \]
\[ Fe = (Pa \cup Ma, Buf_e, Mem_e) . \]

The updating functions are defined as follows:

\[ ACM and PRm : e \mapsto e \text{ if } e \in Or \text{ or } e \in Ec \]
\[ ^{*}a \mapsto ^{*} \text{ if } ^{*}a \in Ma \]
\[ \text{otherwise undefined}; \]
\[ PAm : e \mapsto e \text{ if } e \in Pa \]
\[ \text{otherwise undefined}; \]
\[ PRE : ^{*}a \mapsto ^{*}a \text{ if } ^{*}a \in Ma \]
\[ \text{otherwise undefined}; \]
\[ ACE and PAe : ^{*}a \mapsto ^{*}a \text{ if } ^{*}a \in Ma \]
\[ ^{#}a \mapsto ^{#}a \text{ if } ^{#}a \in Ec \]
\[ \text{otherwise undefined}; \]

The prediction function P is defined as follows:

\[ P(M,E) = \text{if } ^{*} \in M \text{ then } M - \{^*\} \cup E \]
\[ \text{else } M; \]
\[ \text{where } M = Pr(Fm), E = Pr(Fe). \]
III.A.2.2. Proof that Dual-FLM Obeys Constraints

This section gives a brief proof that the Dual-FLM obeys each of the three conditions on \( P(M, E) \).

i) \( P \) is non-empty

Given \( M \not\in \emptyset, \ E \not\in \emptyset, \) then \( P(M, E) \not\in \emptyset \)

Proof

Consider two cases:

a) \(* \in M\)
this implies \( P(M, E) = (M - \{\#\}) \cup E \)
\[ \ni E \] .
From the premise \( E \not\in \emptyset \).

b) \(* \notin M\)
this implies \( P(M, E) = M \).
From the premise \( M \not\in \emptyset \).

End of Proof

ii) \( P \) is reversible

\( e \in P(M, E) \) implies that
if \( \text{PRm}(e) \) is defined then \( \text{PRm}(e) \in M \)
and if \( \text{PRE}(e) \) is defined then \( \text{PRE}(e) \in E \).

Proof

Consider two cases:

a) \( e \in \text{Ma} \)
\( M \cap \text{Ma} = \emptyset \) because the Main FLN is never updated with mark actions as predictions. So \( e \in P(M, E) \) implies that \( P(M, E) \) must be of the form
\[ (M - \{\#\}) \cup E \]
that \( e \in E \),
and that \( * \notin M \).
Let \( e = *a \), say, then :
\( \text{PRm}(e) = \text{PRm}(a) = * \in M \)
and \( \text{PRE}(e) = \text{PRm}(a) = *a \in E \).

b) \( e \notin \text{Ma} \)
\( \text{PRE}(e) \) is undefined so we need only consider \( \text{PRm} \).
In all remaining cases \( \text{PRm}(e) = e \) and as \( e \notin E \)
(the echo memory can only predict mark actions)
then it must be true that
\[ e \in M - \{\#\} \text{ or } e \notin M \]
or in short \( e \in M \).

End of Proof
iii) \( P \) is monotonic

Given \( M \supseteq M' \) and \( E \supseteq E' \) then \( P(M,E) \supseteq P(M',E') \).

**Proof**

Consider the following table of three cases:

<table>
<thead>
<tr>
<th>( \ast \in M, \ast \in M' )</th>
<th>( (M-{\ast}) \cup E )</th>
<th>( (M'-{\ast}) \cup E' )</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M \supseteq M' )</td>
<td>( M'-{\ast} \supseteq M' )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \ast \in M, \ast \notin M' )</td>
<td>( (M-{\ast}) \cup E )</td>
<td>( M' )</td>
<td>( M'-{\ast} = M \supseteq M' )</td>
</tr>
<tr>
<td>( \ast \notin M, \ast \notin M' )</td>
<td>( M )</td>
<td>( M' )</td>
<td></td>
</tr>
</tbody>
</table>

**End of Proof**

**III.A.3. Theorem**

Given an MFLM with FLM memories \( \text{Mem}_1 \ldots \text{Mem}_m \) at the start of one iteration of A.1.2 in run mode and FLM memories \( \text{Mem}_1', \ldots \text{Mem}_m' \) at the end of the iteration;

then for any row of traces \( S_1 \ldots S_m (S_i \) being a valid trace for the \( i \)'th FLM):

\[ P(\text{Pred}(S_1, \text{Mem}_1), \ldots \text{Pred}(S_1, \text{Mem}_m)) \supseteq P(\text{Pred}(S_1, \text{Mem}_1'), \ldots \text{Pred}(S_1, \text{Mem}_m')) \]

**Proof**

Let \( a \) be the action chosen and \( \text{Buf}_i \) be the buffer for the \( i \)'th FLM when the memory is updated with a new trace. The update will be done only if \( \text{PRI}(a) \) is defined, so consider two cases:

a) \( \text{PRI}(a) \) is defined

From the reversibility of \( P \), \( \text{PRI}(a) \in \text{Pred}(\text{Buf}_i, \text{Mem}_i) \) and from the algorithm, \( \text{Buf}_i(0) = \text{PRI}(a) \).

So Theorem II.A.1 for a single FLM can be applied.

This implies that

\[ \text{Pred}(S_i, \text{Mem}_i) \supseteq \text{Pred}(S_i, \text{Mem}_i \cup \{\text{Buf}_i\}) \]

or in the notation of this theorem:

\[ \text{Pred}(S_i, \text{Mem}_i) \supseteq \text{Pred}(S_i, \text{Mem}_i') \quad (1) \]
b) PRI(a) is undefined
   No trace is added to the memory and it is trivially true that \( \text{Pred}(S_i, \text{Mem}_i) = \text{Pred}(S_i, \text{Mem}_i') \)

\[ (2) \]

(1) and (2) together with the monotonicity of \( P \) imply
\[
P(\text{Pred}(S_i, \text{Mem}_1) \ldots \text{Pred}(S_n, \text{Mem}_n))
\geq
P(\text{Pred}(S_i, \text{Mem}_1'), \ldots \text{Pred}(S_n, \text{Mem}_n'))
\]

End of Proof
CHAPTER IV
IMPLEMENTING THE FLM

This chapter is concerned with techniques for implementing an FLM so that it uses storage efficiently and can both be updated and yield predictions quickly. The main emphasis is on practical systems for currently available serial digital computers and mass storage devices such as discs. However, one system which is suitable for implementation using parallel hardware is described.

The chapter starts by considering ways in which the forgetting mechanism described in I.3.2 can be implemented. It is concluded that selecting traces at random is effective and that no great improvement on this is possible especially if the 'improved' technique requires extra storage.

The rest of the chapter examines three broadly different ways of representing an FLM memory. The first of these represents the memory as a tree, the second uses the tail of each trace as a hash key and the third, which is best suited to parallel hardware, uses a single array of overlapping traces. Algorithms will be described in each case for updating the memory, for obtaining best match predictions, for choosing traces to be forgotten and for subsequently deleting them. Also, theoretical estimates of execution times and storage usage will be developed. Actual execution times are given for Felix, one of the tree representations investigated, which has been implemented in Algol 60 for the Burroughs B6700.

The chapter concludes with a comparison of the various implementations and from this comparison makes a selection of those which are, on balance, best for practical implementation.
IV.1. FORGETTING

The process of forgetting, which was introduced in I.3.2 is applied when an FLM memory becomes full; as new traces are added to the memory some other old traces are chosen and deleted to make room for the new ones. A forgetting algorithm, then, will contain two parts; the first selects which trace(s) are to be deleted and the second performs the deletion and any consequent re-arrangement of the memory structure. The process of deletion will be dealt with later as part of each implementation. This section is concerned with two algorithms for selecting traces, namely ladder forgetting and random forgetting.

Section 2.2 shows how ladder forgetting can be used in conjunction with a tree structured memory. Ladder forgetting also forms an integral part of the Simple system described in Section 7. The details of two random forgetting algorithms are given for a tree structured memory in 2.3.4, and another for a hashed memory in 5.7. The current Mew-PUSS (Andreea, 1977) forgets traces at random as does the Felix system described in Section 4.

This section is organized as follows. First the processes of ladder and random forgetting will be defined. Two different models of the set of traces generated by the input to the FLM will then be introduced, together with a criterion of merit whereby the performance of the forgetting algorithms can be judged. Examples of the relationship between memory size and performance are then given for the forgetting algorithms using each model. The technical details and derivations of the results used are given in the appendix, A.1. These results were originally reported in (Cleary, 1974).
IV.1.1. Forgetting Algorithms

For the purposes of ladder forgetting all traces in the memory are linearly ordered. When a trace occurs as input it is placed at the top and any duplicate trace is removed from the memory. If the memory is full and there is no duplicate trace, then the bottommost trace is deleted.

Random forgetting chooses any trace in the memory with equal probability.

For each of the models of the input traces, an upper bound will be found. To do this it will be presumed that complete information about the traces is available so that at any time the trace which is least likely to recur in future can be chosen.

In order to contrast the effect of algorithms which use extra storage, to determine which traces to forget, a best case when the amount of storage per trace is doubled will be included with the results.

IV.1.2. Two Population Model and Performance Criterion

It will be assumed that the inputs to the FLM generate a set P of traces and that P is split into two disjoint subsets Q and R. It is further assumed that each member of Q will occur with some probability q and each member of R with probability r. That these probabilities are well-defined implies that the probability of a trace occurring is independent of the preceding trace. This is strictly untrue for many real input sequences, but it is a reasonable approximation to reality, as in many cases the probability that a trace will occur in any short
sequence of traces will be nearly independent of the first trace in the sequence. This assumption may be restated by saying that the inputs are equi-probable over time intervals which are short compared with the time needed to fill the memory. It will be shown later that violations of this assumption probably favour random forgetting.

The criterion of performance used will be the probability that when a trace occurs it will already be recorded in memory. Relationships between this probability and the size of the memory will be used to compare the forgetting techniques.

**Notation** The results can be expressed in terms of the parameters:

\[ f = \frac{|Q|}{|R|} \] ratio of sizes of Q and R;
\[ a = \frac{q}{r} \] (it is assumed without loss of generality that \( q > r \), that is, \( a > 1 \)).
\[ v \] the size of the FLM memory in terms of the number of traces able to be stored in the memory as a fraction of the size of \( P \).
\[ E \] mean probability that when a trace occurs it will be present in memory.

Fig. 1 gives graphs of \( E \) versus \( v \) for three different values of \( a \) and \( f \). (The derivation of these graphs is given in A.1.) As can be seen, the ladder technique is a little better than random forgetting and both are quite close to the best possible upper bound.
Fig. 1 Results for Two Population Model.
IV.1.3. **Cyclic Model**

A rather extreme model will be investigated now. This assumes that the input consists of a single non-repeating cycle of input symbols. That is, each input occurs but once per cycle. This is of some interest as it is the opposite situation to the last model where traces were assumed to occur randomly.

Graphs of $E$ versus $v$ for the random($R$), ladder($L$) and best($B$) cases are shown in Fig. 2. See A.1 for derivations. In this case the ladder scheme fails catastrophically, for if the memory is one trace shorter than the length of the cycle, then the trace needed for a prediction will be forgotten just before it is to be used. The random case, although declining rapidly with $v$, still achieves a non-zero predictive ability.
Fig. 2 Results for Cyclic Model.
IV.1.4. Conclusion

The random and ladder forgetting algorithms were chosen for consideration because they are simple and as is shown later they can be implemented efficiently. That is traces can be chosen and deleted using them in times comparable to those required to store into and retrieve from the FLM memory. Also they will be seen to require no or a small amount of extra memory to implement. This last point is very important, for if large amounts of extra memory are required then the performance of the memory is degraded as less traces can be stored. This is illustrated by the results above which show that even a best possible forgetting system which used twice the memory per trace would not perform as well as random forgetting.

The results in Fig. 1 also show that, at least for the two population model, there is little room for improvement over the random or ladder algorithms as they approach the best possible case quite closely. This also true for random forgetting in the cyclic input case, illustrated in Fig. 2., when v > 1/2. These results support the general argument that forgetting algorithms cannot make effective use of, say, detailed frequency information about traces because of the extra space this would require. Simple algorithms such as random and ladder forgetting perform sufficiently well that they should enable continued use of an FLM memory when it has filled without any need to resort to more complex and expensive schemes.
IV.2. TREE REPRESENTATION

IV.2.1. Memory Structure

There is a natural and effective representation of an FLM memory as a tree (Knuth, 1973a) gives a good introduction to tree structures). In this tree each node is labelled with a single event and each leaf (terminal node) of the tree represents a single trace as follows:

moving up the tree from the root to the leaf the first (most recent) event in the tail of the trace is encountered and then the second event and so on until the end of the tail. Finally, the leaf itself is labelled with the event at the head of the trace. Fig. 3 shows an FLM memory represented in this way.

The use of a tree structure to represent an FLM and to speed its operation was originally suggested in (Cleary, 1974). The memory compression and ladder forgetting algorithm of 2.2 was also described there.
Traces in Memory

\[
\begin{array}{c}
\text{a } \text{P} \text{ a } \text{P} \\
\text{b } \text{Q} \text{ a } \text{P} \\
\text{b } \text{Q} \text{ b } \text{Q} \\
\text{a } \text{Q} \text{ b } \text{R}
\end{array}
\]

\[
\begin{array}{c}
\text{Po} \\
\text{bo} \\
\text{bo} \\
\text{ao} \\
\text{ao} \\
\text{bo} \\
\text{bo} \\
\text{ar} \\
\text{ao} \\
\text{ao} \\
\text{bo} \\
\text{bo} \\
\text{ao}
\end{array}
\]

Root of Tree

\# Nodes corresponding to the outlined trace

Fig. 3 Representation of FLM Memory as a Tree
IV.2.1.1. Prediction

The great advantage of this tree structure is that it is possible to locate very rapidly the leaf of the tree which corresponds to a particular trace by scanning up the branches of the tree from the root to the leaf. A second advantage is that it allows a best match prediction to be made in a straightforward and natural way. Given a particular trace, say Buf, then a best match prediction can be obtained as follows:

start scanning by setting current-level to 1 and current-node to the root of the tree;
if the current-node is just below the leaves of the tree
then stop searching;
examine all nodes branching up from current-node
if a node is found labelled with Buf[current-level]
then set current-node to this node and add one to current-level;
if no equal node is found then stop searching;

When the search is completed, the prediction is the set of events labelling all the leaves which sprout from current-node or from branches up from it.

Fig. 4 illustrates this prediction algorithm using the memory from Fig. 3. The trace c Q b is matched against the memory. It is found that two nodes above the root which are labelled with b and Q match the corresponding events in the buffer. Neither of the nodes branching up from Q equals the next event e, however. The search therefore stops with current-node pointing at Q. The leaves on the sub-tree branching from Q then yield the prediction set {P, Q, R}.
Nodes which successfully match  
Sub-Tree which yields predictions

This shows the matching of the trace c Q b against the memory of Fig. 3. Two events, b and q, match the memory. The prediction \{P, Q, R\} is obtained from the union of the events on leaves descended from Q.

Fig. 4 Example of String Matching.
IV.2.1.2. Update

The process of adding a new trace to the memory is very similar to that of obtaining a prediction, except that if a matching node cannot be found at any point then it is added to the memory. This is done by obtaining a virgin node from some 'free list' and then initializing it appropriately (details of such a free list are given later). The algorithm below updates the memory with a trace, adding new nodes when necessary:

First initialize a temporary buffer, Temp, with the events of the trace re-arranged so that the tail occurs first followed by the head. (So, c Q b R is re-ordered as R c Q b, i.e. Temp[1] = R, Temp[2] = c etc. This temporary buffer simplifies the rest of the algorithm.)

Set current-node to the root of the tree;
For current-level:=1 to order of FLM +1 do
  Begin examine all nodes branching up from current-node;
    if a node is found labelled with Temp [current-level]
    then set current-node to it
    else if no such node is found obtain a node from the
    free list, label it with Temp [current-level] and set
    current-node to it;
  End.

Fig. 5 illustrates the addition of a new trace to the memory of
Figs. 3 and 4. The trace c Q b S is added. Two nodes are already
present in the memory and match the trace; two more are obtained from
the free list and added to the tree.
This memory is that of Figs. 3 and 4 updated with the trace c Q b S. The "temporary buffer" shows how the Buffer trace is rearranged so as to simplify the updating algorithm.

**Fig. 5 Example of Updating Tree Memory**
IV.2.2. Ladder Forgetting in Trees

I will now show how the ladder forgetting algorithm investigated in Section 1 can be implemented in an FLM tree structure. An arbitrarily small extra amount of storage is needed (however, a larger amount of storage decreases the execution time). It will be shown that the execution time of the algorithm can be kept comparable to those for updating or prediction.

For the moment I will ignore the nodes in the body of the tree structure and consider only the leaves at the top of the tree. For the purposes of this algorithm they will be stored in a single contiguous block of memory, the ladder. Each time a trace occurs it is placed in the next available location at the top of the memory and if it is already present its earlier occurrence is deleted. So consider some tail with the set of leaves X, Y, Z. This might be represented on the ladder as follows:

When X occurs again this will cause the memory to be updated as shown. Finally, when the last location at the top of the ladder is filled it will be necessary to delete leaves and consolidate the ladder.
IV.2.2.1. Forgetting Time

The great difficulty, now, is to implement the consolidation of the ladder in such a way that the execution time to manipulate it doesn't outweigh the update time. In fact, with some care and the use of extra storage, it is possible to keep the average execution time down to an arbitrarily small constant.

Assume that initially T locations are available for the list of leaves. An extra area of $\epsilon T$ is then provided for expansion. When the whole memory, $(1+\epsilon)T$, is full, sufficient entries are elided from the bottom of the list to permit the whole memory to be compressed into the bottom T positions. The algorithm below permits the memory to be compressed and pointers updated in a time proportional to T. At most one trace will occur each update cycle so the memory will become full at most every $\epsilon T$ cycles. The time to forget then averages to $T/\epsilon T = 1/\epsilon$.

IV.2.2.2. Consolidation of Ladder

I will now show how the list of leaves can be consolidated in a time proportional only to the size of the memory.

When the leaf list is full, two things must be done. Sufficient cells must be set empty to enable the memory to fit into the first T locations of the list. Then, the remaining entries must be shifted down and the pointers among the various entries adjusted to reflect the new situation. To enable the pointers to be adjusted without using extra memory is the trickiest part of the operation and the manipulations to do this will form the major part of the following algorithm (see Fig. 6). (A garbage collection algorithm very similar to this is given in (Fisher, 1974).)
It will be assumed that it is known how many entries at the bottom of memory must be elided. This can easily be calculated by keeping count of how many entries are deleted during normal updating. To follow the algorithm it is easiest to work through an example.

Consider a memory with the list of leaves \{A, B, C, D, E\} included in it. This might be recorded in memory as shown here:

(Note that there will be other linked lists forming part of the memory and interspersed with this.)

First, every node in the tree will be visited and the pointer at the node and the first pointer in the leaf set will be swapped to leave the pointer of the first member of the leaf set pointing back at the node and the node pointing at the second member of the leaf set. (It should be noted that it is important for this process to have the pointers pointing up the memory.)

The memory will now look like:

The list is now ready for the eliding of cells to begin.

Starting at the bottom of the list, each cell in turn is examined, any occupied cell is deleted and the pointers adjusted. This continues until sufficient active cells have been elided to permit compression. It is this step that requires prior knowledge of the number of empty cells. Assume that the changeover occurs between A and B so that A is removed and B is not.
The memory has become:

It will be noted that the pointers at B and at the node have been adjusted. When A was encountered the node N could be located because of the pointer at A. B could then be located because of the pointer at the node. Thus, the pointers to be adjusted when A was encountered were available by following a chain of at the most two pointers.

The scan of the list now continues, but when occupied cells are encountered they are pushed towards the bottom of memory. If B', C', D' and E' are the shifted locations of the cells after they have been encountered during a scan, then the memory goes through the following sequence of changes. It will be noticed that each time a cell is encountered which needs to be shifted, then, because of the arrangement of pointers, all pointers which need to be updated as a result of the shift are available by looking down a list of at the most two pointers.

B shifted  C shifted  D shifted  E shifted

The overall tree memory can be organized in such a way that the leaves and the space for the tree structure compete directly with one
another by having the memory for the leaves grow in one direction and that for the nodes in the other; this is illustrated in the diagram.

Fig. 6 Compression of Ladder
IV.2.3. Concrete Representation of Tree Structure

In this section I consider the representation of the tree structure developed in the last section in terms of actual data structures. I will use a Pascal-like notation (Jensen and Wirth, 1978) in expressing these data structures and the algorithms to operate on them. I will first introduce a very simple data structure and in the following sections I will consider various elaborations for speeding up the programs that operate on it, and for reducing its storage requirements.

Let each node in the tree structure consist of three fields:
value - this contains the single event which labels the node;
up - a pointer to the next highest 'level' in the tree, this points to nodes branching from the current node;
next - a pointer to a node at the same level in the tree.

All the nodes which branch from a particular node are joined into a list by next pointers. The list is terminated by a null pointer. Figs. 7 and 8 give examples of FLMs represented in this way. Note that the 'up' pointers are null for nodes at the leaves of trees and that the 'value' for the root node is irrelevant. (Having a single root node which can be used as a starting point for tree-searches simplifies the details of the algorithms.)
Fig. 7 Fragment of Tree with Corresponding Memory Locations

Fig. 8 Tree of Fig. 5 Redrawn
One of the advantages of the technique above is that all the nodes in the tree occupy a single memory location of the same size. This means that all the unused memory locations can be joined together by a single linked list, the free list, and space for a new node can be obtained by taking one memory location from the beginning of the list. Similarly, if a trace is deleted from the tree then the memory locations thus freed can be easily linked onto the free list one by one.

This memory structure can be described in Pascal as follows:

```pascal
var Memory: array[1..MemorySize]
record
  value: event;
  up,next: 0..MemorySize;
end;
FreeCount,FREE: 0..MemorySize;
```

This says that the entire memory consists of an array of locations or 'records' numbered 1 to MemorySize. The 'value' field in the record encodes an event in an unspecified way (I will deal with the details of this later). The pointers 'up' and 'next' can take on any integer value from 0 to MemorySize. A 0 indicates a null pointer, positive values indicate an index into the memory array. 'Free' is a separate variable outside the memory array which points to the first memory location on the free list. 'Freecount' is a count of the number of memory locations on the free list (this is useful later when it is necessary to ensure that there are a certain number of locations on the free list before doing an update of the memory.)
IV.2.3.1. Initialization and Free List

Before using the memory it is necessary to initialize it so that it contains no traces and so that the free list links together all the memory locations. In the following routine the root node is placed at location 1 and the bulk of the memory, locations 2 to MemorySize, are linked together onto the free list:

```pascal
with Mem[1] do {initialize root node}
Begin Up := 0; Next := 1; {see Forget routine}
End;
{link rest of memory onto free list}
FreeCount := MemorySize - 1;
Free := 2;
For i := 2 to MemorySize - 1 do Memory[i].Next := i+1;
Memory[MemorySize].Next := 0; {end of free list}
```

To use the free list it is necessary to be able to extract nodes from it and to place nodes onto it. Extraction is done by the following function, which returns a pointer to a node which has just been excised from the free list:

```pascal
Function GetFree : 1..MemorySize;
Begin
  GetFree := Free; {a run-time error will occur here}
  {if the free list is empty}
  Free := Memory[Free].next;
  FreeCount := FreeCount - 1;
End; {of GetFree}
```

The following routine takes a node and links it onto the free list. The node should have been correctly excised from the tree structure before doing this (see 2.3.5 for an example of its use).

```pascal
Procedure PutFree (i : 1..MemorySize);
{put node i onto free list}
Begin Memory[i].next := Free;
  Free := i;
  FreeCount := FreeCount + 1;
End; {of PutFree}
```

When updating the memory with a new trace it is very convenient to know that there are sufficient nodes on the free list to satisfy all requests for nodes via GetFree. The worst possible case is that a new
node will be added at every level in the tree, that is \((order + 1)\) nodes. So before starting the update the routine 'ensure' is called with a request for the number of nodes which are required to be present on the free list. This calls a routine Forget which selects traces in the memory to delete and returns their nodes to the free list. The advantage of doing deletion in this way before any updating is that it is not necessary to do any deletions while the update is in progress, when it would be necessary to do very careful checks that none of the nodes involved in the update were disturbed by the deletion. The code for 'Ensure' is as follows:

```haskell
Procedure Ensure (n:0...MemorySize - 1);
Begin [ensure at least n nodes on free list]
  While FreeCount < n do Forget;
End;
```

IV.2.3.2. Memory Update

The following routine will update the memory with a trace. It is presumed that the trace is held in the array 'buffer' in the inverted format explained in 2.1.2 and Fig. 5, i.e. Buffer[0] is the first event in the tail of the trace, and Buffer[order] is the head of the trace.
Procedure Update (Buffer: array [0..order] event);
var searchstate : (scanning, nodepresent);
    [control for search up tree]
p : 1..MemorySize; {points to last node definitely matched}
s : 0..MemorySize; {points to each node as it is examined}
l : 0..order; {points to each level of tree in turn}
Begin
    Ensure (order + 1); {make sure enough room on free list for worst case}
p := 1; {start search at root of tree}
For l := 0 to order do
    Begin s := Memory[p].up; {scan all descendants of p}
        searchstate := scanning;
        repeat
            if s <> 0 then
                with Memory[s] do
                    if value = Buffer[l] then {matching node found}
                        searchstate := nodepresent
                    else {not this node, look at next one}
                        s := next
                else {no node matches at this level, add one above p}
                Begin
                    s := GetFree;
                    with Memory[s] do {initialize new node}
                        Begin Value := Buffer[l];
                            Up := 0;
                            Next := Memory[p].up;
                        End;
                    Memory[p].up := s; {put in upward branch}
                    searchstate := nodepresent;
                End;
            until searchstate <> scanning;
            p := s; {step up to next level}
    End; {of For l}
End; {of Update}

IV.2.3.3. Prediction

The process of obtaining a prediction from the memory breaks down into two steps. The first climbs the tree as far as possible while matching nodes, and then when the best match node has been found the leaves of the subtree descended from it must be added to the prediction set. In the procedure below this last step is done by a separate procedure 'Scan' which calls itself recursively.

The routine 'Predict' has two parameters, 'Buffer' which
contains the tail of the trace to be matched and 'result' which will contain the set of events when the prediction is complete. (In practice the prediction set would probably be represented by an ordered list of events, rather than by using the standard Pascal set of construct.)

Procedure Predict (buffer : array [0..order - 1] event;
var result : set of event);
    p : 1..MemorySize; {last node definitely matched}
    s : 0..MemorySize; {points to each node as it is examined}
    l : 0..order - 1; {points to each level of tree in turn}
Procedure Scan (p : 1..MemorySize, l : 1..order);
    {recursively scan tree descended from p, place all leaves in result}
var t : 0..MemorySize; {points to each descendant of p in turn}
Begin t := Memory[p].up;
    while t <> 0 do
        with Memory[t] do
        Begin
            if l = order then [a leaf]
                result := result + [value];
            else Scan(t); {recursively scan sub-tree}
                t := next;
        End; [of while .with]
    End; [of Scan]
Begin(Predict)
    {start search at first branch above root}
    p := 1; s := Memory [p].up; l := 0;
    while l < order and s <> 0 do
        [search until complete match, or no match found]
        with Memory[s] do
            if value = Buffer [l]
                then [match, move up a level]
                    Begin p := s; s := up; l := l + 1;
            End
            else s := next; {no match, step to next node}
        {matching complete}
        result := []; {clear prediction}
        if l > 0 then Scan (p,l); {put all leaves in result}
    End; [of Predict]

IV.2.3.4. Forgetting

I will now consider two routines for randomly selecting traces so that they can be deleted as part of a forgetting mechanism.

The first of these routines is used as part of the Felix system introduced in section 4. It has the unfortunate characteristic, however, that the probability of a particular trace being chosen is
dependent on the structure of the tree and varies from one trace to the next.

The second routine maintains a pointer to the next trace to be deleted in the tree. When the trace is deleted this pointer is stepped to another trace. Each trace then has an equal probability of being chosen. However, this procedure has a tendency to completely delete parts of the memory before moving on. This tendency can be ameliorated by stepping the pointer a number of times between each deletion.

**Random Branches**

This routine starts at the root node of the tree and progressively builds up a list of nodes leading from the root to the leaf of the chosen trace. If the root has \( a_1 \) branches leading from it, then one of them is chosen with probability \( 1/a_1 \). If, from the node so chosen, there are \( a_2 \) branches then one of them is chosen with probability \( 1/a_2 \) and so on until a leaf of the tree is reached. The routine of 2.3.5 can then be applied to the list of nodes in order to delete the trace.

Choosing a branch is done very simply in this routine by maintaining two variables which are moved progressively along the 'next' chain at a particular level. One, \( x \) in the routine below, points to each node in turn, the other, \( i \) below, is set to \( x \) randomly and when the scan is complete points to the chosen node. A count is kept, in the variable \( n \), of the number of nodes traversed so far. At each step \( x \) is set to the next node and \( i \) is set to \( x \) with probability \( 1/n \).

I assert that at each step, and in particular when the scan is finished, \( i \) will point to each of the \( n \) nodes traversed so far with probability \( 1/n \). This can be proven by induction as follows:
On the first step \( n=1 \) and \( i \) is always set to \( x \) so the assertion is true.

If the assertion is true at step \( n-1 \) then each of the first \( n-1 \) nodes will equal \( i \) with probability \( 1/(n-1) \). Then \( i \) will be set to the \( n' \)th node with probability \( 1/n \) leaving it equal to the preceding nodes with probability, \( 1/(n-1).((1-(1/n)) = 1/n. \)

Procedure Forget;

\begin{verbatim}
var 1 : 0.order + 1;
x : 0..MemorySize; {scanned along nodes at each level}
i : 1..MemorySize; {nodes chosen at each level}
ToBeDeleted: array [0..order] 1..MemorySize;
{route to chosen trace}
lri : integer; {large random integer}

Begin
  i := 1; ToBeDeleted [0] := 1; {start at root node}
  For l := 1 to order + 1 do
    Begin
      lri := large random integer;
      x := Memory [i].up;
      n := 0;
      repeat
        n := n + 1;
        if lri mod n = 0
          then i := x; {true with 1/n probability}
          x := Memory[x].next;
      until x = 0;
      ToBeDeleted [l] := i;
    End;
  Delete (ToBeDeleted);
End; {of forget}
\end{verbatim}

This algorithm has one unfortunate characteristic; if the amount of branching varies through the tree then the probability with which particular nodes will be chosen may vary considerably. The probabilities for each trace are listed on the simple tree below:
It is not clear whether this distortion will cause problems in practice.

Successive Traces

This forgetting routine maintains the array ToBeDeleted between successive calls to it. As each trace is deleted this array is stepped to the next trace in the tree which will be deleted in turn on the next call to forget a trace.

On entry to the procedure Forget, the topmost entries of ToBeDeleted may be zero as a result of the last call to Delete (see 2.3.5) which zeroes each entry as the node it points to is deleted. Before Delete is called again it is necessary to reset these pointers to valid nodes. This is done in two distinct steps. The first attempts to move the topmost non-zero pointer to the next node at the same level in the tree. If there is no such node then the entry in ToBeDeleted is set to zero and the node one level down in the tree is tried. In the worst case this will terminate at the root node which, with considerable foresight, was left pointing at itself by the initialization routine. Consider this process when applied to the following tree where the nodes which are pointed at by ToBeDeleted are marked with an *.
After this process some of the topmost values of ToBeDeleted are still zero. They are filled in by moving up the tree from the topmost non-zero entry and taking the first (leftmost) branch at each step. The tree from above will then be marked as follows:

When repeatedly applied, this routine will successively delete all the traces in one sub-tree before moving to the next. For example, all the traces with the same first event on their tail will be deleted before ToBeDeleted is moved to the next event and all its traces are deleted. This concentration on particular parts of the memory may be unacceptable in some applications. It can be easily ameliorated, however, by stepping the ToBeDeleted pointers more than once before the next call to Delete. So if the pointers were stepped twice before each call then half the traces in a particular sub-tree would be weeded out.
before the next sub-tree was entered. For example, if this were done to
the tree above, the next trace marked for deletion would be:

```
```

In the routine below a constant step is provided which
determines how many steps will be done between each deletion. The
larger this is set at, the more 'random' the deletions will appear to
be. However, processing time will be increased accordingly.

To set up the delete routine the following declarations need to
be added to the FLM memory structure:

```c
const step = 2; [must be >= 1];
var ToBeDeleted: array [0..order + 1]0..MemorySize;
```

and the following to be added to the initialization routine:

```c
ToBeDeleted [0] := 1;
For i := 1 to order + 1 do ToBeDeleted [i] := 0.
```

Note that ToBeDeleted [0] always equals 1; that is, it always points to
the root node.
Procedure Forget;
\textbf{var} i : T..step; \{counts number of steps\}
\textbf{var} x : 0..MemorySize;
\{temporary pointer while setting and resetting ToBeDeleted\}
l : 0..order+2; \{points to each level in tree\}
\textbf{Begin}
\textbf{For} i := 1 \textbf{to} step \textbf{do} \{specified number of steps\}
\textbf{Begin} \{first ensure topmost non-zero node moved sideways\}
l := order + 2;
\textbf{repeat}
l := l - 1; x := ToBeDeleted \[l\];
\textbf{if} x <> 0 \textbf{then} x := Memory[x].next;
ToBeDeleted \[l\] := x;
\textbf{until} x <> 0;
\textbf{\{move up tree from where we stopped\}}
\textbf{\{take left hand branches\}}
\textbf{For} l := l + 1 \textbf{to} order + 1 \textbf{do}
\textbf{Begin}
x := Memory \[x\].up;
ToBeDeleted[l] := x
\textbf{End};
\textbf{End}; \{of For step\}
Delete (ToBeDeleted);
\textbf{End}; \{of Forget\}

IV.2.3.5. Deletion

I will now give the details of a routine to delete a trace. It uses the same convention as the Forget routines in 2.3.4; that is, it is given an array, ToBeDeleted, which points at each node from the root to the leaf of the trace to be deleted.

Fig. 9 provides an illustration of the working of the routine Delete. It first deletes the leaf of the trace, it then moves down a level and checks to see if this has been left with no branches; that is it has a zero up pointer. If this is so, it is deleted in turn. As each node is deleted two cases must be distinguished; either the node is pointed at by an up pointer at the level below, or it is pointed at by a next pointer at the same level. In both cases the appropriate pointer must be adjusted to point to the next field beyond the node being deleted. Whenever a node is deleted, the corresponding entry in
ToBeDeleted is zeroed for the convenience of the Forget routines.

```plaintext
Procedure Delete (var ToBeDeleted:array[0..order+1] 0..MemorySize);
[Delete trace whose nodes are pointed at by ToBeDeleted]
var l : 0..order; {points to each level of tree in turn}
    del : 1..MemorySize; {successive nodes as they are deleted}
    downalevel : 1..MemorySize; {node one level below}
        {that being deleted}
    t : 1..MemorySize; {used when looking for del}
    x : 0..MemorySize; {next pointer of del}
Begin l := order + 1; del := ToBeDeleted [1];
    while Memory [del].up = 0 and l > 0 do
        Begin
            downalevel := ToBeDeleted [l-1];
            x := Memory[del].next;
            with Memory [Downalevel] do
                if up = del
                    then up := x {del pointed at by an up pointer}
                        else Begin {del pointed at by a next pointer, find it}
                            t := up;
                            while Memory [t].next <> del do
                                t := Memory [t].next;
                                Memory [t].next := x;
                        End;
            ToBeDeleted [l] := x; {see Forget routine}
            PutFree (Del); {place on free list}
            l := l - 1; Del := Downalevel;
        End; {of while loop}
    End; {of Delete};
```
Fig. 9 Example of Deletion of Trace From Memory
IV.2.3.6. Execution Times

Before obtaining estimates of the execution times of the routines introduced so far I will introduce some notation. This notation is used consistently throughout the chapter and is summarized in section 9.

Notation

T - is the total number of traces in the memory.
N - is the order of the FLM.
L_0, 0 \leq i \leq N, - is the number of nodes at level i in the tree. Level 0 contains only the root node so L_0 = 1.
L - is the total number of nodes in the memory excluding the leaves; L = \sum_{i=1}^{N} L_i.

a_i, 1 \leq i \leq N, is the 'branching ratio' or average number of branches up from a node at level i-1; a_i = L_i / L_{i-1}.
a - is the average number of heads for a given tail; a = T / N.

Update

The most time consuming element of the update routine is the scan of the upward branches at each level in the tree. The time to scan the tree can be estimated by the number of branches examined during the search. At each level an average of half the branches will be examined so the total will be

S = 1/2 \sum_{i=1}^{N} a_i.

Because T = a_1 \cdot a_2 \cdots a_N it is possible to place bounds on S. It will be a maximum when a_1 = T and a_2 = a_3 = \ldots = a_N = 1; that is S = T + N - 1 \leq T.

The minimum value for S will occur when all of the a_i are equal.
that is; \( a_i \triangleq T \) and \( S = N.T \).

**Prediction**

The prediction routine can be split into two parts. In the first part, which consists of matching the current buffer, the execution time will be similar to that for the match in the update routine. However, in the second part which scans the sub-tree above the best match node, a very large part of the tree might have to be examined. In the worst case, when only one event matches the entire sub-tree above, that event will be scanned. The very bad cases are ameliorated by the fact that the larger the sub-tree the more likely it is that a long match will be made. Also, as usually only a small number of the events in a prediction set are actually used, it is possible to place a limit on the number of events allowed in the prediction set. When this upper limit is exceeded the scan of the sub-tree can be halted. The results of Section 4.3 show that in Felix, where this limiting process is used, the scanning part of the routine takes less time than the matching part.

**Forgetting**

The first of the algorithms for selecting traces for forgetting (2.3.4) is very similar to a best match. At each level all the branches up from the last node chosen are examined, so the execution time can be estimated as \( \sum_{i=1}^{N} a_i \), about twice that for a best match.

The second forgetting algorithm steps the pointers of ToBeDeleted sideways one or more steps. The time to do this can be estimated by imagining that the steps are repeated \( T \) times, so that each trace in the memory is visited just once. During this process each
node in the memory will be assigned to one of the pointers in ToBeDeleted just once. The number of iterations of the two innermost loops of the routine will be equal to at most twice the number of such assignments, so they can be used to estimate the execution time. The total number of nodes in the tree is just \( L + T \), so the average number of iterations in one step is 
\[
2(L + T)/T = 2(1 + L/T).
\]
The examples of section 8 give values for \( L/T \) in the range 1.5 to 2.5, so the execution time can be expected to be relatively short.

**Deletion**

The deletion of a trace may be thought of as occurring in two parts. The first part moves from the leaf toward the root, deleting nodes which form just a single branch from the node below them. There will be at most \( N \) nodes deleted in this way. The second part deletes a single node which is one amongst two or more branches. This involves searching the nodes at the same level for the one whose next pointer points to the node being deleted. If the search is at level \( i \) then an average \( 1/2a_i \) nodes will be examined. The deletion then will certainly involve fewer nodes than a best match search.

**IV.2.3.7. Storage Requirements**

I will express the storage requirements for an FLM in terms of the number of bits required to store a single trace. To do this for the tree memory I will need two values as follows:

- \( e \) - the number of bits used to encode an event;
- \( p \) - the number of bits used to hold a pointer into the memory.

Note that \( p \geq \lceil \log_2(M) \rceil \).

The number of bits at each memory location is \( e + 2p \) (one labelling event and two pointers) and the total number of locations
used is $L + T$ so the number of bits per trace is

$$b = (e + 2p)(L + T)/T$$

$$= (e + 2p)(1 + L/T).$$

IV.2.4. **Binary Tree**

The tree representation used above was chosen mainly for its simplicity and efficient use of storage. However, the tree can be structured as a binary tree (Knuth, 1973a) which reduces the execution time at some cost in storage space.

In this binary representation, each location has two pointers, 'left' and 'right', in place of the next pointer. Only nodes whose labelling events are less than that of a particular node will be found along the left pointer and only those which are greater along the right pointer. Each memory location, then, will contain four fields as follows:

up - pointing up to the nodes branching from this node;
value - the event which labels this node;
left - a pointer to those nodes on the same level whose value is less than in this node;
right - a pointer to those nodes on the same level whose value is greater than in this node.

Consider for example the following fragment of a tree rendered as a binary tree:
The memory declaration for such a tree in Pascal might be:

```pascal
var Memory : array [1..MemorySize] of record
  value : event;
  up, left, right : 0..MemorySize;
end;
Freecount, Free : 0..MemorySize;
```

When matching a trace against the memory it is not necessary to examine all the branches up from each node as it is matched. Rather, one can choose to move down the left or right pointer, depending on whether the event in the buffer is less than or greater than the event labelling the node. If there are branches from a node then the time to do such a search will be proportional to \( \log_2(a_i) \). So the time to do a complete match will be

\[
S = \sum_{i=1}^{N} \log_2(a_i)
\]

\[
= \log_2(a_1 a_2 \ldots a_N)
\]

\[
= \log_2(T).
\]

That is; the execution time for a match is proportional to the
logarithm of the number of traces in the memory.

Each location in the memory has an event and three pointer fields so the number of bits per trace is

\[ b = (e + 3p)(1 + L/T). \]

IV.2.5. Symbol Table of Events

One problem I have not considered so far is how to represent an event in the FLM memory. I have presumed that each event can be represented in a field with approximately the same number of bits as a pointer into memory. However, in some applications the individual events may well be longer composite structures; for example, in PURR-PUSS, the events, when interacting with the teacher, may be arbitrarily long strings of characters. The storage and retrieval of such strings can be neatly integrated into a tree structure for the FLM.

Each event string can be represented by the node of a tree in the same way that traces, which are just strings of events, are represented by the leaves of a tree. For example, the three events "awl", "awe", "awed" could be represented by the following tree where the nodes marked 0 represent the events:

![Event Tree Diagram]

If this "event tree" is stored in the same memory as the FLM tree then the value field in the nodes can be a memory pointer to the
nodes of the event tree. Also, the first level of the FLM tree can be replaced by the event tree itself. The node for a particular event will then have an upward pointer to the traces which have it as the first event in their tail. Fig. 10 shows three successive elaborations of a tree integrated in this way. It shows an FLM memory of order 1 with the events in the tree fragment given above. It can be seen there how the nodes in the event tree point up into the FLM tree proper.

The event tree must needs be a little more complex than the FLM tree for two reasons. Firstly, there may be event strings which are substrings of other events, for example "awe" is a substring of "awed". So the node corresponding to "awe" must point up both to the node for "awed" and into the FLM tree proper. Secondly, if the event string is to be decoded when, say, a prediction occurs then it is necessary to be able to traverse the string back down the event tree from its corresponding node to the root. For example, if "awe" is predicted then the prediction will be a pointer to the node labelled "l". It must be possible then to get to "w" and then to "a". This implies that there must be a pointer from each node in the event tree back down to the node from which it branches.

All this implies that each node in the event tree must have five fields:

- symbol - a character from the event string;
- next symbol - a pointer to a node which is descended from the same node as this (analogous to the next field in the FLM tree);
- string - a pointer up the event tree to the next larger string;
- down - a pointer back down the event tree to the node
from which this branches;

trace – a pointer up into the FLM tree proper.

If this is to be integrated into the tree structure of 2.3 then it is convenient, especially for maintenance of the free list, to have all the nodes the same size, that is about three 'pointer sized' fields. This can be done by splitting the nodes of the event tree over two locations, the first containing the symbol and nextsymbol fields, and a 'tsd' pointer to the second containing the trace, string and down fields.

In Fig. 10 the locations of the first type are marked with a '+' and those of the second with an 'x'. The nodes of the FLM tree proper are marked with a 'o'.

The Pascal declaration for such a memory might be:

```pascal
var Memory: array[1..MemorySize] record
  case (FLMnode, Firstevent, Secondevent) of
  FLMnode :
    (value, up, next : 0..MemorySize);
  Firstevent :
    (symbol : char;
      tsd, nextsymbol : 0..MemorySize);
  Secondevent :
    (trace, string, down : 0..MemorySize)
end;
```

The updating and searching of this tree is a straightforward elaboration of the algorithms given earlier for the FLM tree, although there is one point where care is needed. When calling the procedure Ensure at the beginning of an 'Update', allowance must be made for any new events which might be added, that is: two nodes for each character in the event string.
Successive elaborations of an FLM tree.

Tree representation of FLM showing nodes and their labels:

"awe" — "awed" — "awed"
"awl" — "awe" — "awed"

Traces present in FLM (order=1):
"awl" — "awe" — "awed" — "awed"

--- > value pointer to an event
--- > down pointer within event tree
--- > upward or next pointers

upper part of memory holding traces
first level of tree holding single events

Tree representation showing individual memory locations:

Layout of memory locations:
- ovalue: up next
- *trace: string down
- +symbol: tsd next-sym

Fig. 10 FLM Tree Including Symbol Table For Events
IV.2.6. Multiple FLMs in Same Memory

In applications involving more than one FLM it is desirable to place all the traces from the different FLMs into the same memory. In this way the total available space can be utilized most efficiently because it is not necessary to determine beforehand how long each memory will be on its own. Such estimates are likely to be wrong and one memory might become full long before the others.

Some care is necessary when integrating memories in this way as two different FLMs may have traces which are of different lengths but which match over their common length. For example, two traces from different FLMs might be:

\[ \text{A} \#a \#a \text{P} \]

and \[ \#a \#a \text{Q} \]

or in tree form:

```
P
 \_A
  \_/\ 
 #a Q
 #a
```

So a best match on \#a #a - will incorrectly predict A as well as Q.

Such traces are most easily distinguished by inserting a marker event which is unique to each FLM at the start of the tail of each trace. So the traces above might be rendered as follows:
A #a #a 1 P
and #a #a 2 Q

or in tree form:

This requires no change to the update and prediction algorithms and only a trivial change to the update of the FLM buffer.

IV.2.7. Use of Disc Storage

To increase the size of the memory used for storing an FLM it is possible to use some form of backing store such as disc. The salient property of discs is that they can take a relatively long time to locate a particular record but that once located a large volume of data can be transferred quickly. A naive approach to using disc storage might be to split the FLM memory into contiguous fixed size records and to read a record whenever a pointer into it is encountered. This, however, could involve a very large number of disc accesses once the memory becomes reasonably full; for example, it might be necessary to read a record for every node examined during a search of the memory.

It is possible, however, to reduce the number of disc accesses by splitting the memory into two distinct areas. One of these areas, which is permanently resident on central store, contains the root and
all the nodes in the tree up to some fixed level. The second area is normally resident on disc as a number of separate records, which are read into central store as required. Each such disc record contains its own free list and any pointers from nodes within the record point back into the record. Thus any sub-tree within a disc record is contained wholly within that record. For example, a tree might be split as follows:

In this scheme only one disc access is necessary when matching a trace with memory. The match proceeds up the tree until a pointer to a disc record is encountered, the record is read onto central store and the match is completed entirely within this disc record. Similarly, when a prediction is done, provided only those parts of the tree up to the first event in the trace are held on the central store record, the collection of leaves for the prediction set can be done entirely within a single disc record.

A natural way of splitting the tree is to hold in core the event tree (2.5) and any nodes used to separate different FLMs (2.6), with the bulk of the tree split amongst the disc records. Alternatively if the suggestion of section 5.2 were adopted and any matches less than a fixed length, say 3, were ignored, then the central store could hold all nodes up to level 3 in the tree and the disc records only level 4 and above.

One advantage of restricting pointers to the disc record within
which they lie is that the number of bits per pointer can be reduced. One disadvantage is that as soon as the free list in a particular record is exhausted traces must be forgotten from within that record; it is not possible to use free space available on other records. If the disc records are large and the allocation of traces to them is random then the probability of a particular disc record or the central record becoming full is very small until almost the entire memory is full. However, some work is needed to ensure that traces can in fact be assigned sufficiently randomly to disc records for this to happen.
IV.3.  NETWORKS

Various experiments with FLM like devices have attempted to use forward predictions beyond the immediate prediction of the next action. These attempts have followed from the observation that the FLM potentially holds much information about the future behaviour of the external world and that if more of this information is used it may be possible to make better selections of actions.

The mechanism of 'novelty' (Andreae, 1977) marks traces which have occurred but once. When selecting an action, an attempt is made to form an 'hypothesis' that is a sequence of actions, from the current trace to a future novel trace.

Reward and Pain were used in (Heads, 1975a, 1975b) for distinguishing traces which were 'desirable' and should be repeated and those which were undesirable and should be avoided. When selecting an action, those which lead to future reward are preferred over those which lead to future pain. A modified form of pain called 'disapproval' is used in (Andreae, 1977). The idea of reward, pain and other similar notions has been suggested by many workers in artificial intelligence; see for example the 'utility function' used in (Kookan, 1974).

Common to these mechanisms is a need to search forward through significant parts of the memory looking for predicted future occurrences of novelty, pain, reward or whatever. In this section I will present a data structure which allows such look-ahead through the memory to be done very efficiently. I will show also that it can be integrated into the tree-representation. The representation of an FLM as a network preceded the tree representation, see (Cleary, 1973). The possibility of integrating the two was first raised in (Cleary, 1975a).
In Chapter VI I will solve one of the problems encountered when actually evaluating the potential reward or pain which might result from a particular action. Here, however, I am concerned only with the underlying process of searching forward through the memory.

IV.3.1. Network Representation

This representation assumes that any event which appears in the tail of the buffer has previously appeared in the head of the buffer. So the entire FLM buffer, not just its tail, acts as a shift register; events are shifted into the head of the buffer and then step by step down its tail. For example in the notation of the MFLM of Chapter III the update functions PR and AC must be identical. (All the memories used in PURR-PUSS and the Simple system introduced later in this chapter act in this way.)

Let each tail (of length N) represented in the memory of an FLM be assigned a unique node in a network (graph). Then each trace in the memory can be represented by a (directed) transition between the nodes. Provided each node is labelled with the tail it represents, then the network is equivalent to the FLM memory. Fig. 11 shows the relationship between a trace and the transition representing it. It also gives an example of an FLM memory represented as a network.

When writing such a network all except the last symbol of the string at a node can be omitted, as the transitions to the node carry the rest of the symbols in the string. (If there are no transitions to the string, then the node will need to be labelled explicitly with the whole string.)

When doing a forward search of the FLM memory one starts at the
node in memory corresponding to the current buffer and then follows the transitions out from this to succeeding nodes, and from them to their successors and so on. The transitions between nodes can be represented by single pointers so that (at least in central memory) chaining from one node to the next can be very fast.
(a) Relationship between transition and trace it represents 
(N=2)

<table>
<thead>
<tr>
<th>tail</th>
<th>a b</th>
</tr>
</thead>
<tbody>
<tr>
<td>trace</td>
<td>a b c</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>tail</td>
<td>b c</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ab \rightarrow bc 

(b) Example of Memory (N=3)

<table>
<thead>
<tr>
<th>traces</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>aab a</td>
<td>o</td>
</tr>
<tr>
<td>aab b</td>
<td>o</td>
</tr>
<tr>
<td>aab c</td>
<td>o</td>
</tr>
<tr>
<td>abc b</td>
<td>o</td>
</tr>
<tr>
<td>bc b c</td>
<td>o o</td>
</tr>
<tr>
<td>aab d</td>
<td>o o</td>
</tr>
</tbody>
</table>

This can be written as

Fig. 11 Network Representation of an FLM.
IV.3.2. Nets Over Trees

The nodes in the net representation are in one to one correspondence with the nodes in the tree representation which lie just below the leaves. Therefore, the net can easily be inserted over the top of the tree. Each leaf only needs a 'net' pointer down one level in the tree to the appropriate node. The 'up' pointer (the name is rather misleading in this context) is unused by the leaves so it can be used for this purpose. Fig. 12 shows the net of Fig. 11 rendered as an integrated net and tree. A fragment of this memory is shown elaborated into the node structure described in 2.3.

IV.3.3. Update Algorithm

The only change required of the update and prediction algorithms is the insertion of the appropriate net pointer when a new trace is added. This, however, requires that the node being pointed at should exist. Therefore the update algorithm needs to locate (and if necessary create) the node pointed at. This is easily done by repeating the update search using a buffer modified so that the current head has been shifted down into the tail. The net pointer can then be stored into the leaf corresponding to the new trace.
Fig. 12 Integration of Net and Tree Structure
IV.3.4. **Deletion**

The extra pointers which constitute the net cause some difficulties when deleting traces. If a node just below the leaves of the tree is deleted then net pointers may be left pointing at junk in the free list.

It is not too difficult, however, to find these pointers and zero them. Consider some node just below the leaves. This will correspond to the tail of some trace, say, a b c -. Now any trace which has a net pointer to this node must be of the form x a b c for some event x. All such traces can be located by first finding the node corresponding to a b -, then all leaves descended from this node (which is two levels below the leaves) need to be checked for a pointer to a b c -. A fragment of the tree of Fig. 11 is shown below. The nodes which need to be checked for pointers are outlined inside the box.

![Sub-Tree to be searched](image_url)
IV.4. FELIX

This section describes Felix, an Algol 60 implementation of a tree structured FLM. This implementation has been used to obtain order of magnitude estimates of the execution time of an FLM and also to provide data used in comparing the memory requirements of the various FLM representations (see 8). A cycle consisting of an update, the forgetting of a trace and a prediction typically takes about 17 milliseconds. A listing of the program which is written in Algol 60 for the Burroughs B6700 is given in the appendix (A.3). The structure of Felix was proposed in (Cleary, 1973) and (Cleary, 1974). The program was written by early 1975 and the first results from it reported in (Cleary, 1975a).

All the techniques used in Felix have been explained in the preceding sections. However, Felix as a practical implementation contains combinations of and tradeoffs between these techniques so as to achieve a good balance of execution times and storage usage.

In what follows, I will explain the detailed structure of Felix and derive expressions for its execution times and storage requirements. I will also report actual values for these obtained from runs using Felix.

IV.4.1. Memory Structure

Felix is a tree structured FLM which is entirely resident in core. The bulk of the tree is based on the simple tree structure introduced in 2.3. However, it also incorporates a symbol table of events (2.5) in the form of a binary tree (2.4). This provides a good
compromise between decreased execution times and increased storage usage. A network as described in 3.2 is also incorporated into the Felix memory.

Each memory location consists of two pointers, called the value and next pointers, together with a number of single bit flags. The flags are concerned with carrying 'reward' and 'pain' information and I will not consider them further. This arrangement enables each memory location to fit into a single 48 bit word on the B6700.

In the following exposition I will move from the root of the tree to the leaves, explaining the structure of the memory at each level. Fig. 13 summarizes the complete structure of the memory.
Fig. 13 Structure of Felix Memory
IV.4.1.1. Event Tree

Each event in Felix is composed of two six character literal fields corresponding respectively to pattern and action inputs. The literals describing these events are held in a binary tree immediately above the root. Events in the rest of the memory are held as pointers back into this event tree as explained in 2.5.

The use of a binary tree in this part of the memory trades a small increase in memory usage against a substantial decrease in the time taken to search up the memory. When searching up the tree by far the most branching (and so search time) can be expected immediately above the root. By using a binary tree, the search time there is reduced to the log of the branching ratio. However, the number of locations required at each node in the event tree is approximately doubled. The reasonableness of this strategy can be seen in the results reported in section 8. There, an event tree of some 100 events consumed about 5% of the total memory. So at the expense of a 2.5% increase in memory size the search time is decreased from 100 steps to log2(100) ~ 7 steps. Little would be gained by using a binary representation higher in the tree as the branching ratios there varied from only 6 down to 1.2.

Because of the amount of information held at a node in the event tree, it has to be spread over a number of memory locations. The diagram below shows the structure of a typical event node (the notation used is described in Fig. 13). This diagram can be compared with the simpler structure illustrated in Fig. 10 and described in section 2.5.
Each location in the diagram has been given a label depending on its function (E, Binary, etc). I will now briefly explain the function of each type of location:

E - (Event, used for gaining access to the event node, all pointers from elsewhere into the event tree point to these locations;

Binary - holds two pointers to other nodes in the event tree whose literal values are greater or less than the literal at this node;

C - points to a six character literal which occupies the entire Char location;

Back - (used only on nodes holding patterns) points back to the E location of the action down the tree and up into the tree above this node.

IV.4.1.2. Branches

In the bulk of the tree each node contains three pointer fields; to the next node on this level, to the node up one level in the tree and to the event tree. Because in Felix each location can hold only two pointers these fields are split between two locations as shown in the
fragment of the tree below:

The locations labelled U (Up) point to the next node at the same level (another U location) and to the S (String) location. This points in turn to the event which labels this node and up one level to a U location.

IV.4.1.3. Leaves and Nets

Felix incorporates a network structure as described in section 3. However, rather than labelling each leaf and its corresponding net pointer with an event, it uses the fact that all net pointers to the same node will be labelled with the same event. This can be seen in the diagram below which shows two traces with their network and tree representations:

Note that the event "d" which labels both transitions is also the event at the foot of the tree leading to b c d - . Felix holds a
pointer at each node which is pointed at by a net pointer. This points down to the event at the foot of the tree and so saves some storage as there are fewer such nodes than leaves.

The diagram below shows the structure of Felix at the topmost level of the tree:

![Diagram showing Felix's structure]

The functions of the types of location are:

- **S-T** - (String-Transition) like an S location it points to the event labelling the node and also to the Down location;
- **Down** - as explained above this points to the event at the foot of the tree;
- **T** - (Transition) there is one such location for each leaf which has a net pointer to an S-T location and a pointer to the next leaf.

### IV.4.2. Storage Usage

Using the notation of previous sections, the total number of locations used by Felix can be calculated as follows:

- each node in the tree has an associated U and S location contributing 2L locations;
- each node at level N in the tree has a Down location contributing \( L_N \) locations;
- each leaf occupies a single T location giving T locations altogether;
the event tree is found in practice to be a small part of the total memory (usually ~ 5%) so any extra storage above that counted in the $2L$ term above can be neglected.

The total number of memory locations then is:

$$2L + L_N + T.$$ 

Each memory location contains two pointers of $p$ bits each so the number of bits per trace is:

$$b = \frac{2p(2L + L_N + T)}{T}$$

$$= 2p(2L/T + 1/a + 1).$$

(This neglects the extra flags carried in each location so that the expression can be directly compared to other FLM representations.)

Taking the worst case, $a = 1$, the expression above gives the bound:

$$b \leq 4p(L/T + 1)$$

and for a very large, $b \geq 4p(L/T + 1/2)$.

In Felix $p = 16$ and $b \sim 150$ bits (see section 8 for more details).

IV.4.3. **Execution Times**

Felix is written in B6700 extended Algol (Burroughs Corp., 1974) and requires less than 1000 lines of code and 4K bytes of memory when compiled. Unfortunately, a direct comparison of its compiled size and execution times with other computers is difficult as the unusual architecture of the B6700 can lead to compact machine code and can also strongly affect the execution times (Organick, 1973).

The execution times quoted below were obtained using inputs from the Blindman environment (Heads, 1975a, b) and a standard compiler option (Burroughs Corp., 1974, D-12). The runs included FLMs of orders
2, 4, 6 and 8 and input sequences of up to 15000 events. Section 8 gives details of the memory structures resulting from some of these runs.

Typical execution times for the parts of Felix are (in milliseconds of CPU time):

- Updating - 6
- Forgetting - 8
- Predicting - 3

giving a total for one FLM computer cycle, when the memory is full, of 17 milliseconds. Bearing in mind the comments at the beginning of this section these times are guides only to the relative speeds of the various parts of the system. I will now consider each of these times individually.

IV.4.3.1. Updating

The updating time remains relatively unchanged at about 6 milliseconds for a wide range of memory sizes (up to 5000 traces) and for memories of order 2 to 8. (This observation is validated by the uniformity of the execution time estimates given in Table 7, section 8). It increases to 10 milliseconds, however, in nearly empty memories when almost every update involves the creation of an entirely new trace.

The major parts of the update are the scan of the event tree and then the scan of the upper part of the tree. As the event tree is a binary tree, the time to traverse it will be proportional to the log of the number of events. The time to traverse the remainder will be proportional to half the average number of branches at each node: that is, the total time will be \( \log_2(a_1) + 1/2 \sum_{i=2}^{N} a_i \).
It was found that in memories where there were about 100 events \((\ln 2(100)^{-7})\) and where 1/2 \(a\) was between 5.5 and 7 (see section 8) then the times to traverse the two parts of the tree were about equal at 2.5 milliseconds, in good agreement with the predicted times.

IV.4.3.2. Forgetting

The forgetting time of 8 milliseconds is close to the update time of 6 milliseconds. This has been achieved, however, only by modifying an initial version of the forgetting algorithm. Originally traces were chosen at random by first traversing the event tree at random and then traversing the rest of the tree randomly (2.3.4 describes the algorithm to do this). It was found that the time to do this was between 30 and 45 milliseconds, most of which was spent choosing a random path through the event tree. To speed this process a circular list of nodes has been set up. Each location on this list points to an event node and to the next node on the list. A 'circle' pointer into this list is used to select events one at a time. This arrangement is illustrated below:

To choose a random trace the process is:
step circle to the next node in the list;
start at the event pointed to by this node and move up the tree from there, choosing branches at random (as in 2.3.4).
This short circuits the lengthy search of the dictionary and reduces the forgetting time to 8 milliseconds.

IV.4.3.3. Prediction

The prediction time of 3 milliseconds is surprisingly shorter than the update time. This is explained because the prediction process itself does not have to scan up the tree structure. Rather, the update algorithm retains a pointer to the best match node after the buffer has been updated. As a result the prediction routine need only scan the sub-tree above this node. Because in the memories tested most buffers matched over nearly their full length and because the branching ratios near the top of the tree were small (<2) these searches were not very extensive.
An FLM-like memory using a hashing technique for storing traces was originally suggested in (Andreea, 1975) under the name of Mew-PUSS. (A good introduction to hashing techniques is given in (Knuth, 1973c).) Various refinements and improvements have been suggested in (Cleary, 1975b), (Andreea, 1977) and (Palfi, 1977, p19ff). Very briefly, Mew-PUSS calculates a pseudo-random address from the tail of a trace and stores the head of the trace at that address together with information to identify it. Because the identifying information is insufficient to uniquely specify the trace, Mew-PUSS and its variants may predict events which have never been stored with the trace. I will show that the probability of this happening can be kept very small.

I will first describe a hashing technique which is free of the errors which can occur in Mew-PUSS and derive bounds for its execution times and storage requirements. I will then go on to consider Mew-PUSS and derive an expression for the expected rate of erroneous predictions.

IV.5.1. The Basic Algorithm

The algorithm described below is based on the linear probing algorithm in (Amble and Knuth, 1974).

Given a memory of M locations, numbered 1 to M then I will associate with each possible trace a hash address in this range calculated from its tail. To do this I will associate with each trace, T, a transformation $t(T)$ where

$$0 \leq t(T) < 1$$
and \( t(T_1) = t(T_2) \) implies that the tails of \( T_1 \) and \( T_2 \)
are equal. \( t(T) \) is the hash key of the trace \( T \). \( t \) scrambles the traces
in such a way that the tail of the original trace can be deduced from
the result. For effective operation of the algorithm below there should
be little correlation between the values of \( t \) even when applied to
traces which are similar.

A hash address for each trace is obtained by taking the integer
part of the expression \( M \cdot t(T) + 1 \). This address will be in the range 1
to \( M \) (inclusive) and defines the position in memory where the trace is
to be stored. Both the value of \( t(T) \) and the head of the trace will be
stored there.

It is possible that more than one trace will have the same
hash-address. If this is the case the extra traces will be stored in
the immediately following locations up the memory (this is one of many
ways of resolving such conflicts, it is referred to as 'linear
probing'), which may in turn displace entries at these locations and so
on. As a result of the way the hash address is calculated, the stored
values of \( t(T) \) will increase as one moves up the memory. So to find an
entry, \( T \), in the memory, one starts at its hash address and scans up
the memory. The scan stops when either an empty location or one
containing a hash key greater than \( t(T) \) is found, for if the entry
itself is present then it would be located before then.

The following algorithms describe how to obtain a prediction
from the memory and how to add a trace to the memory. Note that the
prediction is that obtained by a full match over the trace not a best
match prediction.
Let the memory be described as follows:

```plaintext
var Memory : array [1..M] record
    HashKey : HashType;
    value : event;  {0 implies an empty location}
end;
```

The algorithm to obtain a prediction for the trace, T, is as follows:

```plaintext
k := t(T);  {calculate hash key}
i := [M * k + 1];  {obtain hash address}
searchloop: with Memory [i] do
    Begin
        if value = 0 or key > k
           then exit from routine;
        if HashKey = k
           then add value to set of events predicted;
           i := i + 1;  {keep looking further up memory}
    End
    goto searchloop;
```

To insert a trace, T, and its head into the memory the algorithm is as follows:

```plaintext
x := head of T;
k := t(T);  {calculate hash key}
i := [M * t(T) + 1];  {obtain hash address}
insertloop:
    with Memory [i] do
        Begin
            if value = 0 then
                Begin {empty location, insert new trace and exit}
                    value := x;
                    HashKey := k;
                    exit from routine;
                End
            else if HashKey = k and value = x
                then {already here, don't do anything}
                    exit from routine
            else if HashKey > k then
                Begin {swap current value in memory and continue}
                    swap k and HashKey;
                    swap x and value;
                End;
            i := i + 1;
        goto insert loop;
```
IV.5.2.  Best Match of Traces

The prediction algorithm above deals only with a fixed length match over the entire string. However, it is very easy to extend this and the insertion algorithm to allow for a best match. All that is necessary is that each sub-trace with a tail of length 1, 2... order of the FLM be stored separately. (Because t scrambles traces these sub-traces will tend to have quite different hash addresses). The algorithm for a best match prediction is then:

For l := order of the FLM downto 1 do
Begin
obtain a prediction for trace
T(1), T(i-1)...T(1) T(0) as above;
if prediction not empty exit from routine;
End;  [end of loop];
If no prediction at all obtained then predict all possible events;

Similarly to completely insert a trace:

For l := order of the FLM downto 1 do
insert trace T(1),...T(i) T(0) into memory as specified above;

Because the sub-traces are not linked as the nodes in a tree representation, one could save memory storage by omitting some of the sub-traces. For example, Mew-PUSS completely omits all except the longest traces, effectively doing no best matching at all. However, one might use trace lengths of say, 3, 4 and 6 events; reasoning that a tail of two events was too short to provide any useful information and that the difference between a tail of 5 and 6 events was too small to warrant storing both. The utility of such an arrangement would depend very much on the particular application, however.
IV.5.3. Variable Length Hash Keys

As it is possible to reconstruct the tail of a trace from its hash key, it is necessary that the hash key contain as many bits as the events in the tail. So, if the largest possible trace in the memory were \( L \) events long and each event required \( e \) bits to specify it, then the hash key must be at least \( L \cdot e \) bits long. So, compared with each predicted event which requires only \( e \) bits, the storage for the hash keys can be large.

Two observations enable reduction of the storage used by the keys. The first is that traces with shorter tails don't need hash keys which are as long as the maximum. If the keys in memory could have their lengths varied then these shorter keys could be stored instead. The second is that when the same trace predicts more than one event, the key for the trace needs only to be stored once.

To take advantage of these observations the individual locations in the memory should be made rather smaller; just large enough to hold one event and a flag to say whether or not the rest of the location is holding part of a hash key or an event.

Using this scheme each hash key, when it is stored, will occupy a sequence of locations for the hash key, followed by one location for each event predicted by the trace. This arrangement is illustrated in Fig. 14. This shows a memory where each location can hold a single digit or an event. The fraction \( t(T) \) has precisely the same number of digits as there are events in the tail of \( T \).
### Fixed length hash keys

<table>
<thead>
<tr>
<th>Location</th>
<th>t(T)</th>
<th>Traces</th>
<th>t(T)</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>615</td>
<td>A a B b C</td>
<td>.8253</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Variable length keys

<table>
<thead>
<tr>
<th>Location</th>
<th>t(T)</th>
<th>Traces</th>
<th>t(T)</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>615</td>
<td>A a B b Z</td>
<td>.8253</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* - part of hash key

The left part shows how the memory might be laid out with a fixed length hash key.

The memory on the right uses variable hash keys and merges successive keys which are the same.

Note that the individual locations on the right are much smaller than the left. In both cases a memory of 1000 locations has been used.

---

**Fig. 14 Layout of Fragment of Hashed Memory**
IV.5.4. **Storage Requirements**

In estimating the storage required for each trace I will presume that a full best match is being used, that is each sub-trace of length 1, 2..etc. is being stored. I will also presume that the hash key for a trace occupies the same number of locations as there are events in its tail.

Let the number of traces with tails of length \( i \) be \( T_i \), \( 0 \leq i \leq N \), and the number of different tails of length \( i \) be \( L_i \), \( 0 \leq i \leq N \), then the total number of memory locations occupied will be

\[
M' = \sum_{i=1}^{N} i.L_i + T_i
\]

Each memory location requires \( e + 1 \) bits (\( e \) for an event plus 1 for the hash key/event flag) so the number of bits per trace will be

\[
b = (e + 1)\left( \sum_{i=1}^{N} i.L_i + T_i \right)/T.
\]

If the FLM obeys the restriction that every event in the tail of the buffer must have appeared in its head (the same restriction applied to FLMs which can be represented as a net) then every tail of length \( i \) will correspond to a trace with a tail of length \( i-1 \) that is

\[
L_i = T_{i-1}.
\]

This enables (2) to be simplified to:

\[
b = (e + 1)(1 + L/T + \sum_{i=1}^{N} i.L_i /T).
\]

An upper bound can be set on \( b \), as

\[
\sum_{i=1}^{N} i.L_i \leq N.L_i; \text{ that is}
\]

\[
b \leq (e + 1)(1 + (N + 1).L/T).
\]

Also \( \sum_{i=1}^{N} i.L_i \) will be a minimum when \( L_1 = L_2 = \ldots = L_N = 1/N.L \), then

\[
\sum_{i=1}^{N} i.L_i = (N + 1)/2.L, \text{ so}
\]

\[
b \geq (e + 1)(1 + (N + 3)/2.L/T).
\]
IV.5.5. **Execution Time**

Estimates for the execution time of the algorithm in 5.1 have been obtained in (Amble and Knuth, 1974). It is shown there that the expected number of memory locations examined during a search is $1/2(1 + 1/(1-d))$ where $d$ is the fraction of the memory locations actually occupied. In the notation of the last section $d = M'/M$. The expected number of locations examined when a trace is inserted is $1/2(1 + (1-d))$.

These numbers rise very quickly as the memory becomes full and $d$ approaches 1. However, provided the memory is never permitted to become more than say 75% full then these two values will not exceed 2.5 and 8.5 respectively. So the overall time for a full best match update or prediction can be kept down to some small constant times the order of the FLM.

If the entire memory is held on a disc backing store then whenever a hash address is calculated the record containing it and one or two succeeding records can be read into central store. The probability that any of the locations outside this range of records will need to be examined is very small, so all manipulations of the memory can be done entirely in central store. The overall execution time then will be the time to read (and possibly write) a disc record times the order of the FLM.
IV.5.6. Calculation of $t(T)$

It is not immediately apparent how one can calculate $t(T)$ so that it is both random and its length is adjusted according to the length of $T$. I will now describe such an algorithm which has the additional advantage that the hash key will tend to shorten as the total number of different events in the FLM decreases. Fig. 15 gives an illustration of the working of the algorithm. This algorithm is new and has not been reported elsewhere.

Let each event in the FLM be assigned a number occupying $e$ bits. An effective way of doing this is to allocate each event a number when it is first seen by the FLM. For the sake of example I presume $e = 4$ but a more likely value in practice might be, say, $e = 15$.

Given a trace, take each event in its tail, convert it to a number and lay it out in a single column of binary digits. This will result in $N$ columns side by side forming a rectangular array of bits, $e$ by $N$. Take the rows of this grid, starting with the row consisting of the least significant bits of the event numbers, and lay them out in a single string of $eN$ bits.

Provided the value of $N$ is known, the original events can easily be reconstructed by taking successive chunks of $N$ bits from the string and reconstituting the grid. So the string forms a unique key for the trace. However, traces with lengths from 1 up to the order of the longest FLM may occur. To remove any ambiguity the string is prefixed with a fixed number of bits holding the length of the trace. In the example I have used three bits, which allows FLMs up to order 8. In practice three or four bits should be sufficient. Now insert a point to the left of the string and discard any trailing zeros and one has a
binary fraction in the range 0 to 1 which is a unique hash key for the trace.

The key will not be completely random however; mainly because the initial bits specifying the length of the trace will tend to cluster at a few values. So when a hash address is calculated some areas of the memory will be more crowded than others. This is easily remedied by multiplying the fraction by an odd constant. The fractional part of the result is another unique key, a scrambled version of the first. In the example I have used 9 as the constant, however, a large odd number comparable in length to the key provides better scrambling.

Finally split the string into chunks of e bits and one has a key ready to insert into the memory. It can be seen that the size of the key will grow and contract with the length of the tail. Also, if there are only a few different events in the memory then the numbers coding for the events will be small, the top rows of the grid of bits will then tend to be zero and the hash key correspondingly shorter. The same effect will be achieved if the most frequent events are coded with small numbers.
Take trace \( t \);

Code events as numbers;

Lay bits of event numbers out into rectangular grid;

Select rows of grids and arrange into string;

Prefix length of tail (use 3 bits in this case);

Form binary fraction and discard trailing zeroes;

Multiply by some odd constant (in this case 9);

Take fractional part and split into chunks of \( e \) bits.

Fig. 15 Calculation of \( t(T) \) for Variable Length Hash Keys
IV.5.7. Deletion and Forgetting

The random deletion of individual traces from the memory is very easy. All that is necessary is to set the memory location holding the trace to null and possibly adjust the location of any immediately following traces. This adjustment will be needed if these following traces are in locations higher than their hash address, so if the memory is to remain intact they must be moved back down the memory.

The following algorithm will perform such a deletion; (it uses the memory structure of 5.1):

\[
i := \text{address of trace being deleted;}
\]
\[
\text{state := compressing; repeat}
\]
\[
\text{Memory [i].value := 0; \{clear deleted or shifted location\}}
\]
\[
\text{with Memory [i + 1] do}
\]
\[
\text{if value = 0}
\]
\[
\text{then state := finished \{next location empty\}}
\]
\[
\text{else}
\]
\[
\text{Begin \{see if next location needs to be shifted down\}}
\]
\[
k := \lfloor M \times \text{HashKey + 1} \rfloor; \{\text{hash address}\}
\]
\[
\text{if } k > i \text{ then state := finished}
\]
\[
\text{else Memory [i] := Memory [i + 1];}
\]
\[
\text{End;}
\]
\[
i := i + 1;
\]
\[
\text{until state = finished;}
\]

It is a straightforward elaboration of this algorithm to allow for variable length keys.

The update algorithm of 5.1 has an insertion loop whereby following entries in the memory are displaced upward whenever a new trace is entered. This suggests a very simple forgetting and deletion scheme. A count can be kept of the number of times this loop is performed. When this count exceeds some constant the loop can be exited, effectively deleting the trace currently being swapped. When the memory is empty, the loop will be performed only once and no traces will be forgotten. As the memory becomes full, however, the number of
loop iterations will increase and cause more deletions. Eventually, the memory will reach a steady state where every insertion is accompanied by a deletion.
IV.6. AN ANALYSIS OF MEW-PUSS

If the hash key in the memories I have just been discussing were reduced to some small constant length then memory would be saved at the expense of events occasionally being predicted when they ought not. The Mew-PUSS algorithm (Andreea, 1977) uses this technique of short hash keys to save storage. I will now specify a simple form of the Mew-PUSS algorithm and obtain an expression for the probability that an event will be mis-predicted. I will show that this probability can be kept low, approaching zero as the length of the key becomes sufficient to hold an entire hash key.

I will also examine two modifications to the basic algorithm, designed to alleviate the effects of mis-predictions. I will show that in each case increasing the length of the key will result in a similar and usually superior result. Results which are similar to those presented here and which apply to an earlier version of the Mew-PUSS algorithm may be found in (Cleary, 1975b).

For clarity in what follows I will refer to the shortened form of a hash key as a tag.
IV.6.1. The Mew-PUSS Algorithm

To store a trace in the Mew-PUSS memory the following procedure is used:

loop: calculate a tag and a hash address from the hash key;
    if the location at the hash address is empty
        then store the tag and the head of the trace there;
        exit from the routine;
    else if the tag at the hash address equals the calculated tag
        and the event equals the head of the trace
        then exit from the routine
    else goto loop;

For effective operation of the algorithm the tags should be uncorrelated with the hash addresses. Any of the standard techniques in (Knuth, 1973c) or (Amble and Knuth, 1974) might be used to calculate the hash addresses on successive loops. For example, adding one each time would give the 'linear probing' sequence used in the last section.

To obtain a prediction from the memory the following algorithm is used (the hash and tag calculations are the same as above):

loop: calculate a tag and a hash address from the hash key;
    if the tag at the hash address is equal to the calculated tag
        then add the event there to those predicted;
    if the location at the hash address is empty
        then exit from the routine;
    goto loop;

IV.6.2. Expected Error

When examining a sequence of hash addresses for a particular trace, one may encounter locations stored into by other traces. If by chance the tag stored should equal that calculated, then the event will be predicted erroneously. The probability of such an error will depend on the number of locations encountered which were stored by other
traces and the probability that the stored tag equals the calculated one.

The following notation will be used below:

\( t \) - the number of bits in a tag;
\( d \) - the ratio of the number of stored events to the total number of locations available;
\( E \) - expected error rate.

The probability that two tags will be equal is \( 2^{-t} \) because the values of different tags are uncorrelated. I will assume that \( t \) is large enough that \( 2^{-t} \) is very small.

When following the sequence of hash addresses for a particular trace, the number of locations examined will be the number of events predicted by the trace plus a number determined only by \( d \). The probability of \( n \) extra events is just \( (1-d)d^n \) and the expected number of errors is \( n \cdot 2^{-t} (1-d)d^n \). This gives the equation

\[
E = \sum_{n=1}^{\infty} \frac{(-t)^n}{n!} n (1-d)d^n
\]

\[
= 2^{-t} -1 \quad d(1-d).
\]

Provided \( d \) does not exceed say 3/4 then

\[
E \leq 3.2^{-t}.
\]

IV.6.3. Storing Events More Than Once

The Mew-PUSS algorithm in (Andreae, 1977) seeks to reduce the probability of an error by storing each predicted event three times. Only if an event is found three or more times during a search will it in fact be predicted. I will develop an expression for the expected
error rate and show that for the same storage usage it is better to
lengthen the tags rather than store the event more than once. In what
follows u will be used for the number of times each event is stored.

IV.6.3.1. Algorithms

The storage and retrieval algorithms are now modified to:

Storage; Calculate a hash key from the tail of the trace;
count := 0;
loop: Calculate a hash address and tag from the hash key;
if the location at the hash address is empty
then store the tag and the head of the trace there;
count := count + 1
else if the location contains the calculated tag and the
head of the trace
then count := count + 1;
if count < u then go to loop;

Retrieval; Calculate a hash key from the tail of the trace;
loop: Calculate a hash address and tag from the hash key;
if the tag at the hash address is equal to the calculated tag
then add the event at the hash address to a list of
those predicted;
if the location at the hash address is empty
then go to exit;
go to loop;
exit: predict all events which are on the list
u or more times;

IV.6.3.2. Expected Error

Using the notation from earlier and as well:

t' - number of bits to store a tag for this case;
e - number of bits to store an event;
p_i - probability of the i'th event occurring.

Using the same reasoning as for (1) the probability of the i'th
event occurring erroneously u or more times when n extra locations are
examined is:

\[ (1 - d) d p_i \left(2^{-t'}\right)^u(n) \]

This is an upper bound because cases where the event occurs u +
1 or $u + 2$ etc. times are counted more than once. However, the probability of an event having the correct tag $u + 1$ times is much less likely than this only happening $u$ times, so the approximation should be a good one. Summing over $n$ and all possible events:

$$E < (1 - d) \sum_{n=0}^{\infty} \frac{n^n}{d p_i^n} 2^{-u(n)}.$$

Rearranging and summing gives:

$$E < 2^{u - 1} \sum_{u} \left( \frac{-t'}{2} \right)^u /u! \cdot \sum_{i} p_i.$$

As a check substituting $u = 1$ and recalling that $\sum_{i} p_i = 1$ gives (1) back again.

A lower bound can be obtained for $E$ by considering only the cases where an event is erroneously predicted exactly $u$ times. This gives the bound:

$$E > \sum_{i} (1 - d) \sum_{n=0}^{u} \frac{n^n}{d p_i^n} (1 - p_i)^{n-u} 2^{-u(n)}.$$

Rearranging and summing gives:

$$E > \frac{d}{u!} (1 - d) \sum_{i} \frac{u}{p_i} (1 - d(1 - p_i)).$$

Expanding $(1 - d(1 - p))$ into a Taylor series and taking the first two terms gives the weaker bound:

$$E > \left[ \frac{-t'}{2} \frac{d}{(1 - d)} \right] u /u! \left[ \sum_{i} \frac{u}{p_i} - (u+1)/(1-d) \sum_{i} \frac{u+1}{p_i} \right].$$

(For the inequality to be retained when the higher terms of the expansion are neglected it is necessary that $(u+1)/(1-d).p_i < 1$ for all $i$. For the typical case where $u = 3$, $p = 3/4$, and so $(u+1)/(1-d) = 16$, this should be true.)
IV.6.4. Comparison of Expected Error

I will now compare the expected error rates when each event is stored once and when it is stored $u$ times. To enable a true comparison the length of the tags in the two cases should be adjusted so that each trace uses the same amount of storage, this will occur when

$$t + e = u(t' + e).$$

(4)

For example if $t' = e$ then either $u$ locations containing a tag one event long together with an event or a single location containing a tag $2u - 1$ events long together with an event would be used.

The sum $\sum_{i}^{u} p_i$ will be as small as possible when all the $p_i$ are equal; that is $p_i = 2^{-e}$. If this is the case then

$$\sum_{i}^{u} p_i = 2^{u-1} \cdot \frac{e}{2} \cdot (2)^{-e(u+1)} = 2.$$

In practice some of the $p_i$ will be larger than this and will contribute increasingly to $\sum_{i}^{u} p_i$ as $u$ is increased.

However, making the assumption that $p_i = 2^{-e}$ will favour the modified algorithm. Substituting in (2) the result is:

$$E < 2 \cdot \frac{d \cdot u}{1-d} /u!.$$

(5a)

For the original algorithm with a single tag one obtains from equation (1)

$$E = 2 \cdot \frac{d}{1-d}. $$

(5b)

But from (4) $t = ut' + eu - e$ so the power of 2 in (5a) and (5b) is equal and the only difference between the two equations is in the terms $(d/(1-d))^u /u!$ and $d/(1-d)$. Assuming $d = 3/4$ the value of this expression for $u = 1, 2$ and $3$ is $3, 4.5$ and $4.5$ respectively.
So by making an assumption about the probabilities of events occurring which strongly favours the modified algorithm (where events are stored more than once) its error rate is found at best to be only comparable with the simpler form using a single longer tag.
This section describes an FLM implementation, called Simple. It is particularly suitable for implementation by parallel hardware and is very economical of storage. It will be shown that in the best case one event per trace will be required for storage and in the worst case the number of events per trace is a slowly increasing (logarithmic) function of the order of the FLM. On a serial computer the execution time is proportional to the size of the memory. However, parallel hardware described in (Beaven and Lewin, 1972) can reduce this to a time proportional to the order of the memory. As with the net representation described in section 3 the Simple system presupposes that every event which appears in the head of the buffer trace is then pushed down into the buffer tail. This section and the supporting derivations in the appendix are taken directly from (Cleary, 1974).

IV.7.1. The Simple Algorithm

To understand the operation of Simple it is helpful to consider an even more transparent scheme. This consists of a pile of traces each with a tail of length $N$. Each time the memory is updated the new trace is placed on top of the pile and any other occurrence of it is deleted from the pile. If the pile (memory) is full then the bottom trace is removed, (note that this is just the ladder forgetting scheme proposed in section 1).

The operation of Simple is the same as this except that the traces are overlapped instead of being held separately. In order to indicate where the heads of these overlapping traces occur, each event
in the memory is accompanied by a flag. This flag is set for the heads of current traces. Each time the memory is updated the head is shifted onto the top of the memory and its flag is set. If the buffer only is updated then the new event is shifted onto the top of the memory with its flag reset. Note that the top few events in the memory hold the current buffer.

Fig. 16 gives an example of the pile of traces generated by an input sequence to an FLM of order 2. The corresponding Simple memories are given with the flags indicated by a 1(set) or 0(reset).
<table>
<thead>
<tr>
<th>Inputs Patterns</th>
<th>Actions</th>
<th>Snapshot of Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Pile of Traces</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Simple Memory</td>
</tr>
<tr>
<td>b</td>
<td>X</td>
<td>Y b X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y b X</td>
</tr>
<tr>
<td>b</td>
<td>Y</td>
<td>X b Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y b X b Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y b X</td>
</tr>
<tr>
<td>b</td>
<td>X</td>
<td>Y b X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Y b X b Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X b Y</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 0 1 0 1</td>
</tr>
</tbody>
</table>

Fig. 16 Pile of Traces and Simple Memory
I will now specify the details of the operation of Simple in terms of four Pascal-like routines. Routines with the same names and functions are used in the parallel hardware version described later. The memory is composed of two parallel arrays, one, Sym is an array of events and the second, Flag, marks those events which are heads of active traces. There are M locations in each array numbered 1 through M. In Pascal notation:

\[
\text{var Sym : array [1..M] event;}
\]
\[
\text{Flag : array [1..M] (active, inactive);}
\]

IV.7.1.1. Update

This routine adds a new event to the top of the memory and sets its activity flag (Fig. 17 contains an example of its operation.) Any traces which duplicate the new trace added to the memory are deleted by having the flag at their head set inactive. This may leave some locations in the memory which are part of no trace. The routine Getspace is used to delete one of these and compress the memory. (If there are no unused locations then the entire memory is shifted down one location.)

Procedure Update (NextSymbol : event);
Begin
    Flag[i] := inactive for all i where
    Sym[i] = NextSymbol
    and Sym[i-k] = Sym[M - k + 1], 1 \leq k \leq N;
    Getspace (nextsymbol);
    Flag[M] := active;
End; {of Update}
Initial memory  
Sym : C B C D C B C D  
Flag: 1 1 0 0 1 1 1 1

Call to Update 
Duplicate strings reset:  
Sym : C B C D C B C D  
Flag: 1 1 0 0 0 1 1 1

Call to Getspace and 
insertion of new cell:  
Sym : C B D C B C D C  
Flag: 1 1 0 0 1 1 1 1

Note: A 1 indicates an 
active symbol (Flag=1).

Fig. 17 Example of Update of Simple Memory
Fig. 17 shows the changes that occur in a simple memory during a call to Update. It is assumed that the following sequence of nine actions constitutes the input: C B C D C B C D C

The memory, which is of order 2 and has eight locations (M=8), is shown after the first eight of these events has been input. The rest of Fig. 17 shows the changes in the memory during the next call to Update. It is found that there is one symbol in memory whose Flag can be reset (at position 4). Getspace then moves the memory from 3 upwards down one position and the next event(C) is placed in the topmost memory position thus freed.

IV.7.1.2. Bufferupdate

This routine inserts a pattern into the top of the memory with its flag reset. Getspace is called first to provide space for the insertion.

Procedure Bufferupdate (NextSymbol : event);
Begin Getspace (NextSymbol); Flag [M] := inactive;
End;

IV.7.1.3. Getspace

This locates the first (topmost) event in the memory which is unused; that is, part of no trace. The memory elements above this position are then moved down one position to make room for nextsymbol in the topmost position. If there are no unused memory locations then the entire memory is moved.
Procedure Getspace (NextSymbol : event);
    var i,j:0..M;
    k:0..N;
    state:(scanning,found);
    Begin [find unused location]
      i := M - N; {N is the order of the FLN}
      state := scanning;
      while state = scanning and i > 1 do
        Begin
          if Flag [i + k] = inactive, 0 ≤ k ≤ N
            then state := found
            else i := i - 1;
            [move memory]
            For j := i to M - 1 do
              Begin
                Sym [j] := Sym [j + 1];
                Flag [j] := Flag [j + 1];
              End;
            [Insert new symbol]
            Sym [M] := nextsymbol;
        End; {of while}
    End; {of Getspace}

IV.7.1.4. Predict

The memory is searched and the heads of all strings matching over the greatest length are placed in the set of predictions.

Procedure Predict (var result : set of event);
    var l : 0..order;
    i : 1..M;
    Begin
      result := [] ;{clear prediction}
      l := 0; {match of length 0 at start}
      For i := N + 1 to M do
        if Flag [i] = active and
          Sym [i-k] = Sym[M-k+1], 1 ≤ k ≤ l
          then
            Begin [string matches over current length]
              result := result + [Sym [i]];
              {see if any longer match}
              while l < N and Sym [i-l-1] = Sym [M-l] do
                Begin l := l + 1; result := [Sym[i]];
              End;
            End;
        End; {of predict}
IV.7.2. **Hardware Implementation of Simple**

(Beaven and Lewin, 1972) describe a parallel symbol-processing system (SPM). After a brief introduction, some small extensions to the system will be given to enable it to be used as a Simple memory. A short explanation of how the various Simple algorithms can be implemented in terms of the SPM together with the detailed SPM routines follow.

IV.7.2.1. SPM

The SPM illustrated in Fig. 18 has a string of memory/processor cells. Each cell has some processing logic, a symbol memory (of, say, 8 bits) and a single bit, the activity bit. The cells can communicate only with their adjacent neighbours and with various common bus lines. In particular, each cell can do the following:

- accept instructions from a common bus;
- test and set the activity bits of itself and its immediate neighbours;
- transmit and receive symbols from its neighbours and a common bus;
- send a single bit to a common output line, the REPLY line.

The output of the REPLY line can have the three values:

- 0 - no cells replying
- 1 - one cell replying
- 2 - two or more cells replying
A section of the SPM memory

A memory cell

Fig. 18 Structure of SPM. (Beaven and Lewin, 1972)
To handle the Simple system, it is necessary for one of the bits in the symbol (the flag bit) to be able to be set and reset and for the activity bit to be able to be set and reset by it. Three new SPM instructions have been added to do this:

SFFA set flag
RFFA reset flag
SAFF set activity from flag.

As with the majority of SPM instructions, these are only performed by cells whose activity bits are on.

IV.7.2.2. Instructions

The following instructions are used in the SPM routines for implementing Simple. Except for SAA, PALL and COA, the instructions are only obeyed by active cells.

Instruction Explanation

SFFA Set flag.
RFFA Reset flag.
SAFF Set activity from flag.
MAC Match symbol with incoming symbol on bus; if equal put a reply on the reply line.
MACK As for MAC, but reset activity if not equal.
MMACK Match symbol with incoming symbol on bus; if unequal put a reply on reply line, if equal reset activity.
PAM Shift all activity bits one cell to the left.
PAR Shift all activity bits one cell to the right.
PEAL Set activity bit on if right neighbour's activity bit is on; remain active if already so.
RAA  Reset all activities.
SAA  Set all activities.
SASL If active, shift symbol to the left.
SASR If active, shift symbol to the right.
SRA  Set activity of rightmost cell on.
SRNI Shift symbol right if right neighbour is inactive.
      All cells shifting output to reply line.
WAC  Store value on input bus with symbol.
COA  Complement activity.
PALL Set activity bit on if any cell to left is active.

The last two instructions have been added to the original instruction set. COA is very similar to SAA and PALL to SRA.

IV.7.2.3. Notation

The programs which follow are given in a pseudo-Pascal style, where the upper case mnemonics (e.g. PAL) are SPM instructions, lower case words (e.g. n) are variables outside the SPM memory, and lower case words underlined (e.g. if) are pseudo-Pascal keywords. The mnemonic REPLY gives the current state of the reply line. If an SPM instruction is followed by a variable, this indicates that the variable has been placed on the symbol bus to the SPM, e.g. MACK NextSymbol.

Because the contents of the individual cells in the memory cannot be accessed randomly the buffer is held separately, as a duplicate of the top events in the memory. It is declared as follows:

    var buffer : array [1..order] event;
IV.7.2.4. Simple Routines Using the SPM

Procedure Update (NextSymbol : event);
[add a new trace to the memory]
var l : 0..order;
Begin [reset activity of any trace equal to new trace]
SAA; SAFF; MACK NextSymbol;
[all traces with head equal to NextSymbol now active]
For l := 1 to order do
Begin PAL; MACK buffer [i]; End;
[the last cell in tail of each equal trace is now active]
For l := 1 to order do PAR; [move activity back to head]
RFFA; [reset the flag of active traces]
Getspace (NextSymbol); [put new event at right of memory]
SFFA; [set flag of new entry which is only cell left active]
End; {of Update}

Procedure Bufferupdate (NextSymbol : event);
[add a new pattern to top of memory]
Begin Getspace (NextSymbol); RFFA
End; {of Bufferupdate}

Procedure Getspace (NextSymbol : event);
[slide rightmost unused cell in memory and insert new event]
{the activity of the rightmost (top) cell is the only one left on}
var l : 0..order;
Begin
SAA; SASL; {move whole memory left one position}
COA; SFFA; {force flag of (empty) rightmost (top) cell}
{in memory on so that buffer is not deleted}
SAA; SAFF; {make all flagged cells active}
For l := 1 to order do PEAL; {make all used cells active}
COA; PALL; {leave all cells above rightmost unused cell active}
SASL; SAA; SASR; {move these cells left one and then whole memory}
{right one}
SAA; PAL; COA; {set rightmost (top) cell active, left this way}
WAC NextSymbol; {when routine exited. Store new event}
{update buffer}
For l := 2 to order do buffer [i] := buffer [i -1];
buffer [1] := NextSymbol;
End; {of Getspace}
Procedure Predict;
(all predicted events are left with their activity bits set on)
var i, l : 0..order;
Begin SAA; SAFF; {set active all flagged cells}
{move activities left one cell at a time, testing for a match}
l := 0;
repeat l := l + 1;
PAL; MAC buffer [l]; {test next event}
if REPLY <> 0 then MMACK buffer [l];
{leave only matching cells active}
until l = order or REPLY <> 2;
{stop at end of buffer or when no or 1 responding cells}
For i := 1 to l do PAR; {move activity back to head of trace(s)}
End; {of Predict}.

IV.7.2.5. Execution Time

All the loops in the above programs are restricted to at most \( N \) iterations (\( N \) is the order of the FLM). So the execution time is at most proportional to the order of the FLM. However, implemented on a serial computer, the execution time will be proportional to the memory size times the order.

IV.7.2.6. Storage

Estimates will now be made of the storage required for Simple. Two special cases will be investigated: one when all the inputs are patterns and the other when patterns and actions alternate.

A factor affecting the estimates is how deterministic the inputs are. For example, if all the inputs are actions and each input is determined by the previous \( N \) symbols, then the memory (of order \( N \)) will contain a single unbroken sequence of consecutive events. Each trace is then represented by one event in the memory. (In the case of alternating actions and patterns, a variety of patterns prevents such a single chain of symbols from building up). If, however, many heads can occur following each string (that is, the inputs appear strongly
non-deterministic) then the average number of symbols per trace becomes a slowly increasing function of the order of the memory. When alternating patterns and actions occur as inputs, the number of symbols per trace is approximately doubled.

Experimental evidence given below suggests that the estimates obtained are an upper bound on the actual requirements.

Simple compares most favourably with the other FLMS (in memory size at least) when the inputs are nearly deterministic. Then its use of the fact that the inputs are presented sequentially allows it considerable memory savings. It is also possible that Simple can achieve memory savings in situations where the memory size is very large: for it uses no internal memory pointers and in large memories the space needed to store a pointer may be substantially larger than that for an event.

Table 1 lists four approximations for the total number of memory locations per trace when the inputs are all actions. It also gives the notation used and a tabulation of these functions. The equation numbers are taken from their derivation in the appendix (A.2.1).

Table 2 lists the storage requirements for the case where there are alternating patterns and actions in the input. These results are derived in the appendix (A.2.2).
Table 1 Memory Usage in Simple - All Action Inputs

\[
M = T \left[ 1 + \frac{a}{a-1} \sum_{k=1}^{N-1} \sum_{k=1}^{N} \frac{k}{k+1} \left( 1-a \right)^{k-1} \left( 1-a \right)^{-k} \right]
\]

\[
M = T \left[ 1 + \sum_{k=1}^{N-1} \frac{k}{(k+1)(k+2)} + \frac{N}{N+1} \right]
\]

\[
M = T \left[ -0.427 + \ln \left( N + 3/2 \right)^2 + \frac{N}{N+1} \right]
\]

\[
M = T \left[ -0.427 + \ln \left( N + 5/2 \right) + \frac{N}{N+1} \right]
\]

Values computed from equations (13), (6b), (16), and (17) are tabulated below.

<table>
<thead>
<tr>
<th>N:</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>N:</td>
<td>1.25</td>
<td>1.46</td>
<td>1.63</td>
<td>1.79</td>
<td>1.93</td>
<td>2.05</td>
<td>2.15</td>
<td>2.25</td>
<td>2.34</td>
<td>2.92</td>
</tr>
<tr>
<td>a=2</td>
<td>1.33</td>
<td>1.60</td>
<td>1.80</td>
<td>1.98</td>
<td>2.13</td>
<td>2.26</td>
<td>2.37</td>
<td>2.48</td>
<td>2.58</td>
<td>2.66</td>
</tr>
<tr>
<td>a=3</td>
<td>1.38</td>
<td>1.66</td>
<td>1.88</td>
<td>2.07</td>
<td>2.22</td>
<td>2.35</td>
<td>2.47</td>
<td>2.58</td>
<td>2.68</td>
<td>2.77</td>
</tr>
<tr>
<td>a=5</td>
<td>1.40</td>
<td>1.70</td>
<td>1.92</td>
<td>2.11</td>
<td>2.27</td>
<td>2.41</td>
<td>2.53</td>
<td>2.64</td>
<td>2.73</td>
<td>2.82</td>
</tr>
<tr>
<td>a=10</td>
<td>1.45</td>
<td>1.76</td>
<td>2.01</td>
<td>2.20</td>
<td>2.37</td>
<td>2.50</td>
<td>2.63</td>
<td>2.74</td>
<td>2.84</td>
<td>2.93</td>
</tr>
<tr>
<td>a=20</td>
<td>1.48</td>
<td>1.80</td>
<td>2.04</td>
<td>2.24</td>
<td>2.41</td>
<td>2.54</td>
<td>2.67</td>
<td>2.78</td>
<td>2.88</td>
<td>2.97</td>
</tr>
<tr>
<td>a=100</td>
<td>1.50</td>
<td>1.83</td>
<td>2.08</td>
<td>2.28</td>
<td>2.45</td>
<td>2.59</td>
<td>2.71</td>
<td>2.82</td>
<td>2.92</td>
<td>3.01</td>
</tr>
<tr>
<td>Eqn(6b):</td>
<td>1.5</td>
<td>1.83</td>
<td>2.08</td>
<td>2.28</td>
<td>2.45</td>
<td>2.59</td>
<td>2.71</td>
<td>2.83</td>
<td>2.93</td>
<td>3.02</td>
</tr>
<tr>
<td>Eqn(16):</td>
<td>1.5</td>
<td>1.83</td>
<td>2.08</td>
<td>2.28</td>
<td>2.45</td>
<td>2.59</td>
<td>2.71</td>
<td>2.82</td>
<td>2.92</td>
<td>3.02</td>
</tr>
<tr>
<td>Eqn(17):</td>
<td>1.33</td>
<td>1.74</td>
<td>2.02</td>
<td>2.24</td>
<td>2.42</td>
<td>2.57</td>
<td>2.70</td>
<td>2.81</td>
<td>2.92</td>
<td>3.00</td>
</tr>
</tbody>
</table>

For a=1, Eqn(13) gives 1

Notation:  
N       order of FLM  
M       number of events occupied in memory  
T       number of traces in memory  
a       T divided by number of different tails of length N.

The results above are derived in (A.2.1). The equation numbers are also taken from there.
Table 2 Memory Usage in Simple - Alternating Patterns and Actions

\[
M/T = \begin{cases} 
2F((N-1)/2) & \text{N odd} \\
2F(N/2) - a^{\frac{-N/2-1}{a-1}}/(N/2+1) - 2^{\frac{-N/2}{a\cdot N}} & \text{N even}
\end{cases}
\]  

(\(F(L)\) is \(M/T\) using (13) in Table 1).

\[
M/T = \begin{cases} 
2F((N-1)/2) & \text{N odd} \\
2F(N/2) - (N/2 + 1) & \text{N even}
\end{cases}
\]  

(\(F(L)\) is \(M/T\) using (6b) or its approximations in Table 1)

Values of \(M/T\) using (29) and (6b):

<table>
<thead>
<tr>
<th>N</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>M/T</td>
<td>2</td>
<td>2.5</td>
<td>3</td>
<td>3.33</td>
<td>3.66</td>
<td>3.91</td>
<td>4.16</td>
<td>4.36</td>
<td>4.56</td>
<td>4.73</td>
</tr>
</tbody>
</table>

All the results above are derived in A.2.2.
Experimental Comparison

These analytic results will now be compared with the actual performance of Simple when its inputs were generated by the following stochastic automaton:

The automaton has two alternating states, in one of which an A or a B is emitted with probability 0.5, and in the other a B or C is emitted with probability 0.5.

A / B  
0.5 0.5

B / C  
0.5 0.5

A sequence of 1000 events generated by this process was input to Simple for a number of different memory sizes and with the order set to 2, 3 and 4, respectively. A count was then made of the total number of errors over the run, where an error is defined as occurring whenever the event which actually occurred was not in the prediction set. This number should decline with increasing memory size until it finally reaches a constant value when the memory is sufficiently large to contain all the memory traces.

Fig. 19 is a graph showing errors plotted against memory size for $N = 2, 3$ and 4. Table 3 compares the expected memory size from the analytical results with the experimental results from Fig. 19.

For the particular input sequence from the automaton, "a" has a value lying between 2 and 3, since a string of the form BB...BB can be followed by A, B or C, while any others by A and B, or by B and C. The number of different traces $T$ was calculated by examining the automaton and working out all possible sequences of length $N + 1$ which could occur. The experimental value of $M$ was estimated by eye from Fig. 19.
The analytic values were calculated from $T$ and Table 1.

In the case when $N = 4$, more than 1000 iterations are needed to obtain a good estimate for the required memory size. This explains why the experimental memory size was significantly smaller than predicted. Otherwise the fit between the two cases is good.
Table 3 Analytic vs Experimental Memory Size

<table>
<thead>
<tr>
<th>N</th>
<th>T</th>
<th>Experimental M</th>
<th>Analytic M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a=2</td>
<td>a=3</td>
<td>a&gt;&gt;1</td>
</tr>
<tr>
<td>2</td>
<td>19</td>
<td>30</td>
<td>28</td>
</tr>
<tr>
<td>3</td>
<td>35</td>
<td>60</td>
<td>57</td>
</tr>
<tr>
<td>4</td>
<td>67</td>
<td>&gt;120</td>
<td>120</td>
</tr>
</tbody>
</table>

Fig. 19 Total 'Errors' vs Memory Size
IV.8. COMPARISON OF FLM IMPLEMENTATIONS

In this section I will compare the execution times and storage requirements of the various FLM implementations. To make the comparisons I will draw on the expressions for memory size and execution times developed in earlier sections and summarized in Table 4. I will also draw on four examples of FLM memories generated using Felix. The 'vital statistics' of these memories are tabulated in Table 5 and section 8.4 explains how they were built up.

It will be seen that Simple uses significantly less storage than any of the other systems. However, its very long execution times on serial computers make it impractical in most circumstances. None of the other systems is spectacularly better than the others. However, a tree representation, with a binary event tree provides a mixture of consistently good storage usage and execution times. In some circumstances Mew-PUSS can provide marginally better storage usage than such a tree structure. However, this is offset by the possibility of it providing erroneous predictions.
Table 4. Execution Times and Storage Requirements for FLM Implementations

<table>
<thead>
<tr>
<th>System</th>
<th>Lower Bound</th>
<th>Exact Expression</th>
<th>Upper Bound</th>
<th>In Core</th>
<th>Disc Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree</td>
<td>(e+2p)(1+\text{L})</td>
<td></td>
<td></td>
<td>\frac{1}{N}L_N \sum_{i=1}^{N} a_i</td>
<td>1</td>
</tr>
<tr>
<td>Binary Tree</td>
<td>(e+3p)(1+\text{L})</td>
<td></td>
<td></td>
<td>\log_2(L_N)</td>
<td>1</td>
</tr>
<tr>
<td>Felix</td>
<td>4p(\text{L}+\frac{1}{\text{d}})</td>
<td>2p(\frac{1}{\text{d}} + \frac{1}{a} + 1)</td>
<td>4p(1+\frac{1}{\text{d}})</td>
<td>\log_2(a_1)+\frac{N}{2} \sum_{i=2}^{N} a_i</td>
<td>1</td>
</tr>
<tr>
<td>Hashing*</td>
<td>\frac{4}{3}(e+1)\left[1+\frac{N+3}{2} \cdot \frac{1}{\text{d}}\right] \frac{1}{3}(e+1)\left[1+\frac{1}{\text{d}} \sum_{i=1}^{N} \text{L}_i \cdot \frac{1}{i}\right]</td>
<td>\frac{4}{3}(e+1)\left[1+(N+1) \cdot \frac{1}{\text{d}}\right]</td>
<td>\frac{N}{2}(1+\frac{1}{(1-\text{d})^2})</td>
<td>8.5N</td>
<td>N</td>
</tr>
<tr>
<td>Mew-PUSS*</td>
<td>\frac{1}{3}(e+\text{t})(1+\frac{1}{\text{d}})</td>
<td>\frac{4}{3}(e+\text{t})(1+\frac{1}{\text{d}})</td>
<td>\frac{N}{2}(1+\frac{1}{(1-\text{d})^2})</td>
<td>8.5N</td>
<td>N</td>
</tr>
<tr>
<td>Simple**</td>
<td>see Tables 1 and 2</td>
<td></td>
<td></td>
<td>N</td>
<td>-</td>
</tr>
</tbody>
</table>

* assuming \(d \leq \frac{3}{4}\)  ** using parallel hardware
IV.8.1. Storage Usage

In this section I will compare the storage usage of the various FLM implementations. The comparison will be on the basis of the number of bits used for each trace stored in a memory.

Fig. 20 plots the memory usage of each of the FLM implementations; the memory usage is given in multiples of \( e \), the number of bits needed to store an event. I will now consider the plots for each implementation individually.

The right half of Fig. 20 plots the storage usage for Simple against \( N \). Two cases are shown, one when all inputs are actions and the other when actions and patterns alternate. The extreme limits for each of these cases are shown; a minimum when \( a = 1 \) and a maximum when \( a \) becomes very large.

The left part of Fig. 20 plots the memory usage for the other implementations against the value of \( L/T \). To obtain these plots it was assumed that \( p = e \) and, for the two hashing systems, that \( d = 3/4 \) (a maximum worst case).

The graph for Mew-PUSS depends on the size assigned to its tag. Two cases have been plotted, one where \( t = 3e \) which corresponds to the usage in (Andreea, 1977) and the other where \( t = e \) which is probably the minimum for effective operation, as the expected error rate increases when the tag length decreases. The two graphs straddle the results for the other systems; the \( t = 3e \) case uses more storage than any of the other systems, while the \( t = e \) case is marginally smaller than the others in almost all situations.

The graphs for the tree and binary tree FLMs depend only on \( L/T \) so they form two straightforward plots. The simple form of tree, while
never having the lowest usage, is nevertheless consistently near the minimum of all the systems. The binary tree, on the other hand, is consistently near the maximum storage usage of the other systems.

Two graphs have been drawn for Felix; one, when \( a = 1 \), is a maximum and the other, when \( a \) is very large, is a minimum (By coincidence, the plots for the binary tree, Felix (\( a = 1 \)) and Mew-PUSS (\( t = 2e \)) are identical.) Actual performance is probably nearer the upper of these two graphs as usually \( 1 \leq a \leq 1.5 \).

The storage for the variable key hash algorithm cannot be easily plotted because of its dependancy on the term \( \frac{\sum_{i=1}^{N} 1 \cdot L_i}{T} \), which bears little relationship to \( L/T \). However, for completeness, three graphs have been drawn. The bottom most of the three is a lower bound for \( N = 2 \), the middle graph is both an upper bound for \( N = 2 \) and a lower bound for \( N = 3 \), the uppermost graph is an upper bound for \( N = 3 \). The memory usage of this representation is economical only for \( N = 2 \) or less. As \( N \) increases, the memory usage climbs steeply because of the corresponding increase in the length of the hash key held with each node.

As well as the graphical representation of Fig. 20, Table 6 tabulates the number of bits per trace (again in multiples of \( e \)) using the example memories of Table 5. The data extracted from these memories is sufficient to enable an exact calculation to be made in each case. For convenience of comparison the results of Table 6 have also been plotted on Fig. 20.

Apart from Simple, the two implementations that consistently use the least storage are the simple non-binary tree and Mew-PUSS, where the tag is just one event in length.
Fig. 20 Comparison of FLM Memory Usage.
### Table 5 Memory Characteristics of Sample FLM Memories

<table>
<thead>
<tr>
<th>Case</th>
<th>Order</th>
<th>( a )</th>
<th>( L/T )</th>
<th>( i.L/T )</th>
<th>( a )</th>
<th>( a )</th>
<th>( a )</th>
<th>( a )</th>
<th>( a )</th>
<th>( a )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>1.24</td>
<td>1.72</td>
<td>5.5</td>
<td>65</td>
<td>3.8</td>
<td>1.9</td>
<td>1.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>1.25</td>
<td>2.06</td>
<td>9.9</td>
<td>9</td>
<td>3.1</td>
<td>2.3</td>
<td>1.9</td>
<td>1.6</td>
<td>1.4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>1.28</td>
<td>2.18</td>
<td>14.4</td>
<td>3</td>
<td>0.27</td>
<td>2.3</td>
<td>2.1</td>
<td>1.9</td>
<td>1.7</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>1.47</td>
<td>1.25</td>
<td>4.2</td>
<td>99</td>
<td>5.5</td>
<td>2.4</td>
<td>1.8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 6 Memory Usage for Sample FLM Memories

<table>
<thead>
<tr>
<th>Case</th>
<th>Tree</th>
<th>Binary</th>
<th>Felix</th>
<th>Hashing</th>
<th>( t=e )</th>
<th>( t=3e )</th>
<th>Simple</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8.2</td>
<td>10.9</td>
<td>10.5</td>
<td>11.0</td>
<td>7.3</td>
<td>14.5</td>
<td>1.36</td>
</tr>
<tr>
<td>2</td>
<td>9.2</td>
<td>12.2</td>
<td>11.8</td>
<td>17.4</td>
<td>8.2</td>
<td>16.3</td>
<td>1.51</td>
</tr>
<tr>
<td>3</td>
<td>9.5</td>
<td>12.7</td>
<td>12.3</td>
<td>23.4</td>
<td>8.5</td>
<td>17.7</td>
<td>1.70</td>
</tr>
<tr>
<td>4</td>
<td>6.8</td>
<td>9.0</td>
<td>8.4</td>
<td>8.6</td>
<td>6.0</td>
<td>12.0</td>
<td>1.56</td>
</tr>
</tbody>
</table>

### Table 7 Estimates of Execution Times

<table>
<thead>
<tr>
<th>Case</th>
<th>Tree</th>
<th>Binary</th>
<th>Felix</th>
<th>Hashing</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>36</td>
<td>9.4</td>
<td>9.6</td>
<td>34</td>
</tr>
<tr>
<td>2</td>
<td>9.6</td>
<td>8.1</td>
<td>8.3</td>
<td>51</td>
</tr>
<tr>
<td>3</td>
<td>8.2</td>
<td>7.9</td>
<td>8.3</td>
<td>68</td>
</tr>
<tr>
<td>4</td>
<td>54</td>
<td>11.2</td>
<td>11.5</td>
<td>34</td>
</tr>
</tbody>
</table>
IV.8.2. Execution Times

There are two distinct cases which arise when considering execution times. In one, the entire FLM memory is resident on central store and the execution time is dominated by the number of memory locations examined during a search. In the other case, the FLM memory is resident on a slow backing store, such as a disc, and the execution time is dominated by the time to read records onto central store.

IV.8.2.1. Central Store Resident

Table 7 lists estimates of the execution time for each system operating on each of the example memories of Table 5. These estimates have been calculated using the expressions of Table 1 which measure the expected number of memory locations examined during a search of the memory. In the case of the tree representations this is the number of nodes visited on the way up the tree. In the case of the hashing algorithms this is the number of probe locations whose keys are examined.

Too much significance should not be read into the higher estimates for the hashing algorithms in contrast to the three tree algorithms. The underlying steps in the hash and tree searches are very different so that the time to examine a single location in a hash probe might be very different from the time to traverse a single node in a tree search. Also, the estimates ignore many constant overheads incurred by both algorithms such as setting up buffers and calculating hash keys so that in practice the actual execution times should be rather closer than these estimates indicated.

The estimates for the three tree algorithms should be comparable
because of their underlying similarity. The simplest tree algorithm suffers rather badly when large numbers of events force a long search at the first level in the tree. However, the compromise in Felix, where a binary event tree is used and the rest of the tree is in the simpler form, successfully removes this sharp dependence on the number of events. Indeed, the estimates for Felix are so close to those for the full binary tree as to be effectively the same.

IV.8.2.2. Disc Resident

The situation when the FLM memory is resident on disc is straightforward. The tree algorithms require 1 disc access for each update and search (this presumes the organisation of section 2.7 is used). The hashing implementations require \( \text{N} \) disc reads, one for each level in the tree. However, this number can be reduced by not storing all the levels in the tree. For example in (Andreae, 1977) only the upper level is stored so no best matching is done and only one disc access is needed.

IV.8.3. Conclusion

The tree FLM provides a good design on which to base a practical implementation of an FLM memory. It provides a good balance of storage economy and, if combined with a binary tree of events, low execution times.

Provided the error rates implied by a short tag field are tolerable, then an incore Mew-PUSS would provide some small memory savings. For example, the error rate when \( t = e = 16 \) is given by equation (1) of 6.2 as 1/20,000. However, when using a disc based
memory Mew-PUSS makes N disc reads against one for a tree FLM.

Using current (1979) technology, mini and micro computers with 16 bit words and memory capacities of up to 64 thousand bytes are readily and cheaply available. On such a system, 48 thousand bytes might be available for use as an FLM memory (the balance of 16 thousand bytes being taken up by programs). Letting \( p = e = 16 \) bits this memory would be able to store about 2,400 traces and execute between 20 and 50 FLM computer cycles per second, depending on the speed of the CPU.

If a disc based memory were used with such a mini-computer then the execution time would be dominated by the disc access times; 70 milliseconds is a typical time to fetch a record at random from such a disc which might contain about 10 million bytes. This system would be able to store about 500,000 traces and execute 10 FLM computer cycles a second.

IV.8.4. Example Memories

The inputs for the example memories of Table 5 were generated by the Blindman environment (Heads, 1975 a, b). In this environment a 'blindman' is walking along a path parallel to a row of poles, as he walks he receives the following patterns:

- the distance to the next pole, (0 to 3);
- the angle between the direction he is facing and the next pole, (0 to 7);
- whether he is on the path or off it (N (near the poles), F (far from the poles), * (on the path), / (off the edge of the world)).

The blindman has three possible actions available to him, in
each of which he takes one step forward. The actions are: C (centre), L (left), R (right). So a typical action and following pattern might be L 60 * or C 31/.

Each of the memories was generated by inputs from this environment to Felix. In each memory an action and its succeeding pattern were packed together into a single event. However, the actual patterns received were varied from memory to memory. The examples 1, 2 and 3 were generated by the same input sequence of 2000 steps. Their input patterns were built up as follows:

example 1, all the information described above, e.g. 60 *;
example 2, only the path information (no distance or angle information), e.g. * or F;
example 3, no pattern information at all only actions were included in the events

Example 4 was generated from a sequence of 15000 steps and received all the information in its patterns.

The data in Table 5 was extracted by a program which counted the number of nodes at each level in the tree.
IV.9. NOTATION

The following notations have been used consistently throughout the chapter and are included here for reference:

N  
Order of FLM.

T  
Number of traces in an FLM memory.

Tᵢ  0 ≤ i ≤ N  
Number of traces of order i.

Lᵢ  0 ≤ i ≤ N  
Number of distinct tails (of traces) of order i.

L  
Total number of tails (\(= \sum_{i=1}^{N} Lᵢ\)).

a  
Number of events predicted by each trace (\(= T/L\)).

aᵢ  1 ≤ i ≤ N  
Number of branches at level i in a tree representation (\(= Lᵢ/Lᵢ₋₁\)).

M  
Number of memory locations available.

b  
Number of bits per trace.

e  
Number of bits required to store an event.

p  
Number of bits to store a pointer into memory (\(≥ \lceil \log_2(M) \rceil \)).

t  
Number of bits to store a hash 'tag'.

p  
Fraction of memory which has been used.
IV.A.1. Forgetting Results

This section derives the results which were used to provide the graphs of Figs. 1 and 2 comparing different forgetting algorithms.

IV.A.1.1. Two Population Model

IV.A.1.1.1. Ladder

Given a ladder memory with \( k \) traces stored on it, let \( Q_k \) be the number of members of \( Q \) expected to be on the ladder, and \( R_k \) the number of members of \( R \) expected to be on the ladder. From these definitions and the assumptions in 1.2, we have the results

\[
Q_k + R = k. \quad (1)
\]

\[
E = qQ_k + rR_k. \quad (2)
\]

\[
1 = q|Q| + r|R|. \quad (3)
\]

Also, let \( b = q|Q| \) and \( c = r|R| \). These quantities are convenient in the ensuing analysis but, along with \( k \), \( Q_k \) and \( R_k \) will be eliminated from the final result.

Consider some sequence of traces selected from \( P \). In the diagram below each trace is assigned a cell and those traces which have not occurred again since are shaded.

![Diagram of ladder memory with shaded cells indicating traces selected from P.](image-url)
Now $Q_k$ can be found by counting the number of shaded cells containing members of $Q$ which occur between the $k$'th shaded cell and the first shaded cell (that is the one which occurred most recently).

Let $Z_k$ be the probability that the $(k + 1)$'th shaded cell will contain a member of $Q$. Consider the cell (shaded or unshaded) immediately preceding the $k$'th shaded cell: the probability that it is from $Q$ is $b$ and from $R$ is $c$. The probability that it will be shaded if it is from $Q$ is $1 - Q/|Q|$, that is the probability that it has not occurred again. Similarly, if it is from $R$, then the probability that it is not shaded is $R/|R|$. $Z_k$ then satisfies

$$Z_k = b(1 - Q/|Q| + QZ_k/|Q|) + cR/|R|Z_k.$$

Now $Q_k = Q_k + Z_k$ or, to use a differential approximation,

$$dQ_k = Z_k dk.$$

Let $Y = 1 - Q/|Q|$, then $dY = (-1/|Q|)dQ_k$;

$$X = 1 + |R|/|Q| - k/|Q|, \quad dx = -k/|Q| dk;$$

$$f = |Q|/|R|.$$ 

Substituting in (4) and (5) and rearranging gives

$$dY = bY.dX/[Y(bf + b - f) + X(1 - b)f].$$

Most of the remainder of this section involves searching for a solution to the differential equation in (7). This equation is an homogeneous differential equation and can be solved by standard techniques.

Let $Y = SX$, then

$$dY = X.dS + S.dX.$$ 

Substituting (8) into (7) and rearranging gives

$$(X.dS + S.dX)(SX(bf + b - f) + f(1 - b)X) = bSX.dX.$$ 

Divide (9a) by $X(b - f(1 - b))$ and one obtains
\[ dS\left(S + f(1-b)/(b-f(1-b))\right)/(S(S-1)) = -dX/X. \]  
\[ (9b) \]

Let \( u = f(1-b)/(b-f(1-b)) = \|Q\|/\|R\| \cdot \|R\|/(\|Q\| - \|Q\|/\|R\| \cdot \|R\|) \)
\[ = r/(q-r) = 1/(a-1) \]

which simplifies (9b) to
\[ dS(S + r/(q-r))/(S(S-1)) = -dX/X, \]
\[ (9c) \]
which is independent of \( b \) and \( c \).

Integrating the LHS of (9c) gives,
\[ \int (S+u)/S(S-1) \, dS = \int -u/S \, dS + \int (1+u)/(S-1) \, dS. \]
\[ (10) \]
Because \( Y = 1 - Q/\|Q\| \) and \( 0 \leq Q \leq \|Q\|, \)
\[ k \]
then \( 0 \leq Y \leq 1. \)
\[ (11) \]
Also, \( X = 1 + |R|/\|Q\| - k/\|Q\| \) and \( 0 \leq k \leq |P| = |Q| + |R| \)
so \( 0 \leq X. \)
\[ (12) \]
From (8), (11), (12), \( 0 \leq S \)
so \( \int -u/SdS = -u\ln S + C'. \)
\[ (13) \]
The sign of \( S-1 \) must also be determined.

Now \( S-1 = (Y/X)-1, \) so \( S-1 \leq 0 \)
\[ \text{iff } Y/X - 1 \leq 0 \]
\[ \text{iff } Y - X \leq 0 \quad \text{(from (12))} \]
\[ \text{iff } 1-Q/\|Q\| - |R|/\|Q\| + k/\|Q\| \leq 0 \]
\[ k \]
\[ \text{iff } k - Q - |R| \leq 0. \]

Also, \( dQ = Z \, dk \) and \( Z \leq 1, \) so \( dQ \leq dk, \)
\[ k \]
that is, \( k - Q_k \) will be monotonic increasing w.r.t. \( k. \)

When \( k = |Q| + |R|, \) then \( Q = |Q|, \) and \( k - Q - |R| = 0 \)
but \( k \leq |Q| + |R|, \) so from monotonicity \( k - Q - |R| \leq 0, \)
and further
\[ S - 1 \leq 0, \] so
\[ (1+u)/(S-1) \cdot dS = (1+u)\ln(1-S) + C''. \]
\[ (14) \]
From (9c), (10), (13) and (14)

\[-u \ln S + (1+u) \ln (1-S) = -X \ln X + C\]  \hspace{1cm} (15)

Now determine C.

When \( k = |Q| + |R| \), then \( Q = |Q| \), then it follows that \( k \)

\[X = 1 + |R|/|Q| = 1 + 1/f, \quad Y = 1 \text{ and } S = f/(1+f)\].

Substituting in (15) and rearranging gives,

\[C = -(1+u) \ln f\]  \hspace{1cm} (16)

Substituting this back into (15) and exponentiating gives

\[u \quad -1-u\]

\[X = S (f(1-S))^{-u}\]  \hspace{1cm} (17)

Let \( T = S(f+1)/f \), then from (8) and (17)

\[1+u\]

\[Y = (T/(1+f-fT))^{1+u}\]  \hspace{1cm} (18)

By definition,

\[v = k/(|Q|+|R|)\]  \hspace{1cm} (19a)

Divide this expression by \(|Q| \) and use (6), then

\[v = (1 + 1/f - X)/(1 + 1/f)\].

From (6) and (8) this yields in turn

\[v = 1 - Y/T\]  \hspace{1cm} (19b)

From (1) and (2), (6) and (19a),

\[E = (q-r)Q + rk = (q-r)(1-Y)|Q| + v(|Q|+|R|)r\]

\[= (1-Y)((a-1)f)/(af+1) + v(f+1)/(af+1)\]  \hspace{1cm} (20)

where \( a = q/r \).

Consider \( T \); from the derivation of (13) and (14)

\[0 \leq S \leq 1\]

so \[0 \leq T \leq 1 + 1/f\].  \hspace{1cm} (21)

Summary
E and v are related parametrically by T as follows:

\[
\frac{a}{a-1} \\
Y = \frac{T}{(1+f)-Tf} \\
v = 1 - \frac{Y}{T} \\
E = (1-Y) \frac{(a-1)f}{(af+1)} + v \frac{(f+1)}{(af+1)}
\]

where \(0 \leq T \leq 1 + 1/f\).

These relations were used to produce the graphs of Figs 1 and 2.

IV.A.1.1.2. Random Forgetting

Take a memory with \(k\) traces in it and assume that its population is in equilibrium with the inputs. Then the rate at which members of \(Q\) are being forgotten will equal the rate at which they are being inserted. The rate of loss will be just the probability that some entry is lost \((1-E)\) multiplied by the probability that the chosen trace is in \(Q\) i.e \((Q_k/k)\). The rate of insertion of members of \(Q\) is the probability that the trace which occurs is not in the memory \((1-E)\), multiplied by the probability that a trace not in the memory is in \(Q\), i.e.

\[
\frac{\text{[}q|Q| - Q\text{]}_k}{\text{[}q|Q| - Q\text{]}_k + r\frac{(|R|+R)}{k}}.
\]

Therefore,

\[
(1-E)Q_k = (1-E)\frac{q|Q| - Q}{q|Q| - Q} + r\frac{(|R|+R)}{k}.
\] \hspace{1cm} (22)

Cancelling and using (3) gives

\[
Q_k = q\frac{|Q|-Q}{(1-E)}.
\] \hspace{1cm} (23)

Now, from (2),

\[
E = qQ_k + rR_k, \text{ so } Q_k = \frac{(E-rk)}{(q-r)}.
\] \hspace{1cm} (24)

Substituting for \(Q_k\) in (23) and rearranging gives 236
\[ E = \frac{1}{2} E(1+r^2k+q) + rk + \frac{q^2}{2}(q-r)|Q|k = 0. \]  
(25)

Let \[ B = 1+r^2k \]
\[ = 1 + v(q+r)(|Q|+|R|)/(q|Q|+r|R|) \]
\[ = 1 + v(a+1)(f+1)/(af+1) \]  
(26)

Let \[ C = rk + qk + q(q-r)|Q| \]
Then using (19a)
\[ C = (r + q(q-r)|Q|(|Q| + |R|)/(q|Q| + r|R|) + rqv(|Q| + |R|) \]
Dividing by \( r \) and \( |R| \) gives
\[ C = v(f+1)/(af+1).[1 + (a(a-1)f)/(af+1) + v(a(f+1))/(af+1)] \]  
(27)

Solving for \( E \) in (25) gives
\[ E = \frac{1}{2} \sqrt{B - 4C} \]  
(28)

When \( v = 1 \) then \( E = 1 \) (memory holds all traces), but
\[ B = 1 + (a+1)(f+1)/(af+1) \]
so \( B = 2 + (f+a)/(af+1) \); but \( a > 1 \) and \( f \geq 0 \), so \( B > 2 \). Therefore the smaller of the roots in (28) must be used.

Summary

Given \( v \) and for
\[ B = 1 + v(a+1)(f+1)/(af+1) \]  
(26)
and \[ C = v(f+1)/(af+1)[1 + (a(a-1)f)/(af+1) + v(a(f+1))/(af+1)] \]  
(27)
then
\[ E = \frac{1}{2} \sqrt{B - 4C} \]  
(29)

IV.A.1.1.3. Best Case

\( E \) will be greatest for a given memory if the memory is full of traces from \( Q \) \( (q \geq r) \). That is,
In terms of $E$ and $v$ the graph will be two straight line segments joining $(0,0)$ and $(1,1)$ with $(f/(f+1), af/(af+1))$. 

IV.A.1.2. **Cyclic Model**

IV.A.1.2.1. Random Forgetting

Given a cycle of $N$ traces, then the probability that a given trace will be forgotten as each input occurs is $(1-E)/k$, where $k$ is the number of traces in the memory. $E$ is just the probability that a trace will survive a full cycle of $N$ inputs, so

\[ E = (1 - (1-E)/k)^N \sim e^{-N(E-1)/k} \quad \text{for } k \gg 1 \]

but $v = k/N$

which can be substituted above to give $v = (E-1)/\ln E$.

IV.A.1.2.2. Best Case

The best case will be achieved by holding a fixed set (of size $k$) of traces so that

\[ E = k/N = v \].

238
IV.A.2. Memory Usage in Simple

IV.A.2.1. All Action Inputs

An approximation will now be obtained for the memory occupied by a set of traces in simple when it is assumed that they have been generated by an input sequence of actions. This will be done by estimating gaps between successive active events in the memory, and then summing these values.

Notation:

\( N \) order of memory

\( M \) storage occupied (in number of events)

\( T \) number of different traces present in memory

\( a \) number of different symbols which can follow any given string of length \( N \)

\( n \) position of an active symbol in memory, i.e. \( n=1 \) for the symbol at Top; \( n=T \) for the last active symbol in memory.

This analysis assumes that the frequency with which the different traces occur is relatively uniform and that almost all the traces which can occur (because of the nature of the input) are actually represented in the memory. It is uncertain what effect violation of the first assumption will have, although rough checks indicate that it will tend to increase the needed memory. However, if the population from which the traces are being drawn is greater than that present, (that is, the possible number of traces is greater than \( T \)), then the memory space required can be much reduced. In the extreme case where \( T \) is a very small fraction of the total number of traces, no trace would ever have been repeated in the time for an event to be
introduced at the start of the memory and to be lost from the bottom. Then each trace requires the space of only one event, as every event in the memory will be active.

First, an expression for $M$ depending only on $T$ and $N$ is obtained. Later, a more exact result depending on $T$, $N$ and $a$ is derived.

The probability that some event lying between the $n$th and $(n+1)$th active event will be active is approximately $n/T$. (1)

This is just the probability that the trace of which the symbol is a head will have occurred later among the $n$ distinct traces there. If we let $P_k$ be the probability that there will be a gap of length $k$ between the $n$th and $(n+1)$th event, then

$$P_k = \frac{n}{T} \left(1 - \frac{n}{T}\right)^{k-1} \quad 0 \leq k < N.$$ (2a)

Any gap of length greater than $N$ will be closed up, so

$$P_k = \frac{n}{T} \quad \text{and} \quad P_k = 0 \quad k > N.$$ (2b)

The expected gap between the $n$th and $(n+1)$th active symbols (including the $n$th active symbol) will be denoted by $L(n)$ and can be estimated by the formula

$$L(n) = 1 + \sum_{k=1}^{N} kP_k \quad \text{(3a)} = 1 + \sum_{k=1}^{N-1} k\left(\frac{n}{T}\right)\left(1 - \frac{n}{T}\right)^{k-1} + N\left(\frac{n}{T}\right).$$ (3b)

The lengths of the different gaps will be approximately independent, so the total length of the memory can be obtained by summing all the values of $L(n)$:

$$M = \sum_{n=1}^{T} L(n).$$ (4)

(4) and (3b) together give
\[ M = \sum_{n=1}^{T} \left[ 1 + \sum_{k=1}^{N-1} \frac{k}{k(n/T) \left(1-n/T\right) + N(n/T)} \right]^N \]  \hspace{1cm} (5)

A closed expression for (5) is now found by approximating the sum with an integral:

\[ M \sim T \int_0^T \left[ 1 + \sum_{k=1}^{N-1} \frac{k}{k(n/T) \left(1-n/T\right) + N(n/T)} \right] \, dn. \]

Let \( x = n/T \), then

\[ T \, dx = dn \quad \text{and} \quad M = T \int_0^1 dx \sum_{k=1}^{N-1} k \int_0^x (1-x)dx + N \int_0^x dx. \]  \hspace{1cm} (6a)

The first result sought is found by evaluating the integrals in the expression (6a) giving

\[ M = T \left[ 1 + \sum_{k=1}^{N-1} \frac{k}{(k+1)(k+2) + N/(N+1)} \right]. \]  \hspace{1cm} (6b)

The analysis will now be repeated but this time account will be taken of the fact that when an event is inactive then the event following it has a higher probability of being inactive. The expression obtained for \( M \) by including this effect will depend on \( a \).

For a symbol between the \( n \)th and \( (n+1) \)th active symbols and preceded by an inactive symbol, the probability of it being inactive is

\[ \frac{n}{T} + \frac{1}{a} - \frac{n}{aT}. \]  \hspace{1cm} (7)

This is obtained by the following reasoning. The probability that the trace of which the symbol (say, Sym(i)) is a head will have occurred later is \( n/T \). However, the trace corresponding to the preceding symbol (Sym(i-1)) will also have occurred later (it is inactive) and the probability that the symbol following this later occurrence will be equal to Sym(i) is \( 1/a \) (see below). The probability of either of these events happening is just the expression in (7).
Situation in Memory for Derivation of Equation (7)

Let \( w = 1 - 1/a \) then the expressions for \( P_k \) are:

\[
P_k = \begin{cases} 
(1 - wn/T), & 0 \\
w(n/T)(1 - n/T - 1/a + n/aT), & 1 \\
b^{k-1} & 1 \leq k \leq N, \\
w(n/T)(n/T + 1/a - n/aT) (1 - n/T - 1/a + n/aT), & k \\
w(n/T)(n/T + 1/a - n/aT), & N \\
0, & k > N.
\end{cases}
\]

Equations (3a), (4) and (5) when applied to (8) give

\[
M = \sum_{n=1}^{T} \left[ \sum_{k=1}^{N-1} k(n/T)(n/T + 1/a - n/aT) (1-n/T-1/a+n/aT) \\
+ N(n/T)(n/T + 1/a - n/aT) \right].
\]

Let \( x = n/T \) and transform (9) into an integral:

\[
M \sim T \int_{0}^{x} \left[ \sum_{k=1}^{N-1} kx(x+1/a-x/a) (1-x-1/a+x/a) \\
+ Nx (x+1/a-x/a) \right] dx.
\]
Recall that \( w = 1-1/a \), then (10) can be rearranged to yield,

\[
M = T \left[ 1 + \sum_{k=1}^{N-1} \frac{1}{k} \int_0^1 x(1-x)(w^k+1/a) \ dx + N \int_0^1 x(w^k+1/a) \ dx \right]. \tag{11}
\]

Now substitute \( z = wx + 1/a \), \( dz = w \, dx \) to eliminate \( x \) from (11). After a little manipulation one obtains,

\[
M = T \left[ \frac{N-1}{1+w} \sum_{k=1}^{N-1} \frac{1}{k+1/a} \int_{1/a}^1 (z-1/a)(1-z)z \ dz + w \int_{1/a}^1 (z-(1/a)z) \ dz \right] \tag{12}
\]

Further manipulation after integration yields the second result sought;

\[
M = T \left[ 1 + \frac{N-1}{1+w} \sum_{k=1}^{N-1} \frac{k}{(k+1)[(1-a)^{-k} + (1-a)^{-k}/ak(1-a)]} \right.
\]

\[
\left. + N(1-a)^{-N}/(N+1) - (1-a)^{-N}/a \right]. \tag{13}
\]

As a check, assume \( a > 1 \), put \( w \sim 1 \) and \( 1/a \sim 0 \), and recover the first result (6b). Thus the earlier result is the special case when \( a \) is large.

A further approximation for \( M \) can be obtained from (6b) by finding an analytic expression for

\[
S = \sum_{k=1}^{N-1} \frac{k}{(k+1)(k+2)} \int_{1/2}^2 \frac{k}{(k+1)(k+2)} \, dk
\]

\[
= \ln(N+3/2)/(N+1/2) + \ln(6)/25, \quad (14)
\]

\[
- \ln(N+5/2) + \ln(6)/25. \quad (15)
\]

Substituting (14) and (15) into (6b) gives us two further approximations for \( M \); equations (16) and (17).

\[
M = T \left[ \ln(N+3/2)/(N+1/2) + N/(N+1) - 0.427 \right]. \tag{16}
\]

\[
M = T \left[ \ln(N+5/2) + N/(N+1) - 0.427 \right]. \tag{17}
\]

243
IV.A.2.2. Alternating Actions and Patterns

The analysis just given assumed that all the inputs were actions. The analysis will now be repeated for the case when actions and patterns alternate. It will be found that in this case about twice as many events are needed as for the purely action input case. As for equation (2a), we can define $P'$ as the probability of a gap of length $k$ occurring between the $n$th and $(n+1)$th active events. Now

$$P' = 0 \text{ (as at most every second symbol can be active), also}$$

$$0$$

$$P' = (1-(n/T)),$$

$$1$$

$$P' = 0$$

$$2$$

$$P' = (n/T)(1-(n/t)),$$

$$3$$

and, in general,

$$P' = 0 \text{ and}$$

$$2j$$

$$P' = \frac{1}{2j+1} \sum_{j=0}^{N/2} (n/T)(1-(n/T)) \text{ also,}$$

$$N/2$$

if $N$ is even, \( P' = \frac{1}{N} \) and

$$N$$

if $N$ is odd, \( P' = \frac{(N-1)/2}{N} \).

(18)

IV.A.2.2.1. N Odd

Consider the case where $N$ is odd; then,

$$P' = 0$$

$$2j$$
\[ P'_k = \frac{(n/T)(1-(n/T))}{2j+1}, \text{when } 0 \leq j \leq (N-1)/2, \]
and \[ P'_k = 0 \quad \text{for } k > N. \]  
(19)

Relating this to \( P'_j \) given in equation (2a) gives
\[ P'_j = P_{2j+1}. \]  
(20)

If \( L'(n) \) is the expected gap between the \( nth \) and \( (n+1)th \) active symbols in the action pattern case, then
\[ L'(n) = 1 + \sum_{k=0}^{N} kP'_k. \]  
(21)

Using (19), (20), and (21) one obtains
\[ L'(n) = 1 + \frac{(N-1)/2}{\sum_{j=0}^{(N-1)/2} (2j+1)P_j} = 2(1 + \sum_{j=0}^{(N-1)/2} jP_j). \]  
(22)

But, \( L(n) \) in equation (3a), when applied to a memory of order \( (N-1)/2 \), is just half the value of \( L'(n) \) given by equation (22), that is
\[ L'(n) \text{(order } N) = 2L(n) \text{(order } (N-1)/2). \]  
(23)

Let \( M' \) be the memory required in the action pattern case; then, from equation (23) and equation (4),
\[ M' = T 2F((N-1)/2) \quad (N \text{ odd}), \]  
(24)

where \( F(N) \) is \( M/T \) for the all action case.

IV.A.2.2.2. N Even

Consider now the case when \( N \) is even:
\[ P'_j = 0, \quad 2j \neq N, \quad 2j \]
\[ P'_j = P, \quad 2j+1 < N, \quad 2j+1 \]
\[ P'_N = P, \quad N = N/2. \]  
(25)
Now \( L'(n) = 1 + \sum_{k=0}^{N} k P' = 1 + \sum_{j=0}^{N/2-1} (2j+1) P + 2(N/2) P \n/2 \)

\[ = 1 + 2 \sum_{j=0}^{N/2} j P + \sum_{j=0}^{(N/2)-1} P \n/2 \]

\[ = 2 \left( 1 + \sum_{j=0}^{N/2} j P \right) - P \n/2 \]

\[ = 2 L(n) \text{ (Order} = N/2) - (n/T) \]

(26a)

(26b)

Corresponding to equation (24), we obtain

\[ M' = T \cdot 2 F(N/2) - \sum_{n=1}^{T} (n/T) \n/2 \]

(27)

To simplify the summation in (27), let \( x = n/T \) and approximate by an integral:

\[ \sum_{n=1}^{T} (n/T) \n/2 - T \int_{0}^{1} x \n/2 \ dx = T/(N/2+1) \]

Substituting this back into equation (27) gives

\[ M' = T[2 F(N/2) - 1/(N/2+1)] \text{ (N even).} \]

(28)

Combining equations (24) and (28) one obtains in summary,

\[ M'/T = \begin{cases} 
2 F ((N-1)/2) & \text{N odd} \\
2 F (N/2) - 1/(N/2 + 1) & \text{N even}
\end{cases} \]

(29)

This result is true when \( a \) is large, however taking into account a finite value for \( a \) and using reasoning similar to that above leads to the same results as (23) and (26a), that is:

\[ L'(n) \text{ (order} N) = \begin{cases} 
2(1 + \sum_{j=0}^{(N-1)/2} j P) & \text{N odd} \\
2 \left( 1 + \sum_{j=0}^{N/2} j P \right) - P & \text{N even}
\end{cases} \n/2 \]

(30)

For \( N \) odd this implies

246
\[ M' = T.2 F ((N-1)/2) \quad . \quad (31) \]

For \( N \) even and using \( P_{N/2} \) from (8), (30) implies

\[ M' = T.2 F(N/2) - \sum_{n=1}^{T} \int (n/T + 1/a - n/aT)^{(N/2)-1} \cdot \] \( \int \frac{w(n/T)}{(n/T + 1/a - n/aT)} \cdot \]

Making the substitution \( x = (n/t) \) and replacing the summation by an integral the last term of (32) becomes:

\[ \int \frac{w}{T} \left[ x \left( x + 1/a - x/a \right)^{(N/2)-1} \right] dx \]

\[ = \frac{T}{w} \left[ \left( \frac{-N/2-1}{(1-a)/(N/2+1) - 2(1-a)/aN} \right) \right] . \quad (33) \]

From (31), (32) and (33) the following general result is obtained:

\[ M'/T = \begin{cases} 
2F((N-1)/2) & \text{N odd} \\
2F(N/2)-1/w(1-a)/(N/2+1)-2(1-a)/aN & \text{N even.} \end{cases} \quad (34) \]
This is a listing of a version of Felix written in Burroughs Extended Algol 60 for the B6700 (Burroughs Corp., 1974).

IV.A.3.1. Memory Structure as Defined by Macros

The following macro defines specify the mapping of the memory structure of Felix onto an array of integers and of the individual fields at each location onto a 48 bit word. All memory manipulation by later routines is done in terms of these macros.

Each macro has the syntax
'DEFINE' <name> '(', '<1 or more macro parameters>', ')', '=' <text of macro> '#'. A number of definitions may occur after a single DEFINE.

COMMENT
SETUP RESERVED WORD STRUCTURE FOR B6700;
DEFINE
COMMENT
DEFINITIONS OF BIT FIELDS FOR USE IN B6700 NODE STRUCTURE;
DEFINE
FIELD1=[15:16]#, FIELD2=[31:16]#, FIELD3=[47:16]#
MAXPOINTER = REAL(^BOOLEAN(0)).FIELD1#
BIT1 = [37:1]#
BIT2 = [38:1]#
BIT3 = [32:1]#
BIT4 = [33:1]#
FREQ = [36:31]#
% THESE DEFINES ARE USED IN ANSWER ARRAYS AS WELL AS THE MEMORY
RPF = [38:2]#
RPF2 = [4:2]#
TYPE = [2:3]#
MAXFREQ = REAL(^BOOLEAN(0)).FREQF#

COMMENT
READ AND SET ROUTINES FOR NODE STRUCTURE;
DEFINE
NEXT(M, I) = CHECK(I, M, M[I].FIELD1)#
SETNEXT(M, I, X) = SET(I, M, (M[I] = # & X).FIELD1)#
VAL(M, I) = CHECK(I, M, M[I].FIELD2)#
SETVAL(M, I, X) = SET(I, M, (M[I] = # & X).FIELD2)#
REWARD(M, I) = CHECK(I, M, M[I].BIT1)#
SETREWARD(M, I, X) = SET(I, M, (M[I] = # & X).BIT1)#
PAIN(M, I) = CHECK(I, M, M[I].BIT2)#
SETPAIN(M, I, X) = SET(I, M, (M[I] = # & X).BIT2)#
RP(M, I) = CHECK(I, M, (M[I].RPF))#
SETRP(M, I, X) = SET(I, M, (M[I] = # & X).RPF) RP)#
EOG(M, I) = CHECK(I, M, M[I].BIT3)#
SETEOG(M, I, X) = SET(I, M, (M[I] = # & X).BIT3) BIT3)#
EOA(M, I) = CHECK(I, M, M[I].BIT4)#
SETEOA(M, I, X) = SET(I, M, (M[I] = # & X).BIT4) BIT4)#
TOUCH(M, I) = CHECK(I, M, M[I].BIT4)#
SETTOUCH(M, I, X) = SET(I, M, (M[I] = # & X).BIT4) BIT4)#
FREQ(M, I) = CHECK(I, M, M[I].FREQF)#
SETFREQ(M, I, X) = SET(I, M, (M[I] = # & X).FREQF) FREQF)#
CHAR(M, I) = CHECK(I, M, M[I])#
SETCHAR(M, I, X) = SET(I, M, M[I] = X)#
ZERONODE(M, I) = SET(I, M, M[I] = 0) #
IV.A.3.2. Memory Access Routines

These routines perform some common access sequences down chains of pointers. For example ACTION finds the value of the action node (in the event tree) given a down node in the leaves of the tree.

INTEGER PROCEDURE ACTION(MEM,S);
VALUE S;INTEGER S;REAL ARRAY MEM[0; ]
COMMENT
S POINTS TO A NODE WHOSE VAL FIELD POINTS TO AN E(PAT) NODE
S-->(DOWN,S,ST)-VAL->E(PAT)-NX->C-NX->BACK-VAL->E(ACT);
ACTION:=
   VAL(MEM,NEXT(MEM,NEXT(MEM,VAL(MEM,S))));

INTEGER PROCEDURE BRANCH(MEM,K);
VALUE K;INTEGER K;REAL ARRAY MEM[0; ]
BRANCH:=
   IF K<1 OR K>N(MEM) THEN ERROR(MEM,3,1) ELSE
      IF K=1 THEN CURRP(MEM,1) ELSE
         IF CURRP(MEM,K)=0 THEN 0 ELSE VAL(MEM,CURRP(MEM,K));

REAL PROCEDURE APIC(MEM,S);
VALUE S;INTEGER S;REAL ARRAY MEM[0; ]
COMMENT
   S-->(DOWN,S,S-T)-ACTION->E(ACT)-NX->C-VAL->CHAR;
APIC:=
   PIC(CHAR(MEM,VAL(MEM,NEXT(MEM,ACTION(MEM,S))));

REAL PROCEDURE PPIC(MEM,S);
VALUE S;INTEGER S;REAL ARRAY MEM[0; ]
COMMENT
   S-->(DOWN,S,S-T)-VAL->E(PAT)-NX->C-VAL->CHAR;
PPIC:=
   PIC(CHAR(MEM,VAL(MEM,NEXT(MEM,VAL(MEM,S))));
IV.A.3.3. **Setfree**

PROCEDURE SETFREE(MEM,I);
VALUE I;INTEGER I;REAL ARRAY MEM[0];
BEGIN
COMMENT
    RETURNS NODE POINTED TO BY I TO THE FREELIST;
    SETNEXT(MEM,I,FREE(MEM));FREE(MEM):=I;
    FREENUMBER(MEM):=FREENUMBER(MEM)+1;
END OF SETFREE;

IV.A.3.4. **Get**

INTEGER PROCEDURE GET(MEM);
REAL ARRAY MEM[0];
BEGIN INTEGER I;
COMMENT
    DELINKS A NODE FROM THE FREE LIST AND RETURNS A POINTER TO IT
    THE NODE IS ZEROED;
    ENSURE(MEM,1);
    I:=GET:=FREE(MEM);FREE(MEM):=NEXT(MEM,I);
    FREENUMBER(MEM):=FREENUMBER(MEM)-1;
    ZERONODE(MEM,I);
END OF GET;
PROCEDURE ENSURE(MEM,F);
VALUE F;INTEGER F;REAL ARRAY MEM[0];
BEGIN
COMMENT
  KEEPS FORGETTING MEMORY ELEMENTS UNTIL AT LEAST F MEMBERS ON FREE LIST
  PROCEDURE FORGET DOES ALL THE WORK;
  PROCEDURE FORGET;
  COMMENT
  REMOVES AT LEAST ONE NODE FROM MEMORY AND RETURNS TO FREE LIST
  THERE ARE DIFFICULTIES WITH SMALL MEMORIES WHEN REPEATED ATTEMPTS
  MAY BE MADE TO FORGET THE CURRENT STRING.THIS IS IGNORED
  AND FORGET CALLED AGAIN BY ENSURE UNTIL SOMETHING IS FORGOTTEN
  THIS WILL POSE NO PROBLEM WITH LARGE MEMORIES;
BEGIN
  INTEGER ARRAY RBUF[2:N(MEM)];
  COMMENT
    RBUF HOLDS SYMBOL POINTERS USED FOR LOCATING ALL TRANSITIONS
    POINTING TO THE STRING TO BE DELETED(USED BY FINDPREVIOUS IN
    CHECKTRANS);
  PROCEDURE RANDOMTERMINAL(S);
  VALUE S;INTEGER S;
  COMMENT
    S POINTS TO AN S-T NODE
    IF THERE ARE NO TRANSITIONS FROM OR TO (CHECKED BY CHECKTRANS)
    THE STRING THEN THE DOWN NODE IS DELETED
    A CHECK IS MADE TO ENSURE THAT THE CURRENT STRING IS NOT FORGOTTEN
    ;
BEGIN
  PROCEDURE DELETETRANS(S);
  VALUE S;INTEGER S;
  COMMENT
    S POINTS TO AN S-T NODE
    A TRANSITION IS SELECTED FROM THE LIST AND EXCISED;
    IF EOG(MEM,NEXT(MEM,S))=0 THEN
BEGIN
  INTEGER D,T,N,RI,TT;
  COMMENT
    D WILL POINT TO TRANS TO BE DELETED
    T POINTS ONE TRANSITION PREVIOUS TO THE NODE TO BE DELETED
    TT SCANNED THROUGH TRANSITIONS
    RI LARGE RANDOM INTEGER
    N COUNTS NUMBER OF TRANSITIONS(SEE WRITE UP OF SEQUENTIAL
    SELECTION OF ALTERNATIVES);
    T:=NEXT(MEM,S);TT:=NEXT(MEM,T);RI:=ENTIER(RAND*1345677);N:=1;
    WHILE EOG(MEM,TT)=0 DO
BEGIN N:=N+1;
  IF(RI MOD N)=0 THEN T:=TT;
  TT:=NEXT(MEM,TT);
END;
D:=NEXT(MEM,T);
SETEOG(MEM,T,EOG(MEM,D));
IF EOA(MEM,D)=1 THEN SETEOA(MEM,T,1);
SETE(SELECT(MEM,T,NEXT(MEM,D)));
SETFREE(MEM,D);
END OF DELETETRANS;
BOOLEAN PROCEDURE CHECKTRANS(RCURR);
VALUE RCURR;INTEGER RCURR;
COMMENT
RETURNS TRUE IF NO TRANSITIONS POINT AT RCURR,FALSE OTHERWISE;
BEGIN
INTEGER PREV;LABEL EXIT;
COMMENT
PREV POINTS AT AN S NODE AT LEVEL N-1(SET BY FINDPREV) SUCH
THAT ALL TRANS TO RCURR(IF ANY) WILL LIE ABOVE IT IN THE TREE
EXIT GO HERE IF A TRANS TO RCURR IS FOUND;
PROCEDURE FINDPREV;
COMMENT SETS PREV;
BEGIN
INTEGER LEVEL;LABEL EXIT;
COMMENT
LEVEL HOLDS TREE LEVEL OF S NODES BEING SEARCHED INTREE
RBUF IS USED TO SELECT THE BRANCHES AT EACH LEVEL
EXIT GO HERE IF RUN OUT OF TREE;
COMMENT START AT BACK NODE ON PATTERN TREE;
PREV:=NEXT(MEM,NEXT(MEM,RBUF[2]));
FOR LEVEL:=2 STEP 1 UNTIL N(MEM)-1 DO
BEGIN COMMENT SEARCH ALONG LEVEL;
INTEGER X;LABEL EXITT;
COMMENT
X TEMPORARY VALUE FOR SEARCH
EXITT GO HERE WHEN CORRECT UP POINTER FOUND;
X:=RBUF[LEVEL+1];
PREV:=NEXT(MEM,PREV);COMMENT MOVE TO THE FIRST UP NODE;
WHILE PREV~0 DO
IF VAL(MEM,VAL(MEM,PREV))=X THEN
BEGIN COMMENT SHIFT PREV UP A LEVEL;
PREV:=VAL(MEM,PREV);GO TO EXITT;
END ELSE PREV:=NEXT(MEM,PREV);
EXITT:
IF PREV=0 THEN GO TO EXIT;
END OF SEARCH AT LEVEL;
EXIT;
END OF FINDPREV;
COMMENT MAIN BODY OF CHECKTRANS;
CHECKTRANS:=TRUE;
FINDPREV;
IF PREV~0 THEN
BEGIN INTEGER U;
COMMENT U SET TO ALL UP POINTERS AT LEVEL N-1;
U:=NEXT(MEM,PREV);
WHILE U~0 DO
BEGIN INTEGER T;
COMMENT T SCANS THE TRANSITIONS;
T:=NEXT(MEM,VAL(MEM,U));
COMMENT T NOW POINTS TO A DOWN NODE;
WHILE EO(MEM,T)=0 DO
BEGIN T:=NEXT(MEM,T);
IF VAL(MEM,T)=RCURR THEN
BEGIN CHECKTRANS:=FALSE;GO TO EXIT;
END;
END OF LOOP SCANNING TRANSITIONS;
U:=NEXT(MEM,U);
END OF LOOP SCANNING S T NODES;
END;
EXIT:
END OF CHECKTRANS;
COMMENT MAIN BODY OF RANDOMTERMINAL;
DELETETRANS(S);
IF S^=CURR(MEM) THEN
IF EOG(MEM,NEXT(MEM,S))^=0 THEN
IF CHECKTRANS(S) THEN
BEGIN SETFREE(MEM,NEXT(MEM,S));SETNEXT(MEM,S,0);
STATISTICS(MEM,2):=#+1;
END ELSE STATISTICS(MEM,4):=#+1;
END OF RANDOMTERMINAL;
PROCEDURE RANDOMNODE(S,LEVEL);
VALUE S,LEVEL;INTEGER S,LEVEL;
COMMENT
SELECTS AN UPWARD TRANSITION AT RANDOM FROM LEVEL IN TREE
DELETES U AND S NODES ON WAY BACK OUT;
IF NEXT(MEM,S)=0 THEN ELSE
IF LEVEL=N(MEM) THEN RANDOMTERMINAL(S) ELSE
BEGIN
INTEGER T,U,TT,NN,RI,SS;
COMMENT
SS S NODE UP ONE LEVEL
T ONE UP NODE PRIOR TO NODE WHICH MAY BE DELETED
U UP NODE WHICH MAY BE DELETED
IT SCANNED THROUGH ALL THE UP NODES
NN,RI SEE WRITE UP OF SEQUENTIAL RANDOM CHOICE;
T:=S;TT:=NEXT(MEM,S);NN:=1;RI:=ENTER(RAND*176543729);
WHILE NEXT(MEM,TT)^=0 DO
BEGIN NN:=NN+1;
IF(RI MOD NN)=0 THEN T:=TT;
TT:=NEXT(MEM,TT);
END;
U:=NEXT(MEM,T);
SS:=VAL(MEM,U);
RBUF[LEVEL+1]:=VAL(MEM,SS);
RANDOMNODE(SS,LEVEL+1);
COMMENT ON THE WAY OUT HERE CHECK IF DELETION CAN BE MADE;
IF NEXT(MEM,SS)=0 THEN
BEGIN
SETNEXT(MEM,T,NEXT(MEM,U));
SETFREE(MEM,SS);SETFREE(MEM,U);
END;
END OF RANDOMNODE;
PROCEDURE RANDOMMAP;
COMMENT PICKS A RANDOM NODE IN THE TREE;
BEGIN
INTEGER E;
E:=VAL(MEM,CIRCLE(MEM):=NEXT(MEM,CIRCLE(MEM)));
RANDOMNODE(NEXT(MEM,NEXT(MEM,E)),1);
END OF RANDOMMAP;
INTEGER I;
COMMENT MAIN BODY OF FORGET;
FOR I:=2 STEP 1 UNTIL N(MEM) DO RBUF[I]:=0;
IF NEXT(MEM,ROOT(MEM))=0 THEN RANDCMA = ERROR(MEM,21,10);
END OF FORGET;
INTEGER OF ERRORCOUNT;
COMMENT
OF USED TO CHECK IF NO INCREASE IN FREENUMBER AFTER CALL TO FORGET(PATHOLOGICAL CASE)
ERRORCOUNT DETECTS HOW OFTEN THIS OCCURS AND ABORTS AFTER 10 OCCURRENCES;
OF:=-1;ERRORCOUNT:=0;
WHILE F>FREENUMBER(MEM)DO
BEGIN
  STATISTICS(MEM,3):=#+1;
  FORGET;
  IF OF>FREENUMBER(MEM)THEN
  BEGIN
    IF DEBUGFLAG THEN ERROR(MEM,22,10);
    STATISTICS(MEM,5):=STATISTICS(MEM,5)+1;
    ERRORCOUNT:=ERRORCOUNT+1;
  END ELSE ERRORCOUNT:=0;
  OF:=FREENUMBER(MEM);
  END;
END OF ENSURE;
IV.A.3.6. Initialize

COMMENT
*****
*****
*****

INITIALIZE
*
CREATES INITIAL VALUES FOR RESERVED WORDS AND SETS UP
FREE LIST
*****
*****
*****;

PROCEDURE INITIALIZE(MEM,ORDER,MEMORYSIZE,INAME);
VALUE ORDER,MEMORYSIZE,INAME;INTEGER ORDER,MEMORYSIZE;REAL INAME;
REAL ARRAY MEM[0];
BEGIN INTEGER I;
COMMENT
SETS UP A NEW FELIX MEMORY IN MEM
ORDER AND MEMORYSIZE ARE SPECIFIED IN CALL, THE REST OF THE RESERVED
WORD VALUES ARE DEDUCED FROM THESE
THE BULK OF THE MEMORY IS LINKED ONTO A SINGLE FREE LIST;
IF MEMORYSIZE>MAXPOINTER THEN ERROR(MEM,7,0);
N(MEM):=ORDER;M(MEM):=MEMORYSIZE;NAME(MEM):=INAME;
FREE(MEM):=START(MEM);
FREENUMBER(MEM):=N(MEM)-FREE(MEM)+1;
IF FREENUMBER(MEM)<1 THEN ERROR(MEM,2,0);
IF ORDER<2 THEN ERROR(MEM,8,0);
CURR(MEM):=FORWDEPTH(MEM):=CIRCLE(MEM):=0;
FOR I:=1 STEP 1 UNTIL N(MEM) DO CURRP(MEM,I):=BUF(MEM,I):=0;
COMMENT SETUP FREE LIST;
FOR I:=FREE(MEM) STEP 1 UNTIL M(MEM)-1 DO SETNEXT(MEM,I,I+1);
SETNEXT(MEM,M(MEM),0);
FOR I:=1 STEP 1 UNTIL 5 DO STATISTICS(MEM,I):=0;
ROOT(MEM):=GET(MEM);
SPARE(MEM):=GET(MEM);
END OF INITIALIZE;
IV.A.3.7. Update

COMMENT
****
****
**** UPDATE
****
**** INSERTS A NEW ACTION-PATTERN PAIR INTO THE MEMORY
****
****
PROCEDURE UPDATE(MEM,NEWACT,NEWPAT,NEWRP);
VALUE NEWACT,NEWPAT,NEWRP;
REAL NEWACT,NEWPAT;INTEGER NEWRP;
REAL ARRAY MEM[0];
BEGIN
INTEGER ACTP,PATP,I,K,NUCURR;
COMMENT
ACTP POINTS TO E NODE OF ACTION SET BY FINDAP
PATP POINTS TO E NODE OF PATTERN SET FINDAP
I POINTER USED BY ALL ROUTINES IS MOVED UP TREE FROM ROOT TO NUCURR
K INDEX FOR UPDATING BUF AND CURRP
NUCURR POINTS TO S-T NODE TO WHICH CURR WILL POINT WHEN
UPDATING IS COMPLETED
*
AT ALL STAGES IF A NODE IS MISSING THEN IT IS CREATED AT THE TIME;
PROCEDURE FINDAP;
BEGIN
COMMENT USES DICTSEARCH TO LOCATE ACTION AND PATTERN IN AP DICTIONARY;
INTEGER PROCEDURE DICTSEARCH(x);
VALUE X;REAL X;
BEGIN
COMMENT
X VALUE OF A OR P BEING SEARCHED FOR
STARTS WITH I POINTING TO E NODE AT ROOT OF TREE.
I IS LEFT POINTING ONE BEYOND V NODE OF LOCATED A OR P
IF THERE IS NO NODE THEN IT IS CREATED;
LABEL EXIT;
COMMENT JUMP TO EXIT OUT OF LOOP WHEN NODE LOCATED;
WHILE TRUE DO
IF NEXT(MEM,I)=0 THEN
BEGIN
INTEGER K;
COMMENT NO V NODE SO MUST INSERT V,C AND D FOLLOWING NODE
(MAY BE E OR BACK NODE);
DICTSEARCH:=I;
I:=SETNEXT(MEM,I,GET(MEM));
K:=SETVAL(MEM,I,GET(MEM));
SETCCHAR(MEM,K,X);
I:=SETNEXT(MEM,I,GET(MEM));
GO EXIT;
END ELSE
BEGIN REAL CHARS;
CHARS:=CHAR(MEM,VAL(MEM,NEXT(MEM,I)));
IF CHARS IS X THEN
BEGIN DICTSEARCH:=I;I:=NEXT(MEM,NEXT(MEM,I));GO TO EXIT;
BEGIN

COMMENT VALUE AT NODE NOT EQUAL TO X SO MOVE UP VIA BINARY NODE IF NECESSARY CREATING IT AND AN E NODE;
I := IF VAL(MEM, I) = 0 THEN SETVAL(MEM, I, GET(MEM)) ELSE VAL(MEM, I);
I := IF ORDER(CHARS, X) THEN
   (IF NEXT(MEM, I) = 0 THEN SETNEXT(MEM, I, GET(MEM))
   ELSE NEXT(MEM, I)) ELSE
   (IF VAL(MEM, I) = 0 THEN SETVAL(MEM, I, GET(MEM)) ELSE
   VAL(MEM, I));

END;
END OF SCAN LOOP;

EXIT;
END OF DICTSEARCH;

COMMENT MAIN BODY OF FINDAP;
I := ROOT(MEM);
ACTP := DICTSEARCH(MEMACT);
PATP := DICTSEARCH(MEMPAT);
COMMENT IF NECESSARY SET THE BACK POINTER;
IF VAL(MEM, I) = 0 THEN
BEGIN INTEGER F, G; // THIS IS A NEW A-P ENTRY SET BACK
% POINTER AND ENTRY IN FORGET CIRCLE
SETVAL(MEM, I, ACTP);
G := GET(MEM);
IF F := CIRCLE(MEM) = 0 THEN
SETNEXT(MEM, G, G) ELSE
BEGIN SETNEXT(MEM, G, NEXT(MEM, F)); SETNEXT(MEM, F, G);
END;
CIRCLE(MEM) := G; SETVAL(MEM, G, PATP);
END;
END OF FINDAP;

PROCEDURE FINDNODE(K);
VALUE K; INTEGER K;
BEGIN
LABEL EXIT; INTEGER X;
COMMENT
THE UP POINTERS AT LEVEL K ARE SEARCHED TO FIND ONE WHICH POINTS AT A SYMBOL EQUALLING BUF(K)
A NEW UP ENTRY IS CREATED IF NECESSARY
CURP(K) IS UPDATED TO POINT TO THE UP POINTER
I INITIALLY POINTS AT AN S NODE (OR POSSIBLY A BACK NODE IN THE AP DICTIONARY) AND IS LEFT POINTING AT AN S OR S-T NODE UP ONE LEVEL
*
EXIT GO HERE WHEN ENTRY LOCATED
X := BUF(MEM, K);
IF X = 0 THEN I := 0 ELSE
WHILE TRUE DO
BEGIN
IF NEXT(MEM, I) = 0 THEN
BEGIN COMMENT CREATE NEW U AND S NODES;
CURR(MEM, K) := I := SETNEXT(MEM, I, GET(MEM));
I := SETVAL(MEM, I, GET(MEM)); SETVAL(MEM, I, X);
GO TO EXIT;
END;
END;
COMMENT STEP ON ONE TO THE NEXT UP POINTER;
I:=NEXT(MEM,I);
IF VAL(MEM,VAL(MEM,I))=X THEN
BEGIN CURRP(MEM,K):=I;I:=VAL(MEM,I);GO TO EXIT;
END;
END OF SCAN LOOP;
EXIT:
END OF FINDNODE;
PROCEDURE FINDDOWN;
BEGIN
COMMENT CHECKS OUT DOWN POINTER IMMEDIATELY FOLLOWING S-T NODE;
IF NEXT(MEM,I)=0 THEN
BEGIN COMMENT CREATE NEW DOWN POINTER;
I:=SETNEXT(MEM,I,GET(MEM));SETVAL(MEM,I,PATP);
SETEOA(MEM,I,1);SETEOG(MEM,I,1);
END ELSE
I:=NEXT(MEM,I);
END OF FINDDOWN;
PROCEDURE FINDTRANSITION;
BEGIN
INTEGER L;LABEL EXIT;BOOLEAN B;
COMMENT
TRANSITIONS RADIATING FROM CURR(OLD VALUE)ARE CHECKED TO SEE
IF THEY POINT TO NUCURR.A NEW TRANSITION IS INSERTED IF NECESSARY
L SET TO EACH TRANSITION IN TURN
EXIT GO HERE WHEN TRANSITION LOCATED OR CREATED
B USED TO CONTROL LOOP;
PROCEDURE CREATE;
BEGIN INTEGER NU;
COMMENT CREATE A NEW TRANSITION IMMEDIATELY FOLLOWING L
FREQUENCY FIELD SET TO ZERO(PAIN REWARD WILL BE SET LATER)
NU POINTS TO NEW NODE;
NU:=GET(MEM);
SETNEXT(MEM,NU,NEXT(MEM,L));SETEOA(MEM,NU,1);
SETEOG(MEM,NU,EOG(MEM,L));SETVAL(MEM,NU,NUCURR);
SETEOG(MEM,L,0);SETNEXT(MEM,L,NU);
L:=NU;
COMMENT FORCE SEARCH WHEN GETGOODACTION NEXT CALLED;
FORMDEPTH(MEM):=0;
USTATUS(MEM):=1;
END OF CREATE;
PROCEDURE MUSTBEHERE;
BEGIN
BOOLEAN B;INTEGER IL;LABEL EXIT;
COMMENT
A TRANSITION WITH THE CORRECT ACTION(BUT NOT NECESSARILY THE
CORRECT PATTERN)HAS BEEN FOUND THE CURRENT TRANSITION MUST BE
IMMEDIATELY FOLLOWING OR BE INSERTED THERE
AT START OF ROUTINE L POINTS ONE PRIOR IN THE TRANSITION LIST TO
THE NODE WITH THE FIRST OF THE CURRENT ACTION
AT THE END L POINTS TO THE LOCATED TRANSITION
*
EXIT GO HERE WHEN TRANSITION LOCATED(OR CREATED)
IL INITIAL VALUE OF L(WHEN ROUTINE ENTERED)
B BOOLEAN TO CONTROL OUTER LOOP(CAN BE REPLACED IF
259
DO...UNTIL...,AVAILABLE);
IL:=L;B:=TRUE;
WHILE B DO
BEGIN
L:=NEXT(MEM,L);
IF VAL(MEM,L)=NUCURR THEN
BEGIN INTEGER F;
COMMENT TRANSITION HAS BEEN LOCATED
UPDATE FREQUENCY INFO FOR ALL TRANSITIONS WITH CURRENT ACTION;
F:=FREQ(MEM,L)+1;
IF FC<MAXFREQ THEN SETFREQ(MEM,L,F)ELSE
BEGIN INTEGER I;BOOLEAN BB;
I:=IL;BB:=TRUE;
WHILE BB DO
BEGIN I:=NEXT(MEM,I);SETFREQ(MEM,I,(FREQ(MEM,I)+1)/2);
BB:=EOA(MEM,I)=0;
END;
END;
GO TO EXIT;
END OF IF ;
B:=EOA(MEM,L)=0;
END OF OUTER LOOP;
COMMENT INSERT NEW TRANSITION;
SETOA(MEM,L,0);CREATE;
EXIT:
END OF MUSTBEHERE;
COMMENT MAIN BODY OF FINDTRANSITION;
USTATUS(MEM):=0;
L:=NEXT(MEM,CURR(MEM));
COMMENT SEQUENCE OF NODES TO GET TO UNIQUE ID FOR ACTION
L-->TRAN-NX-->TRAN-VAL-->ST-NX-->DOWN-ACTION-->E(ACT);
WHILE EOQ(MEM,L)=0 DO
IF ACTION(MEM,NEXT(MEM,VAL(MEM,NEXT(MEM,L))))^=ACTP THEN
BEGIN BOOLEAN B;
B:=TRUE;
WHILE B DO
BEGIN L:=NEXT(MEM,L);B:=EOA(MEM,L)=0;
END;
END ELSE
BEGIN MUSTBEHERE;GO TO EXIT;
END;
CREATE;
EXIT:
SETRP(MEM,L,NEWRP);
END OF FINDTRANSITION;
COMMENT MAIN BODY OF UPDATE;
COMMENT ENSURE THERE IS ENOUGH ROOM TO CREATE A COMPLETELY
NEW STRING WITHOUT FORGETTING DURING THE UPDATE;
USTATUS(MEM):=2;
STATISTICS(MEM,1):=FREENUMBER(MEM);
ENSURE(MEM,2*N(MEM)+10);
STATISTICS(MEM,1):=FREENUMBER(MEM)-STATISTICS(MEM,1);
FINDDAP;
FOR K:=N(MEM)STEP -1 UNTIL 2 DO BUF(MEM,K):=BUF(MEM,K-1);
BUF(MEM,1):=PATP;
CURRP(MEM,1):=I;
FOR K:=2 STEP 1 UNTIL N(MEM) DO FINDNODE(K);
NUCURR:=I;
IF NUCURR^2=0 THEN FINDDOWN;
IF CURR(MEM)\^=0 AND NUCURR^2=0 THEN FINDTRANSITION;
CURR(MEM):=NUCURR;
END OF UPDATE;
IV.A.3.8. Predict

PROCEDURE PREDICT(MEM, UNRELIABILITY, ANS);
VALUE UNRELIABILITY;
INTEGER UNRELIABILITY;
REAL ARRAY MEM[O], ANS[O, O];
BEGIN
COMMENT
UNRELIABILITY SAYS HOW UNRELIABLE A PREDICTION MAY BE USED IN ACTION
SEARCH
ANS HOLDS PREDICTED ACTIONS(TYPE 1 OR 2 FOR FIRST ACTION OF
GROUP) AND PATTERNS(TYPE 0);
LABEL ABORTPREDICT;
PROCEDURE COLLATEACTIONS(T); VALUE T; INTEGER T; FORWARD;
PROCEDURE INITIALCASE;
COMMENT
SETS UP INITIAL LINKED LIST FOR START OF SEARCH
MAKES A RECURSIVE TREE SEARCH IF THERE IS AN UNRELIABLE PREDICTION;
BEGIN COMMENT UNRELIABLE CASE;
INTEGER LO, C;
COMMENT
EXIT GO HERE WHEN HAVE FINISHED SETTING UP LIST
LO MINIMUM LEVEL TO WHICH ALLOWED TO DESCEND
C TEMPORARY POINTER USED IN CHECKING BRANCHES FROM
NODLES IN TREE;
PROCEDURE HOOKUP(LEVEL, NODE);
VALUE LEVEL; INTEGER LEVEL, NODE;
IF LEVEL=N(MEM) THEN COLLATEACTIONS(NODE) ELSE
BEGIN INTEGER T;
T:=NEXT(MEM, NODE);
WHILE T<>0 DO
BEGIN HOOKUP(LEVEL+1, VAL(MEM, T)); T:=NEXT(MEM, T);
END;
END OF HOOKUP;
COMMENT MAIN BODY OF INITIALCASE;
LO:=N(MEM)-UNRELIABILITY;
IF LO<1 THEN
BEGIN ERROR2(UNRELIABILITY, 9, 10); LO:=1;
END;
UNCERT(MEM):=N(MEM);
C:=BRANCH(MEM, LO);
IF C^0 THEN
BEGIN HOOKUP(LO, C); UNCERT(MEM):=N(MEM)-LO;
END;
END OF INITIALCASE;
PROCEDURE COLLATEACTIONS(T);
VALUE T; INTEGER T;
COMMENT INSERTS ACTIONS AND PATTERN FOR EACH GROUP PREDICTED;
BEGIN INTEGER TYP; REAL V;
TYP:=2;
T:=NEXT(MEM, T);
WHILE EOQ(MEM, T)=0 DO
BEGIN
IF PUT(ANS, TYP, APIC(MEM, NEXT(MEM, VAL(MEM, NEXT(MEM, T))))))
THEN GO TO ABORTPREDICT;

262
TYP := 1;
DO
T := NEXT(MEM, T);
V.RPF := RP(MEM, T); V.RPF2 := RP(MEM, VAL(MEM, T));
V.TYPEF := 0; V.FREQF := FREQ(MEM, T);
IF PUT(ANS, V, PPIC(MEM, NEXT(MEM, VAL(MEM, T))))
THEN GO TO ABORTPREDICT;
END UNTIL EOA(MEM, T) = 1;
END;
END OF COLLATEACTIONS;
COMMENT MAIN BODY OF PREDICT;
CLEAR(ANS);
UNCERT(MEM) := 0;
INITIALCASE;
ABORTPREDICT;
END OF PREDICT;
CHAPTER V
BEST MATCHING ALTERNATIVES

The particular form of best match retrieval used by the FLM (Chapter I) has three important properties:

i) it allows traces which have been stored to be retrieved even when none of them are identical to the current buffer. This greatly increases the number of circumstances in which the FLM can predict a reasonable action and so makes programming easier;

ii) programs once entered will not be disturbed by later inputs;

iii) it can be implemented on a serial computer so that the time to store and retrieve a trace is independent of the memory size.

If any general theory of adaptive devices, such as the FLM, is to be developed then it is necessary to consider alternative best matching schemes. The first property above might be shared by many other best match schemes. For example, one might assign each event in the buffer a real coordinate and then predict, using the traces with the smallest Euclidean distance from the buffer. Alternatively, one might assign each event a bit pattern and use those traces with the smallest Hamming distance from the buffer. However, few results are available about the computational speed of such algorithms or their ability to stably hold a program once it has been entered.

In (Cleary, 1979), which has been bound into this thesis, I consider the computational speed of best matching using Euclidean distance as a measure. It is shown that the computing time on a serial computer, for the best available algorithm, increases exponentially with the dimension of the Euclidean space. So except for small dimensions (say 2 or 3) it would be infeasible to include such a best
match in a device like the FLM.

However this criterion of computational speed can be completely dropped if sufficient parallel hardware is available to compare each trace simultaneously with the current buffer. For example, the brain might well be able to do this, in which case it could plausibly employ any one of a large number of best match schemes. Only criterion ii) above remains to delimit these possibilities. However no results are available at present about the ability of any other best match schemes to stably retain programs.
In Chapter IV I introduced the network representation of an FLM memory. This representation allows one to start at a particular trace in the memory and quickly examine paths of predicted events into the future. This is useful when one is attempting to choose actions, and when the 'utility' or 'value' of some of these future events is known. One might then choose actions leading to paths of events with high value and avoid those of low value.

One way to do this is to assign a derived value to every trace in the memory. The derived value is calculated from the value attached to the trace itself and the derived values of its immediately following traces. The action leading to the trace with the highest derived value is then chosen. This chapter is concerned with the process of calculating such derived values within a network. The algorithm which will be developed to do this is cast in a general form which applies to any network, not just those representing FLM memories.

A well known example of such networks is game trees, where each possible position in a board game, such as chess, is represented by a node in a network. (See (Slagle,1971) for an introduction to this subject.) Depending on whose move it is, the value of a node is the minimum or maximum of the nodes which can be reached from it in one move. In this case the values at each node are well defined and easily calculated by moving from the leaves of the game tree back to the root. However, in the more general case the net may contain loops of nodes and this simple process does not necessarily lead to the assignment of a consistent derived value to each node.
This problem of calculating derived values will be formulated in terms of sets of simultaneous equations. An iterative or relaxation algorithm for solving such sets of equations will be given. It is similar to the familiar iterative methods for solving real vector equations of the form $\vec{x} = f(\vec{x})$. The solution is obtained by a series of approximations $\vec{x}_n$ where $\vec{x}_n = f(\vec{x}_{n-1})$, (Henrici, 1964, p97ff), (Hildebrand, 1974, p561ff). The Gauss-Seidel algorithm for the solution of linear simultaneous equations is a well known example of such a process. Note however that for the class of equations considered here this simple iterative technique is insufficient and it must be periodically interrupted and the variables reset if convergence is to be guaranteed.

A sufficient condition for the algorithm to converge will be proven. It will be shown to be dependant only on the set of functions used to construct the equations and independent of any particular equation. If it holds, then any set of equations using just these functions can be solved. A number of examples of sets of functions satisfying the condition will be exhibited, including an infinite family of such sets of functions.

Both the problem formulated here and its solution are new. The algebraic techniques used in proving the convergence of the solution while involving some detail require only a familiarity with elementary notions such as reflexive relations.
VI.1. STATEMENT OF PROBLEM

Consider a set of simultaneous equations over some arbitrary number of variables, say:

\[
\begin{align*}
N &= F(N, N) \\
&= F(N, 1) \\
&= F(N) \\
&= F(N)
\end{align*}
\]

where \( F_1 \) and \( F_2 \) are functions over a finite domain. One way to calculate a solution to this problem might be to assign values to \( N_1 ... N_k \) then use the equations to calculate new values and to continue recalculating until a solution is found. In general, however, this procedure will not converge to a solution. This chapter proposes a condition which ensures that a generalized form of this recalculation procedure will converge. The condition is on the set of functions allowed (in the equations above \( F_1, F_2 \) and the constant function 1). If the condition holds then a solution exists for any set of equations using the allowed functions and the procedure will converge to a solution.

This imposes a very strong restriction on the functions involved. As it guarantees a solution for any set of equations, then it cannot hold unless every possible set of equations does in fact have a solution. If, for example, one function allowed was addition modulo-\( n \) then the condition cannot hold as the equation \( N = N + 1 \) has no solution.

Some feeling for the types of functions which satisfy the
condition can be obtained from the results of section 8. There it is shown that if there exists a partial order on the domain of the functions which is preserved by the functions then they satisfy the condition. This provides a varied and infinite (although not exhaustive) family of functions which satisfy the condition.

It is in situations where large numbers of different sets of equations are to be solved that it is worthwhile to spend the time pre-analysing the set of functions involved. If, say, only a single set of equations were involved then checking out the condition may be little different from enumerating all possible solutions and checking them. If a small number of equation sets is to be solved then possibly it will be worthwhile to re-arrange the equations so as to achieve a more manageable set of functions. For example, the non-trivial relation to be solved in the set of equations above is:

\[ N_1 = F_1(N_1, F_1(N_1, 1)) \]

(Once the value for \( N_1 \) has been solved it is trivial to solve for \( N_2, \ldots, N_k \).) This can then be investigated in terms of the single allowed function \( \lambda x. F_1(x, F_1(x, 1)) \) rather than the three functions, \( F_1, F_2 \) and 1. This reduced problem may allow of solutions for all its equation sets where the more general problem does not.

There is a slight restriction on the equations allowed; that each variable, \( N_i \), can appear on the left of an equation only once. The formalization of this uses a map, \( I \), which specifies, for a given assignment of values to the variables, how to calculate a new value for each variable. If \( I \) causes no change then a solution has been obtained.

This restriction allows the variables in the equations to be treated informally as the nodes of a directed graph. The value at a node is then a function of the values at the nodes descended from it.
Fig. 1 shows the set of equations above illustrated this way.

Section 1 below formalizes the notions of a set of equations and a solution.

---

Fig. 1 Example Equation Set as a Directed Graph
VI.1.1. Preliminary Definitions

VI.1.1.1. Equation Set

An equation set (e.s) is a four-tuple \((A,F,N,I)\) where:

- \(A\) is a finite set of values;
- \(F = \{ F : (F : A \rightarrow A) \} \) is a set of functions over \(A\) (where \(o(F) \geq 0\) is the order of the function);
- \(N\) is a finite set of variables;
- \(I : A \rightarrow A\) satisfying the conditions
  \(\forall N \in N \exists F \in F, N \dots N \in N\) such that
  \(\forall V : N \rightarrow A, I(V) : N \rightarrow F(V(N_{1}), V(N_{2}) \ldots V(N_{o(F)}))\).

It is \(I\) which specifies the equations which relate the values of the variables. A mapping \(V : N \rightarrow A\) can be thought of as an assignment of values to the variables in \(N\). \(I\) then specifies how to calculate a new assignment of values using the old values.

VI.1.1.2. Evaluation

An evaluation \(V\), on \((A,F,N,I)\) is a mapping

\(V : N \rightarrow A\).

VI.1.1.3. Consistent

An evaluation \(V\) on \((A,F,N,I)\) is consistent iff

\(I(V) = V\).
VI.1.2. Solution

The problem considered here is that of finding a consistent evaluation when given some e.s. The results to be obtained depend only on the set \( F \), however, and not on the structure of any particular e.s.

The technique which forms the basis of the following results is illustrated in Fig. 2. This shows a fragment of a graph representing the equation \( N_4 = F(N_2, N_3) \). Three snapshots of the graph are given, the nodes having been re-evaluated between each. There are three "pairs of successive values" in this diagram, \( \langle a_1, b_1 \rangle, \langle a_2, b_2 \rangle, \) and \( \langle c, d \rangle \). By successive value is intended; successive value for a particular node obtained by recalculating it. So \( \langle c, d \rangle \) is a pair of successive values for the node \( N_4 \). Throughout what follows all pairs considered will be such successive pairs of values.

That \( \langle c, d \rangle \) occurs can be deduced from the occurrence of the pairs \( \langle a_1, b_1 \rangle, \langle a_2, b_2 \rangle \) (\( c = F(a_1, a_2), d = F(b_1, b_2) \)). Consider a situation where the set of all pairs which can occur on the first re-evaluation are known. Then all pairs which can occur on the second re-evaluation are known, and so on until all possible pairs which can ever occur are known. For example, if all the nodes are initialized to one value \( a_0 \) then the only pairs which can occur on the first re-evaluation are of the form \( \langle a_0, b \rangle \), where \( b \in A \).

Given some such set of pairs \( S \subseteq A \times A \) which can occur on the first re-evaluation, \( C(S) \), defined below, is the set of all pairs which can ultimately occur. (Note that this is independent of any particular set of equations and depends only on the set \( F \) of functions.) Given a knowledge of this set it is possible to tell whether on successive re-evaluations the value of some node might
execute a loop. For example, if \(<0, 1>, <1, 2>\) and \(<2, 0>\) are members of \(C(S)\), then some node might go through the loop, \(\ldots 0, 1, 2, 0, 1, 2, \ldots\). \(Cyc(S)\), defined below, is just the pairs of values which can be joined by such cycles (for example \(<0, 2>\) is a member of \(Cyc(S)\) as well as \(<0, 1>, <1, 2>, <2, 0>\)).

If \(Cyc(S)\) contains only pairs of the form \(<a, a>\) then no cycles of values can occur. It can be seen then that if the nodes are successively re-evaluated under these circumstances, a steady solution must eventually be attained.

---

**Fig 2. Illustration of the Use of Successive Pairs of Values**
VI.2. PRELIMINARY RESULTS

Before embarking on the main proof later I need some preliminary definitions and results. These provide 'tools' for handling the sets of pairs described above. In general the definitions are dependent on the sets A, F, N, but where no ambiguity can arise these will be omitted as subscripts. The set S used below is presumed to be a subset of A x A.

VI.2.1. Definitions

VI.2.1.1. D

\[ D = \{<a, a> : a \in A\} \]

VI.2.1.2. \( \| \)

Given two evaluations V, W : \( \overline{N} \rightarrow A \) then

\[ V \upharpoonright W = \{<V(N), W(N)> : N \in \overline{N}\} \]

VI.2.1.3. \( \mathcal{C} \)

\( \mathcal{C}(S) \) is the minimum set such that

i) \( S \subseteq \mathcal{C}(S) \);

ii) \( <a, b> \in \mathcal{C}(S), 1 \leq i \leq 0(F) \]

\[ \Rightarrow <F(a, ...a_{i}), F(b, ...b_{i})> \in \mathcal{C}(S) \] for all \( F \in F \).

1 \( \alpha(F) \)

1 \( \alpha(F) \)

274
VI.2.1.4. $S^*$

$S^*$ is the minimum set such that:

i) $S \subseteq S^*$;

ii) $(a, b), (b, c) \in S^* \Rightarrow (a, c) \in S^*$.

($S^*$ is the minimum transitive set of pairs containing $S$.)

VI.2.1.5. Cyc

$\text{Cyc}(S) = \{(a, b) : (a, b), (b, a) \in C(S)^*\}$

VI.2.2. Examples from Definitions

Consider for example the set of pairs in Fig. 3a, which might be $\text{Cyc}(S)$ for some set $S$. This indicates that the sequences \ldots 2, 3, 2, 3, \ldots or \ldots 4, 5, 4, 5, 4, 5, \ldots can occur as the successive values of one node. However, the sequence \ldots 1, 2, 1, 2, \ldots cannot occur. (Figs. 3b, c are explained later.)

It may be noted that there are no pairs of the form $(0, a)$ or $(a, 0)$ in the set of Fig. 3a. This indicates that 0 does not occur as the value of any node. It is this possibility which prevents sets of the form $\text{Cyc}(S)$ being reflexive, (that is every element of the form $(a, a)$ is in the set) and makes it necessary for the notion of semi-reflexive to be introduced below. It will be shown that semi-reflexivity is preserved by $\mathcal{C}$, $*$ and Cyc. This property is sufficient for the proof of theorem 2.3.6 which, in turn, forms the basis for the main result of Theorem 5.2. All the sets (of pairs) considered in Theorem 5.2 will be semi-reflexive as a direct consequence of their definition.
(a) $\text{Cyc}(S)$

\begin{array}{cccccc}
 0 & 1 & 2 & 3 & 4 & 5 \\
 0 & & & & & \\
 1 & x & & & & \\
 2 & & x & x & & x \quad \text{indicates that} \quad <a,b> \in \text{Cyc}(S) \\
 a & 3 & & x & x & \\
 4 & & x & x & & \\
 5 & & x & x & & \\
\end{array}

(b) Classes: \{1\}, \{2,3\}, \{4,5\}

(c) If $R = \text{Cyc}(S)$ then the functions $e_i$ which need $i$ to be considered are: (see 5.)

\begin{tabular}{c|cccc}
 0 & - & - & - & - \\
 1 & 1 & 1 & 1 & 1 \\
 2 & 2 & 3 & 2 & 3 \\
 3 & 2 & 3 & 2 & 3 \\
 4 & 4 & 4 & 5 & 5 \\
 5 & 4 & 4 & 5 & 5 \\
\end{tabular}

\textbf{Fig. 3. Example of a Set of the Form Cyc(S).}
VI.2.3. Semi-Reflexive Sets

A set $S \subseteq A \times A$ is semi-reflexive iff

$$\forall a, b \in A, \langle a, b \rangle \in S \Rightarrow \langle a, a \rangle \in S.$$  

VI.2.3.1. Lemma

If $S$ is semi-relexive then $\mathcal{C}(S)$ is semi-reflexive.

Proof:

Consider the following construction of $\mathcal{C}(S)$:

let $S = S$

and $S = S \cup \{ \langle F(a, \ldots, a^i), F(b, \ldots, b^j) \rangle :$

$1 \leq j \leq o(F), \langle a, b \rangle \in S, F \in F \}, i \geq 0$

After a finite number, $m$, of steps $S_{m+1} = S_m$, because $A \times A$ is finite, and so $\mathcal{C}(S) = S$. Consider the first $i$ such that

$$\exists a, b, \langle a, b \rangle \in S^i, \text{ and } \langle a, a \rangle \notin S^i.$$  

$\langle a, b \rangle$ is not a member of $S^i$ so

$$a = F(a, \ldots, a),$$

$1 \quad o(F)$

$$b = F(b, \ldots, b),$$

$1 \quad o(F)$

and $\langle a, b \rangle \in S, 1 \leq j \leq o(F)$

$\langle a, a \rangle \notin S.$

This implies $\langle F(a, \ldots, a), F(a, \ldots, a) \rangle \in S$

$1 \quad o(F) \quad 1 \quad o(F) \quad i$

that is $\langle a, a \rangle \in S.$

But this is a contradiction so each $S^i$ must be semi-reflexive in

277
particular $S_m$ which is equal to $C(S)$.

VI.2.3.2. Lemma

If $S$ is semi-reflexive then $S^\#$ is semi-reflexive.

Proof:
Consider a construction (similar to above) for $S^\#$.

Let $S = S$

and $S^\# = S \cup \{ <a, c> : <a, b>, <b, c> \in S \}, i \geq 0.$

As above a contradiction is obtained for the first $S_{i+1}$ which is not semi-reflexive. For if

$<a, c> \in S$ and $<a, a> \notin S$ then

$S_i \cup \{ <a, c> : <a, b>, <b, c> \in S \}, i \geq 0.$

$S_{i+1}$

$<a, b>, <b, c> \in S \Rightarrow <a, a> \in S_i$

$\Rightarrow <a, a> \in S$, a contradiction!

VI.2.3.3. Lemma

If $S$ is semi-reflexive then $C(S)^\#$ and $Cyc(S)$ are semi-reflexive.

Proof:

By direct application of 2.3.1 and 2.3.2.

VI.2.3.4. Lemma

The relation between $a, b$ defined by

$<a, b> \in Cyc(S)$ is symmetric and transitive

and

the relation between two evaluations $V, W$ defined by

$V \preceq W \subseteq Cyc(S)$ is symmetric and transitive.

Proof:
Both these follow immediately from the definition of $\text{Cyc}(S)$.

VI.2.3.5. Lemma

If $\langle a, b \rangle \in S^*$ then $\exists a = a, a, a \ldots a = b$ such that

\[
\begin{array}{cccc}
\text{0} & \text{1} & \text{2} & \text{1} \\
\text{i} & \text{i} & \text{i} & \text{i+1} \\
\end{array}
\]

$\langle a, a \rangle \in S, 0 \leq i \leq 1$.

Proof:

This follows from the construction used in 2.3.2 to calculate $S^*$.

VI.2.3.6. Theorem

The following theorem is the central one of this section and ties together the preceding lemmas.

Theorem

If $\langle a, b \rangle \in G(S)^*, 1 \leq i \leq o(F),$

\[
\begin{array}{cccc}
\text{i} & \text{i} & \text{i} & \text{i} \\
\end{array}
\]

and $S$ is semi-reflexive then

$\langle F(a, \ldots a), F(b, \ldots b) \rangle \in G(S)^*$.

Proof:

From 2.3.5 for each $i$ there is a sequence,

\[
\begin{array}{cccc}
\text{0} & \text{1} & \text{2} & \text{m}(i) \\
\text{a} = a, a, a, \ldots a = b \\
\text{i} & \text{i} & \text{i} & \text{i} \\
\text{t} & \text{t+1} \\
\end{array}
\]

such that $\langle a, a \rangle \in G(S), 0 \leq t < m(i)$.

Now $\langle a, a \rangle \in G(S), 0 \leq t < m(i)$

\[
\begin{array}{cccc}
\text{0} & \text{0} \\
\text{i} & \text{i} \\
\end{array}
\]

and in particular $\langle a, a \rangle \in G(S)$.
Note that it is not necessarily true that
\[
\langle a_i, a_i \rangle \in \mathbb{S}(S).
\]

For the following construction each sequence needs to be of equal length so let \( m = \max_i m(i) \) and for each \( i \) where \( m(i) < m \) the beginning of the sequence can be expanded by \( m - m(i) \) repetitions of \( a_i^0 \). So for each \( i, 1 \leq i \leq o(F) \) there is a sequence

\[
0 \ 1 \ 2 \ m
a = b, b, b \ldots b = b
i \ i \ i \ i \ i \ i
\]

where \( \langle b, b \rangle \in \mathbb{S}(S), 0 \leq t < m. \)

Now consider the sequence:

\[
0 \ 0 \ 0
\]
\[
c = F(b, b, \ldots b) = F(a, \ldots a)
\]
\[
0 \ 1 \ 2 \ o(F) \ 1 \ o(F)
\]
\[
1 \ 1 \ 1
\]
\[
c = F(b, b, \ldots b)
\]
\[
1 \ 1 \ 2 \ o(F)
\]
\[
\ldots
\]
\[
c = F(b, b, \ldots b) = F(b, \ldots b)
\]
\[
m \ 1 \ 2 \ o(F) \ 1 \ o(F)
\]

But, \( \langle c, c \rangle \in \mathbb{S}(S), \) from definition 2.1.3(ii),
\[
i \ i+1
\]

and \( \langle c, c \rangle \in \mathbb{S}(S)*, \) from definition 2.1.4(ii).

So finally \( \langle F(a, \ldots a), F(b, \ldots b) \rangle \in \mathbb{S}(S)*. \)

\[
1 \ o(F) \ 1 \ o(F)
\]

**Corollary**

If \( S \) is semi-reflexive and

\[
\langle a, b \rangle \in \text{Cyc}(S), 1 \leq i \leq o(F)
\]

then \( \langle F(a, a, \ldots a), F(b, \ldots b) \rangle \in \text{Cyc}(S). \)
Proof:

\[ <a, b>, <b, a> \in C(S)^i \]

so apply the above theorem twice.

VI.3. EVALUATION MAPS

The condition for convergence obtained later is to a large extent independent of the precise way in which the recalculation of the nodes is done. Using \( I \), each node is recalculated using the old values whereas in \( 6 \), it is shown that each node as it is recalculated can then be used immediately for recalculating following values. For example, there are two different orders in which values at the nodes in the graph below could be recalculated:

- \( N_2 \rightarrow G(a) \rightarrow N_4 \)
- \( N_2 \rightarrow G(a) \rightarrow N_4 \)

The different techniques for evaluating the nodes are just functions from \( \mathbb{A}^N \rightarrow \mathbb{A}^N \) and the 'evaluation maps' defined below are
such mappings which satisfy a sufficient condition to ensure convergence.

The different evaluation maps can be compared with the different algorithms used in solving linear simultaneous equations. For example Jacobi iteration uses the old value of variables, as in $I$ above, and Gauss-Siedel iteration uses the latest value, as in $I$ below (Hildebrand, 1974).

VI.3.1. Definition - Evaluation Map

$H : A \rightarrow A^N$ is an evaluation map (e.m.) iff for evaluations $V, W : N \rightarrow A$ and $S$ semi-reflexive

1. $V \upharpoonright W \subseteq C(S)^* \Rightarrow H(V) \upharpoonright H(W) \subseteq C(S)^*$

2. $H(V) = V$ iff $I(V) = V$.

VI.3.2. Theorem

$I$ is an evaluation map.

Proof:
For condition (i) apply Theorem 2.3.6 and (ii) is trivial.

VI.3.3. Theorem

For an e.m. $H$ and a sequence of evaluations

$W_0, W_1, \ldots, W_i, \ldots$

where

$W_i = H(W)_{i+1} = i$

$W_{i+1} = i$
and \( W \leq S \leq 0 \leq 1 \) 

then \( \exists 1 \) such that

\[
W \mid W \leq \text{Cyc}(S) .
\]

Proof:

\[
W \mid W \leq S \text{ implies } W \mid W \leq C(S)^* ,
\]

so applying 3.1(i) repeatedly

\[
W \mid W \leq C(S)^* , \ i \geq 0 .
\]

But, by the transitivity of \( C(S)^* \) this implies

\[
W \mid W \leq C(S)^* , \ i \geq 0, j \geq 0 .
\]

For a particular e.s. there is only a finite set of possible evaluations so \( \exists 1, m \geq 0, l < m - 1 \) such that

\[
W = W .
\]

This implies

\[
W \mid W \leq C(S)^* .
\]

Also, \( W \mid W \leq C(S)^* \) (from (1)),

\[
W \mid W \leq C(S)^* .
\]

and as \( W = W ,
\]

\[
W \mid W \leq C(S)^* .
\]

From (2), (3) and Definition 2.1.5

\[
W \mid W \leq \text{Cyc}(S) .
\]
VI.4. CASCADES

I now have sufficient tools to state the condition for convergence and to prove its sufficiency. However, to obtain a general and effective procedure it is necessary to extend the simple process of re-evaluation. Consider the set $\text{Cyc}(S)$ given as an example in Fig. 3a. This guarantees (from the last theorem) that, using the simple process of re-evaluation, only certain cycles of values can occur. It is possible, however, to go from such non-steady situations to a steady solution.

Consider a set of equations which has been brought to the stage where another re-evaluation generates only pairs of values lying in some set $\text{Cyc}(S)$ (see Fig. 3 for example). At this point each node can be re-initialized. The new re-initialization value is dependent only on the previous value of the node. If, after this re-initialization, the nodes are again re-evaluated repeatedly then they will converge to a steady solution or to another (more restricted) cycle.

For an example of such a re-initializing function consider the set $\text{Cyc}(S)$ of Fig. 3. This naturally partitions $A$ into the classes $\{1\}$, $\{2, 3\}$ and $\{4, 5\}$. The re-initializing function must assign all members of such a class to a single member of the class. So all nodes with value 1 would be undisturbed, with 2 or 3 re-initialized to 2 (say) and 4 or 5 to 4. It is the existence of an appropriate set of re-initializing functions (defined below as a cascade) which ensures eventual convergence to a steady solution.

284
VI.4.1. Definition: Cascade, Length of Cascade

A sequence \( e, \ldots, e \), where \( e : A \to A \) \((0 \leq i < 1)\) is a cascade over \( F \) iff:

the sequence of sets \( R \subseteq A \times A \) defined by:

\[
\begin{align*}
&i) \quad R = A \times A ; \\
&0 \quad i \\
&ii) \quad R = \text{Cyc} \left( \{ \langle e(a), b \rangle : \langle a, b \rangle \in R \} \right), 0 \leq i < 1 ; \\
&i+1 \quad i \quad i \\
\end{align*}
\]

satisfies:

\[
\begin{align*}
&iii) \quad \langle a, a \rangle \in R \Rightarrow \langle e(a), a \rangle \in R ; \\
&i \quad i \quad i \\
&iv) \quad R \subseteq D . \\
&1 \\
\end{align*}
\]

1 is said to be the length of the cascade.

These conditions are the minimum required to ensure Algorithm 5.1 converges. However, it will be shown below that when searching for a cascade for a particular set \( F \) not all possible functions \( e_i \) need be considered.

VI.4.1.1. Definition \( E \)

\[
E = \{ \langle e(a), b \rangle : \langle a, b \rangle \in R \} \\
\]

(\( \text{Note that } R = \text{Cyc}(E) \).)
VI.4.1.2. Theorem

All the $R_i$, $0 \leq i \leq 1$ are semi-reflexive.

Proof:

This is trivially true for $R_0$. Assume $R_i$ is semi-reflexive and consider $\langle c, b \rangle \in E_i$

$\implies \exists a, c = e(a)$ and $\langle a, b \rangle \in R_i$

$\implies \langle a, a \rangle \in R_i$ (as $R$ is semi-reflexive)

$\implies \langle e(a), a \rangle \in R_i$ (from 4.1(iii))

$\implies \langle a, e(a) \rangle \in R_i$

$\implies \langle e(a), e(a) \rangle \in E_i$.

That is $E_i$ is semi-reflexive and by Theorem 2.3.3 so is $R_i^{i+1}$.

VI.4.1.3. Theorem

$R_i^{i+1} \subseteq R_i$.

Proof:

$\langle a, b \rangle \in R_i \implies \langle a, a \rangle \in R_i$ (as $R$ is semi-reflexive)

$\implies \langle e(a), a \rangle \in R_i \implies \langle e(a), b \rangle \in R_i$

$\implies E_i \subseteq R_i$.

But $C(R_i) \subseteq R_i$ and $R$ is symmetric and transitive so $R_i^{i+1} = Cyc(E_i) \subseteq R_i$.

286
VI.4.2. Construction of Cascades

Consider now the problem of selecting some $e_i$ given $R_i$. From above it is known that any $e_i$ satisfying 4.1(iii) will generate an $R_{i+1}$ no larger than $R_i$. The problem then is to generate some $R_{i+1}$ which is a proper subset of $R_i$. It is therefore only worthwhile to consider those $e_i$ which minimize $E_i$. Consider for example the set $R_i$ of Fig. 3a. This divides A naturally into four classes, \{1\}, \{2, 3\}, \{4, 5\} and \{0\}. These classes will be denoted by \{1\} etc. in the notation introduced below. (0 is not paired with any other element and as will be seen below $e_i(0)$ is a "don't care" value which does not affect $R_{i+1}$.) Each of the members of a particular class must be mapped onto some member of the same class by $e_i$ (in order to satisfy 4.1(iii)). 4.2.1 and 4.2.2 establish that $e_i$ will generate a minimal $E_{i+1}$ if all members of a class are mapped to a single element of the class. In the example of Fig. 3 there are four distinct functions which can be constructed this way.

One consequence of this is that $e_o$ need only be a constant as $R_o = A \times A$ has only a single class (the whole of A).

In the following two lemmas I first show that the value of $e_i$ on classes such as \{0\} in Fig. 3 have no effect on the values of $R_{i+1}$ and that the smaller the number of values that $e_i$ maps onto the smaller will be the resulting $R_{i+1}$. Finally it is shown that the smallest possible $R_{i+1}$ can be obtained by mapping each class onto a single value.

The definitions below distinguish two different functions $e_i$ and $e'_i$ which are used in the next two lemmas.
For $e : A \rightarrow A$ and $e' : A \rightarrow A$

$$E = \{<e(a), b> : <a, b> \in R \}, R = \text{Cyc}(E)$$

$$E' = \{<e'(a), b> : <a, b> \in R' \}, R' = \text{Cyc}(E')$$

VI.4.2.1. Lemma

If $\forall a <a, a> \in R \Rightarrow e(a) = e'(a)$ then

$$R = R'$$

Proof:

$$<e(a), b> \in R \Rightarrow <a, b> \in R$$

$$\Rightarrow <a, a> \in R$$

$$\Rightarrow e(a) = e'(a)$$

$$\Rightarrow <e(a), b> \in R'$$

$$\Rightarrow R \subseteq R'$$

Similarly $R' \subseteq R$ so $R = R'$.

VI.4.2.2. Definition of $[a]$

$$[a] = \{b : <a, b> \in R \}$$
VI.4.2.3. Lemma

$$\forall a \in ([a]) \subseteq e([a]) \Rightarrow R \subseteq R'. \quad i \quad i \quad i+1 \quad i+1$$

Proof:

From definition 4.1.1 $$E \subseteq E' \quad i \quad i$$

which implies $$R \subseteq R' \quad i+1 \quad i+1$$

End of Lemma

I will now establish that it is possible to assign all the members of \([a]\) to the same value in a way which is consistent and which satisfies 4.1(iii). Because of the result above this will generate a minimal set \(R_{i+1}\).

VI.4.2.4. Theorem

$$\forall a, <a, a> \in R \text{ let } e([a]) = \{c\} \text{ for some } c \in [a] \quad i \quad i \quad i$$

then \(e\) satisfies 4.1(iii).

Proof:

From its definition \([a]\) divides \(A\) into equivalence classes.

That is either \([a] = [b]\) or \([a] \cap [b] = \emptyset\). So one can consistently map an entire class onto a single value.

That \(<c, a> \in R_i\), and so 4.1(iii) is satisfied, follows directly from 4.2.1 and the definition of \(R_i\) in 4.1(i) and (ii).

End of Theorem

For a particular set \(F\) the results above lead to a very simple if tedious process for enumerating all the sequences \(\{e_i\}\) which are candidates for cascades. (The results in section 8 were obtained using
VI.5. RELAXATION ALGORITHM

The algorithm below takes some e.s. and generates a succession of evaluations for it until a consistent one is obtained. The initial evaluation is created by assigning $e_0$ to each node. Some e.m., say $H$, is applied repeatedly to the evaluation. Eventually the pairs of successive values for each variable will be members of $R_i$ (by Theorem 3.3). Then $e_1$ is applied to the value of each variable. Again $H$ is applied until all successive pairs lie in $R_2$ and $e_2$ is applied and so on until two successive evaluations have pairs entirely in $R_k$. These two evaluations must then be equal and consistent. Theorem 5.2 shows that the conditions of 4.1 are sufficient to ensure convergence in a finite number of steps.

VI.5.1. Algorithm

Given an e.m. $H$, a cascade $e_i : 0 \leq i < 1$, and any evaluation $V$ then the following algorithm is defined.

Calculate $V_0, V_1, \ldots, V_l$ by applying the process below repeatedly,

\[
\begin{align*}
V_{i+1} &= V_i + H \cdot V_i \\
V_l &= \text{the consistent evaluation sought.}
\end{align*}
\]

Calculate $V_{i+1}$ from $V_i$ as follows:

\[
\begin{array}{c|c}
\text{let } W &= e(V_i) \\
0 & i & i
\end{array}
\]

290
\[ W_{j+1} = H(W_j^i), \quad j \geq 0 \]

then calculate \( W \) and each \( W_j^i \) until

\[ W_{j-1}^i, \quad W \subseteq R \]

and let \( V_{i+1}^i = W_i \).

The appendix contains an example of the operation of this algorithm.

VI.5.2. **Theorem - Algorithm Terminates Correctly**

Algorithm 5.1 terminates and upon termination \( V_x \) is consistent.

Proof:

Before embarking on the main body of the proof two lemma's will
be proven.

VI.5.2.1. Lemma

\[ W_{i-1}^i, \quad W \subseteq E \implies \exists m \text{ such that} \]

\[ W_{m-1}^i, \quad W \subseteq R \], \( 0 \leq i < 1 \).

Proof:

\[ R = \text{Cyc}(E) \] so the result follows directly from 3.3.

\[ V_{i+1}^i = W_i \]
VI.5.2.2. Lemma

If \( \frac{\scriptstyle i}{\scriptstyle m-1} \frac{\scriptstyle i}{\scriptstyle m} \subseteq_{\scriptstyle \text{R}} \frac{\scriptstyle i+1}{\scriptstyle 0} \frac{\scriptstyle i+1}{\scriptstyle 1} \)

then \( \frac{\scriptstyle W}{\scriptstyle 0} \frac{\scriptstyle W}{\scriptstyle 1} \subseteq_{\scriptstyle \text{E}} \frac{\scriptstyle 0}{\scriptstyle i} \frac{\scriptstyle 1}{\scriptstyle i} \), \( 0 \leq i < 1 \).

Proof:

Consider the diagram:

\[
\begin{array}{c}
\scriptstyle i & \scriptstyle H & \scriptstyle i & \scriptstyle H & \scriptstyle i \\
W & \rightarrow & W & \rightarrow & W \\
\scriptstyle m-1 & \scriptstyle m & \scriptstyle m+1 & \scriptstyle 1 & \\
\scriptstyle l & \scriptstyle i & \scriptstyle e & \scriptstyle i+1 & \\
\scriptstyle W & \scriptstyle W & \scriptstyle W & \scriptstyle W & \\
\scriptstyle 0 & \scriptstyle 1 & \scriptstyle 1 & \scriptstyle 1 & \\
\end{array}
\]

Each value \( W \upharpoonright (N), (N \in \mathbb{N}) \) is of the form \( e_i(a) \) so using \( 0 \).

4.1(iii) it is sufficient to show that

\[
\begin{array}{c}
\scriptstyle i+1 & \scriptstyle i+1 \\
W & \rightarrow W & \rightarrow W \\
\scriptstyle 0 & \scriptstyle 1 & \scriptstyle 1 \\
\end{array}
\]

This will be done by showing that each of the pairs of evaluations joined by dotted lines in the diagram satisfy this relation.

From Theorem 3.3 \( \frac{\scriptstyle i}{\scriptstyle m} \frac{\scriptstyle i}{\scriptstyle m+1} \subseteq_{\scriptstyle \text{R}} \frac{\scriptstyle 1}{\scriptstyle i} \).

(1)

To show that \( \frac{\scriptstyle i+1}{\scriptstyle m} \frac{\scriptstyle i+1}{\scriptstyle 0} \subseteq_{\scriptstyle \text{R}} \frac{\scriptstyle 1}{\scriptstyle i} \)

it is necessary to show for each \( N \in \mathbb{N} \) that

\[
\begin{array}{c}
\scriptstyle i & \scriptstyle i+1 \\
\langle W(N), \frac{\scriptstyle i}{\scriptstyle m} \rangle \subseteq_{\scriptstyle \text{R}} \langle W(N), \frac{\scriptstyle i}{\scriptstyle m} \rangle \\
\scriptstyle m & \scriptstyle 0 & \scriptstyle m & \scriptstyle i & \scriptstyle m & \scriptstyle i \\
\end{array}
\]

From 4.1(iii) it is sufficient to show that

292
\[ \langle W(N), W(N) \rangle \in R, \forall N \in N. \]
\[ \text{From (1)} \]
\[ W \upharpoonright W \subseteq R \]
\[ m \quad m+1 \quad i \]

That is \[ \langle W(N), W(N) \rangle \in R, \forall N \in N. \]
\[ m \quad m+1 \quad i \]

4.1.3 implies \( R_\zeta \) is semi-reflexive so
\[ \langle W(N), W(N) \rangle \in R \]
\[ m \quad m \quad i \]

\[ \text{so } W \upharpoonright W \subseteq R. \]
\[ m \quad 0 \quad i \]

(2)

Theorem 3.3 can be applied to (2) giving
\[ W \upharpoonright W \subseteq R. \]
\[ m+1 \quad 1 \quad i \]

(3)

Recall that \( R_\zeta \) generates a symmetric and transitive relation on the evaluations so that (1), (2) and (3) imply
\[ W \upharpoonright W \subseteq R. \]
\[ 0 \quad 1 \quad i \]

Proof: (of 5.2 continued)
\[ W \upharpoonright W \subseteq E \]
\[ 0 \quad 0 \quad 1 \quad 0 \]

is trivially satisfied, so by 5.2.2 \( V_\zeta \) will be constructed in a finite number of steps.

By applying 5.2.1 and 5.2.2 in turn it can be seen that each \( V_{i+1} \) can be constructed in a finite number of steps from \( V_\zeta \). So by induction on \( i \), \( V_\zeta \) can be constructed in a finite number of steps. I will now show that \( V_\zeta \) is consistent.
Also, \( V = W \) and \( R \subseteq D \).

So

\[
W^{m-1} W^m = V
\]

and

\[
V = H(V), \text{ that is } V \text{ is consistent.}
\]
where: \[ L_i(V):N \mapsto \begin{cases} I(V)(N) & \text{if } j = i \\ V(N) & \text{if } j \neq i \end{cases} \]

(that is \( L_i \) recalculates the i'th node).

then: \[ L = \prod_{n} L_{n-1}(\ldots(L_1)\ldots) \]

VI.6.2. Theorem

\( L \) is an evaluation map

Proof:

(i) To show \( V; W \subseteq C(S)^* \Rightarrow L(V); L(W) \subseteq C(S)^* \)

it is sufficient to show this for each

\[ L_i, 1 \leq i \leq n. \]

Consider \( \langle L_i(V)(N), L_i(W)(N) \rangle \).

If \( j \neq i \) then this is equal to

\[ \langle V(N), W(N) \rangle \]

and the result holds.

If \( j = i \) then this is equal to

\[ \langle I(V)(N), I(W)(N) \rangle \]

which from Theorem 2.3.6 is a member of \( C(S)^* \).

(ii) To show \( L(V) = V \) iff \( I(V) = V \)

it is sufficient to show

\[ L_i(V)(N) = V(N) \text{ iff } I_i(V)(N) = V(N) \]

(as \( L \) affects only the value of \( N \)).

This follows directly from the definition of \( L \).
VI.7. CONVERSE PROBLEM

It has been shown that every set $\mathcal{F}$ which has a cascade has a consistent evaluation for all equation sets using $\mathcal{F}$. It is not known if the converse of this is true. That is, does every set $\mathcal{F}$, all of whose equation sets have consistent evaluations, possess a cascade?

VI.8. EXAMPLES

This section shows that the existence of a cascade of length 1 is (nearly) equivalent to there being a partial order on the members of $A$ preserved by the functions of $\mathcal{F}$. Some examples of sets of functions which possess cascades will also be given.

VI.8.1. Cascades of Length 1

If there is a partial ordering, p.o., (a reflexive, anti-symmetric and transitive relation) on $A$ which is preserved by the functions of $\mathcal{F}$ and there is an element of $A$ which is a minimum in the ordering, then $\mathcal{F}$ will possess a cascade of length 1. Examples of this are:

i) $\mathcal{F} = \{\text{Min}, \text{Max}\}$ for some linear ordering of $A$;

ii) $\mathcal{F} = \{\text{Union}, \text{Intersection}\}$ where $A = 2^C$ for some set $C$, and the ordering is set inclusion.

The existence of a cascade of length 1 does not imply the existence of such an ordering but it does imply a slightly weaker condition of an ordering on a particular subset of $A$. 

296
VI.8.1.1. Theorem:

If there is a p.o. \( \leq \) on \( A \) such that,

(i) \( \exists z : z \leq a, \forall a \in A \)

and (ii) \( a \leq b \), \( 1 \leq i \leq \sigma(F) \)

\[ \Rightarrow F(a \ldots a) \leq F(b \ldots b) \text{ for all } F \in F \]

then \( F \) has a cascade of length 1.

Proof:

Let \( e : a \mapsto z, \forall a \in A \).

Let \( Q = \{ \langle a, b \rangle : a \leq b \} \),
then \( E \subseteq Q \) from 4.1(ii).

Also, from (ii) above \( \mathcal{L}(Q) = Q \)
and by the transitivity of \( \leq \), \( Q^* = Q \),
so \( \mathcal{L}(E^*) \subseteq Q \)
that is \( \text{Cyc}(E) = R \subseteq \{ \langle a, b \rangle : a \leq b, b \leq a \} \subseteq D \).

So \( e_0 \) is a cascade of length 1.

VI.8.1.2. Theorem:

If \( F \) has a cascade of length 1 then

\( \exists B \subseteq A \) and a p.o., \( \leq \) on \( B \) such that:

(i) \( \exists z \in B, z \leq a, \forall a \in B \);

(ii) \( a \leq b \), \( a, b \in B, 1 \leq i \leq \sigma(F) \)

\[ \Rightarrow F(a \ldots a) \leq F(b \ldots b) \text{ for all } F \in F; \]

(iii) \( a \in B, 1 \leq i \leq \sigma(F) \),

\[ 1 \leq F(a \ldots a) \leq F(b \ldots b) \text{ for all } F \in F; \]

\[ 1 \leq F(a \ldots a) \leq F(b \ldots b) \text{ for all } F \in F; \]

\[ 1 \leq F(a \ldots a) \leq F(b \ldots b) \text{ for all } F \in F; \]
\[ \Rightarrow F(a \ldots a) \in B \quad \text{for all } F \in E. \]

Proof:

Let \( e_0 \) be the cascade of length 1, then

let \[ B = \{ a: \langle a, a \rangle \in \text{Cyc}(E) \} \],

\[ z = e(a), \]

and for \( a, b \in B, \ a \leq b \iff \langle a, b \rangle \in \text{C}(E)^* \).

Verify each condition in turn.

(i) Assume \( \exists a \in B, \ a \leq z \) and \( a \neq z \)

then \( \langle a, z \rangle \in \text{C}(E)^* \).

But \( \langle z, a \rangle \in E \)

so this implies \( \langle z, a \rangle \in \text{C}(E)^* \)

and \( \langle a, z \rangle \in \text{Cyc}(E) \).

But \( a = z \) and \( \text{Cyc}(E) \subseteq D \) which is a contradiction.

(ii) This is a direct consequence of Theorem 2.3.6.

(iii) \( \langle a, a \rangle \in \text{Cyc}(E) \) iff \( \langle a, a \rangle \in \text{C}(E)^* \)

and the condition is thus just a special case of (ii).

I will now verify that \( \leq \) is a p.o. \( \text{Cyc}(E_0) \) is transitive and (over \( B \)) reflexive. So only anti-symmetry need be checked; that is,

\[ a \leq b \quad \text{and} \quad b \leq a \Rightarrow b = a. \]

Now if \( a \leq b \) then \( \langle a, b \rangle \in \text{C}(E)^* \)

and if \( b \leq a \) then \( \langle b, a \rangle \in \text{C}(E)^* \).

298
which implies \( \langle a, b \rangle \in \text{Cyc}(E) \leq D \)

which implies \( a = b \).

**End of Proof**

The theorem above generates the following partial order for the set of functions in Fig. 5:

![Graph](image)

and \( B = A \{0,1,2,3\} \).

**VI.8.1.3. Comment**

From condition (iii) above, the set \( B \) must be closed under the application of functions from \( F \). So in those cases where the functions of \( F \) are sufficiently rich to generate all the members of \( A \), (this is true of the two examples in section 8), then \( B \) will equal \( A \) and the last two theorems are the converse of one another.

It is readily seen that these theorems imply the existence of an infinite number of sets \( F \) possessing a cascade. For, given a set \( A \), a p.o. on \( A \) can be constructed and, from it, functions which preserve this ordering.

There are sets \( F \) which possess cascades and lie outside this family (see the examples of 8.2 which possess cascades of length 2). However, even in these examples, the set \( A \), in some sense, 'nearly' has a p.o. which is preserved by the members of \( F \).
Further Examples of Cascades

The set of functions which motivated this work (see introduction) had associated the following sets $A$ and $E$. $A$ was all subsets of $\{r,n,p\}$ excluding the empty set. (The letters $r,n,p$ stand for different "reward" values: reward, neutral and pain respectively.)

The functions of $E$ were: the constant functions $\{r\}, \{n\}$ and $\{p\}$, and the dyadic functions, one set union, the other a maximum of a linear ordering on $A$. Fig. 4 summarises this and gives the ordering on $A$. A search for a cascade for this system yielded one of length 2, where $e_0 : a \mapsto \{r,n\}$ and $e_1$ is given in the figure (the sets of $A$ have been abbreviated there to the numbers $0, 1, ..., 6$ for notational ease). In the search the classes of $R_1$ were obtained for each possible value of $e_0$; these are also listed in Fig. 4.

As well as this system there are some five other 'reasonable' sets of functions which differ from this only in the ordering on $A$. (These are obtained by varying the relative order of $\{r,p\}$, $\{n\}$ and $\{r,p,n\}$.) Each of these has a cascade of length 1 or 2.

A simplified set of functions was also used in the original application. In it, $A = \{ \{r\}, \{n\}, \{r,p\}, \{p\} \}$ and $E$ contained a maximum function and a modified union (see Fig. 5). This simplified system has a cascade of length 1, $e_0 : a \mapsto \{n\}$.
\( \{ r, n, p \} \)
\[ A = 2 - \{ \emptyset \} \]
\[ F = \{ \{ r \}, \{ n \}, \{ p \}, \text{Union}, \text{Max} \} \]
where Max is taken from the ordering:
\[ \{ r \} > \{ r, n \} > \{ r, p \} > \{ n \} > \{ r, p, n \} > \{ p, n \} > \{ p \} \]
\[ 6 \quad 5 \quad 4 \quad 3 \quad 2 \quad 1 \quad 0 \]
(For clarity, these numbers, rather than subsets of \( \{ r, p, n \} \), are used below)

<table>
<thead>
<tr>
<th>Classes of R for different Possible Values of e</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e )</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

Cascade of Length 2

\[ e = 5 \]
\[ a \quad e (a) \]
\[ 0 \quad 1 \]
\[ 1 \quad 2 \]
\[ 2 \quad 3 \]
\[ 3 \quad 4 \]
\[ 4 \quad 5 \]
\[ 5 \quad 6 \]

Fig. 4 Example of Cascade of Length 2
$A = \{ \{r\}, \{n\}, \{r,p\} \{p\} \}$
$F = \{ \{r\}, \{n\}, \{p\}, \text{Union, Max} \}$

where Max is taken from the ordering:
$\{r\} > \{r,p\} > \{n\} > \{p\}$
$3 \quad 2 \quad 1 \quad 0$

and Union is given by the table:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

that is: $\{n\} \cup \{r\} = \{r\}$
$\{n\} \cup \{p\} = \{p\}$
$\{n\} \cup \{r,p\} = \{r,p\}$

The Classes of $R$ for different possible values of $e$

<table>
<thead>
<tr>
<th></th>
<th>classes of $R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>${0,1,2,3}$</td>
</tr>
<tr>
<td>1</td>
<td>${0}$ ${1}$ ${2}$ ${3}$</td>
</tr>
<tr>
<td>2</td>
<td>${0,1,2,3}$</td>
</tr>
<tr>
<td>3</td>
<td>${0}$ ${1,3}$ ${2}$</td>
</tr>
</tbody>
</table>

so $e = 1$ provides a cascade of length 1.

Fig. 5 Example of Set of Functions with Cascade of Length 1
VI.A.1. Example of Operation of Relaxation Algorithm

Fig. 6 gives an example of a set of functions and the calculation of cascades from them. The functions have been chosen because they are the simplest example of a set $F$ whose shortest cascade is of length 2. Fig. 6a shows the calculation of $R_\phi$ if $e_\phi = 2$ (recall that $e_\phi$ is always a constant function). This gives a set of pairs $R_\phi$ with two classes $\{0\}$ and $\{1,2\}$. The other two possibilities for $e_\phi (e_\phi = 1, e_\phi = 0)$ lead to an identical $R$. Given this set $R_\phi$ then there are two possible functions for $e_\phi$ (depending on whether 1 or 2 is chosen as the representative for the class $\{1,2\}$). Fig. 7 gives an example of an equation set represented by a digraph. This shows the successive values taken by the nodes when Algorithm 5.1 is used to obtain a solution. ($I$ is used as the evaluation map.)
A = \{0, 1, 2\}
F = \{M, M', 0, 1, 2\} where 0, 1, 2 are the appropriate constant 

functions and \( M : A \rightarrow A \), \( M : A \rightarrow A \) are defined below.

<table>
<thead>
<tr>
<th>M</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

\[ E = \{\langle e(a), b \rangle : \langle a, b \rangle \} \quad (\text{recall } R = A \times A) \]

<table>
<thead>
<tr>
<th>E</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

New Pairs generated by \( C(E) \) (see 5.1(ii))

<table>
<thead>
<tr>
<th>F</th>
<th>( \langle a, b \rangle )</th>
<th>( \langle a', b' \rangle )</th>
<th>( \langle f(a, a'), f(b, b') \rangle )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

\( \text{Cyc}(E_0) = C(E_0)^* \)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

\( R = \text{Cyc}(E) \)

Classes of \( R \):

\[ \{0\}, \{1, 2\} \]

Possible functions \( e : \)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Repeat application of $I$ until successive pairs lie in $R_4$ (see Fig. 6). Initial value $e_0 = 2$.

All the pairs $<0,0>$, $<1,2>$ lie in $R_4$, so $e$ (0 $\rightarrow$ 0,1,2 $\rightarrow$ 1) is applied. Then $I$ is repeated until a solution is reached. (This occurs on the first step.)

Fig. 7 Example of Application of Algorithm 5.1.
VII.1. IMPLEMENTATION OF A.I. SYSTEMS

One of the aims when designing the FLM has been to provide a system which could be used in artificial intelligence applications. The FLM is able to accept inputs in a simple and unstructured format. Also, its memory is being continually updated as it builds a simple 'model' of these inputs. For example, it is shown in Chapter II that a significant class of finite automata can be simulated after observing only the inputs to and outputs from them. The FLM is thus suitable for inclusion in automatic systems requiring adaption and learning.

In Chapter II it has been shown, by using echo actions, that the FLM can perform any specifiable computation (i.e. it can simulate any finite automaton). Perhaps more importantly, Chapters II and III show how it can make use of techniques such as subroutines to reduce the complexity of its computations. In particular, Chapter III shows how these techniques can be extended by using systems containing more than one FLM.

Chapter IV has shown that the FLM can be implemented efficiently on current digital computers using memories of millions of bytes at execution speeds independent of memory size.

Together, these results provide a firm foundation for building actual systems of significance to artificial intelligence. The development of such systems will require continued improvement in techniques for structuring and programming the FLM. The combination of the FLM into novel multiple systems promises to provide the needed
improvements in power and applicability.

VII.2. STABILITY

Another aim when designing the FLM has been to provide a precisely defined system whose properties can be analyzed. The precise description of the FLM in Chapter I contributes to this aim and provides a basis for the results of Chapters II and III. One important condition explored there is that if the environment generating the inputs to the FLM (or MFLM) is unchanging then the behaviour of the FLM itself should be unchanging. This condition allows the contents of the FLM memory to be examined and precise statements to be made about its future behaviour. This places the results of Chapters II and III about programming on a firm foundation because it is possible to determine whether or not an algorithm has in fact been programmed into the FLM memory. This condition of stability is thus in accord with the aim that the FLM should be well defined and amenable to analysis.

It is possible that there is some significant type of behaviour which is impossible in stable systems such as the FLM. It would be a very significant result if this could be demonstrated and one which would make any analysis of the resulting unstable systems more difficult.

In run mode the inputs to the FLM appear to it to be unchanging as the actions it executes are just those it predicts from its previous inputs. Using the definition in Chapter I of the FLM it is shown in Chapter II that the FLM is stable in run mode. This is done by showing that there exists a description of the memory which is both unchanging.
and sufficient to specify the course of any deterministic algorithm already programmed. This description, however, will be ambiguous about the course of any behaviour not yet programmed or incompletely programmed. So there is room for the FLM to alter and refine its behaviour within the constraints of this invariant description.

While the stability of the FLM follows almost incidentally from its best match structure rather more care is needed to ensure that a multiple FLM system is stable. This is illustrated by the example at the end of Chapter III of an 'invalid MFLM'. This employs a simple technique, used for example in PURR-PUSS, to select actions from two FLM memories. Because of the interactions between the two memories which can 're-program' each other the total system is unstable in runmode. It is shown elsewhere in the Chapter that there exist at least two ways of ensuring that a multiple FLM is stable. The first of these imposes a number of conditions on the global prediction function which synthesizes an overall prediction from the individual FLM predictions. In the second, individual memories are updated only when their predictions include the action actually chosen.

When the FLM is in program mode it is not stable in the sense above as its behaviour can be altered by the programmer. It is stable, however, in the weaker sense that any trace which has occurred is always retained (forgetting aside). So, as was noted in Chapter I, previous behaviour can never be deleted only circumvented. Again it is not clear if this significantly restricts the abilities of the FLM.
VII.3. PROGRAMMING

As a result of the precise definition of the FLM it has been possible to say which programming techniques can be used with it and which cannot. For example, it is shown in both Chapters II and III that it is possible to use programming techniques on the FLM similar to constructs available in high level computer languages. The importance of these techniques is that quite short programming sequences can be used to impress behaviour which would otherwise require the repetition of many combinations of similar cases. For example it is possible to achieve in effect similar to calling a subroutine. It is shown that provided the subroutine is shorter than the order of the FLM then a single FLM computer can make use of the technique. Further, by making use of echo actions and multiple FLM memories, the size of the subroutine is unrestricted and the amount of input needed to program it is further reduced.

It is shown, however, that it is not possible for any multiple FLM system to simulate negation. While it is possible to execute an action completely ignoring a recent input pattern it is not possible to execute one action for a particular pattern and a different action for all other possible patterns. Each case of 'all other patterns' must be programmed separately. Note that to obtain this result one must assume that there is no pattern which is present precisely when the original is not. Consider, for example, a sensor which can see the patterns red, blue, green, yellow, magenta etc. To execute an action whenever red is not present it is necessary to program the action for each of the separate cases blue, green, yellow, magenta... However if there is another sensor present which returns just the patterns 'red' and 'not
red’ then it is of course possible to execute an action when red is not present using the pattern 'not red'.

The proof that multiple FLMs cannot use negation is dependent on the best matching retrieval algorithm of the FLM and the requirement that multiple FLMs be stable. It is possible that other systems using storage and retrieval algorithms slightly different from those of the FLM could successfully handle negation.

VII.4. IMPLEMENTATION

Because of the best matching technique used in the FLM it is possible to implement it efficiently. This is in contrast to the expectations of workers in A.I. who have claimed that best match searches of large memories must involve explicitly searching large parts of the memory (Minsky and Papert, 1969), (Uhr, 1973). These opinions are emphasized by the results of Chapter V where it is shown that finding nearest neighbours in Euclidean space rapidly becomes more expensive as the dimension of the space increases. However, Chapter IV shows that the FLM can be implemented so that the time for both storage into it and retrieval from it are almost independent of the memory size and proportional only to the order of the FLM.

Chapter IV demonstrates two implementations of the FLM suitable for serial computers. The first structures the memory as a tree so that storage and retrieval require only the traversal of a unique route from the root of the tree to its leaves. An actual implementation of such a tree structured memory, Felix, is demonstrated and is shown to provide a good balance of storage usage and execution times. In the second type
of implementation the tail of each trace forms a key which is hashed and used to retrieve the head of the trace. One problem with this technique is that it uses rather more space than the tree structured memory. This is alleviated by a new algorithm which enables a variable length hash key to be constructed. Another way of saving space is to truncate the hash key and only store part of it. This leads to the possibility that traces will be erroneously retrieved but an analysis shows that the probability of this can be kept very low.

A third implementation, Simple, is of interest because of its low storage requirements and the simplicity of its internal structure. This simplicity enables it to be implemented easily on parallel hardware. However the need for complete scans of the memory on each store and fetch make it unsuitable for serial computers.

VII.5. FORGETTING

When an FLM becomes full some exceptional action must be taken if it is to continue running. Chapter I introduces the notion of 'forgetting' where old traces are deleted from the memory to make way for new ones to be added. The idea is developed more fully in Chapter IV where two very simple algorithms for forgetting are investigated. The first of these selects traces at random to delete them; the second maintains an ordered list of traces and deletes the oldest trace on the list. It is shown analytically that these algorithms can be effective and in some situations approach the best possible forgetting where complete information is available about the frequency of occurrence of traces.
Chapter IV also shows how these algorithms can be implemented for each of the different memory structures developed in the chapter. In each case it is shown that forgetting can be accomplished in times comparable to the storage and retrieval of traces and using no, or a very small amount of, extra memory.

VII.6. NETWORKS

The FLM potentially contains much information about the future behaviour of the environment in which it is embedded. One way to use this information is to examine all possible future paths in the memory radiating from a particular trace. To do this effectively requires that it be possible to very quickly find each trace which can follow a particular trace. It is shown in Chapter IV that a network of pointers from each trace to its descendants can be incorporated naturally into a tree structured FLM memory. (The Felix implementation includes such a network.)

To use the information provided by such a net in selecting the next action it can be helpful to summarize it by assigning to each trace an estimate of its 'value' (for example its likelihood of leading to reward or pain). It is possible however for the calculations of such values to be unable to arrive at a consistent value for each trace because of loops and interdependencies within the network. Chapter VI gives an algorithm which ensures convergence to consistent evaluation. For those types of evaluation where the algorithm is applicable it is shown that it will always converge no matter what the structure of the network it is applied to.
In this thesis I have demonstrated an associative and impressible computer, the FLM, and systems composed of a number of FLMs. Using a precise definition of the FLM I have shown that it:

- can perform efficient associative retrieval on large memories as a result of its best matching algorithm;
- can simulate any finite automaton and can imitate a significant class of finite automata merely by observing their inputs and outputs;
- can stably maintain algorithms even while its memory is being updated and modified;
- can make use of programming techniques such as subroutines to enhance its power. However, it is unable to simulate negation.

A number of different implementations of the FLM have been demonstrated and analyzed. These:

- enable best match storage and retrieval to be done in times independent of the size of memory on serial computers;
- can be used to construct large memories resident on either fast random access memory or disc backing store;
- allow the FLM to keep executing and storing information with a full memory by forgetting old traces;
- allow efficient access to future paths through the memory by maintaining a network of pointers between traces in addition to the standard FLM memory structure;
- permit a very compact simple representation of the FLM memory suitable for use with parallel hardware.

An analysis of best matching in Euclidean space has been given showing
that its execution time rises exponentially with the dimension of the space in contrast to the FLM whose execution time is proportional to its order.

A new algorithm has been described for solving large simultaneous equations over a finite domain. This can be applied to networks in an FLM memory to obtain consistent evaluations of the 'value' of traces.

The FLM thus provides an example of an adaptive system which attains its power by a simple underlying algorithm combined with a large data base of experience. Such systems should prove to be significant in the future development of Artificial Intelligence.
I would first like to thank my supervisor, J.H. Andreae, for his tremendous enthusiasm and dedication which has been an inspiration throughout my work.

I would also like to thank my willing helpers with the typing, Beth Hobbs and Sue Graham and my wife, Dorothy Cleary, who has much reduced the level of grammatical errors and mis-spellings by careful checking and proof-reading.

I am indebted to my employers, PROGENI Systems Ltd., who have provided the computer and word processing facilities used in preparing this manuscript.

This work has been supported in part by the Defence Scientific Establishment of the New Zealand Ministry of Defence.
REFERENCES

AMBROZE, O and KNUTH, D.E. 1974

ANDREEAE, J.H. 1975

ANDREEAE, J.H. 1977
Thinking with the Teachable Machine. Academic Press.

ANDREEAE, J.H. and CLEARY, J.G. 1976

ARBIB, M.A. 1969

BEAVEN, P.A. and LEWIN, D.W. 1972

BIERMAN, A.W. 1972

BIERMAN, A.W. and FELDMAN, J.A. 1972
On the Synthesis of Finite-State Machines from Samples of their Behavior. IEEE Transactions on Computers C-21, June:592-597.

BURROUGHS CORP. 1974

CLEARY, J.G. 1973
CLEARY, J.G. 1974

CLEARY, J.G. 1975a

CLEARY, J.G. 1975b

CLEARY, J.G. 1979

CRICK, F.H.C. 1979
Thinking about the Brain. Scientific American 241(3), September:181-188.

FISHER, D.A. 1974

GAINES, B.R. 1976

HEADS, W.R. 1975a

HEADS, W.R. 1975b

HENRICI, P. 1964
HILDEBRAND, F.B. 1974

JENSEN, K. and WIRTH, N. 1978

KNUTH, D.E. 1973a
Fundamental Algorithms. The Art of Computer Programming. 2d ed. 1, Addison-Wesley.

KNUTH, D.E. 1973b

KNUTH, D.E. 1973c
Sorting and Searching. The Art of Computer Programming. 2d ed. 3, Addison-Wesley.

KOCHEN, M. 1974

MACDONALD, B.A. 1979

MAN-MACHINE STUDIES. 1971-1979
Andresen, J.H. Ed. Christchurch, University of Canterbury. (Reports UC-DSE/1 to UC-DSE/75).

MINSKY, M. and PAPERT, S. 1969

NEWELL, A. and SIMON, H.A. 1972

ORGANICK, E.I. 1973
PALFI, A.F. 1977

SLAGLE, J.R. 1971

SUPPES, P. 1969a

SUPPES, P. 1969b

UHR, L. 1973
Pattern Recognition Learning and Thought. Prentice-Hall.

WATERMAN, D.A. 1970

WITTEN, I.H. 1979