

THE UNIVERSITY OF HULL

ROBOTIC WORKCELL ANALYSIS AND OBJECT LEVEL PROGRAMMING

being a Thesis submitted for the Degree of Doctor of Philosophy

in the University of Hull

by

Gareth John Monkman, BA, BSc, MSc.

June 1990.

## ACKNOWLEDGEMENTS

I would like to acknowledge the help and assistance of my supervisor and mentor Prof. Paul Taylor, Dr. Gaynor Taylor for her help with the more mathematical aspects of this work and all other members of the robotics research group involved in error recovery research.

ROBOTIC WORKCELL ANALYSIS AND OBJECT LEVEL PROGRAMMING.

## ABSTRACT

For many years robots have been programmed at manipulator or joint level without any real thought to the implementation of sensing until errors occur during program execution. For the control of complex, or multiple robot workcells, programming must be carried out at a higher level, taking into account the possibility of error occurrence. This requires the integration of decision information based on sensory data.

Aspects of robotic workcell control are explored during this work with the object of integrating the results of sensor outputs to facilitate error recovery for the purposes of achieving completely autonomous operation.

Network theory is used for the development of analysis techniques based on stochastic data. Object level programming is implemented using Markov chain theory to provide fully sensor integrated robot workcell control.

## CONTENTS

1.	INTRODUCTION.	1
2.	NETWORKS.	5
2.1	Flowcharts and Stategraphs.	5
2.2	Signal Flowgraphs.	7
2.2.1	Isomorphisms and Homomorphisms.	9
2.2.2	Continuous Sensing Representations.	11
2.3	Event Graphs.	12
2.4	Petri Nets.	13
2.4.1	The Token Machine.	14
2.4.2	Stochastic and Time Petri nets.	16
2.5	PERT and GERT.	17
2.6	Concurrency.	18
2.7	Logistics and Planning.	20
2.8	Summary.	22

3.	NETWORK ANALYSIS	23
3.1	Flowgraphs, Costs and Probabilities.	24
3.2	Independance and the Markov Property.	26
3.3	The Z Transform.	28
3.4	A Practical Example.	30
3.5	Isochronic Curves.	32
3.6	Flowgraphs with Continuous Sensing.	36
	3.6.1 Tolls and Probabilities.	37
	3.6.2 Functions and Distributions.	40
3.7	The Laplace Transform.	42
	3.7.1 Topology.	43
	3.7.2 Variance.	45
3.8	AND Input Analysis.	48
	3.8.1 Logic Functions and Distributions.	50
3.9	Summary.	52
4.	MATRIX ANALYSIS	54
4.1	Adjacency Matrices.	54
	4.1.1 Boolean Adjacency Matrices.	56
	4.1.2 Transition Matrices.	56
4.2	The Realization Matrix.	58
4.3	The Incidence Matrix.	60

4.4	Matrix Analysis.	61
4.4.1	Flows.	61
4.4.2	Sensor Implementation.	63
4.4.3	Matrix Formulation.	65
4.4.4	Matrix Derivatives.	66
4.5	Markov Processes.	70
4.5.1	The Limiting Matrix.	70
4.5.2	Ergodic Chains.	74
4.5.3	Regular Chains.	75
4.5.4	Variance.	78
4.5.5	The Reverse Markov Chain.	79
4.6	Summary.	80
5.	INTERACTIVE SENSORY SYSTEMS	81
5.1	Robot Programming Levels	83
5.2	Sense Parameters.	87
5.3	Interrupts.	89
5.4	Sensor Driven Programming.	92
5.5	Object Driven Programming.	93
5.6	Summary.	95
6.	PROGRAMMING AND SIMULATION	96
6.1	Conventional Structures.	99

6.2	Simulation and Modelling.	101
6.2.1	Object and Task Level Programming.	102
6.2.2	Readability.	108
6.2.3	Extensions to General Programming.	110
6.2.4	Buffering and Partitioning.	113
6.3	Aspects of Parallel Processing.	120
6.4	Some Thoughts on Task Level Programming.	123
6.5	Summary.	126
7.	OVERALL STRUCTURE AND IMPLEMENTATION.	128
7.1	The New Model.	128
7.2	Selected Case Studies.	130
7.2.1	Pick and Place Model.	130
7.2.2	An Intelligent Robot workcell.	138
7.3	Summary.	142
8.	EVALUATION	143
8.1	Markov Simulator.	143
8.1.1	Modula-2 Procedures.	143
8.1.2	Program Operation and User Guide.	147



8.2	A Comparative Study.	159
8.2.1	Queueing Simulators.	159
8.2.2	Network and Flowgraph Methods.	160
8.2.3	Petri Net Simulation Packages.	161
8.2.4	Geometrical Robot Animation Systems.	161
8.3	Further Research.	163
9.	CONCLUSIONS.	165
	REFERENCES.	168
	APPENDIX.	175
A.	Queueing Theory Nomenclature.	176
B.	Digraph Nomenclature.	178
C.	Markov Process Nomenclature.	180
D.	Statistical Distributions.	181
E.	Program Listings.	183

## 1. INTRODUCTION.

The object of this work is to formulate a selection of techniques by which a computer system can accept a graphical representation of the actions of a robot (or other automated) workcell to form the basis of a complete planning, simulation and object level programming package.

The procedure for any planning method basically follows the manner of figure 1.1. Often however, this is achieved using three different techniques on three separate systems. The methods considered here form the basis of a strategy on which a complete workcell can be planned, simulated and executed.

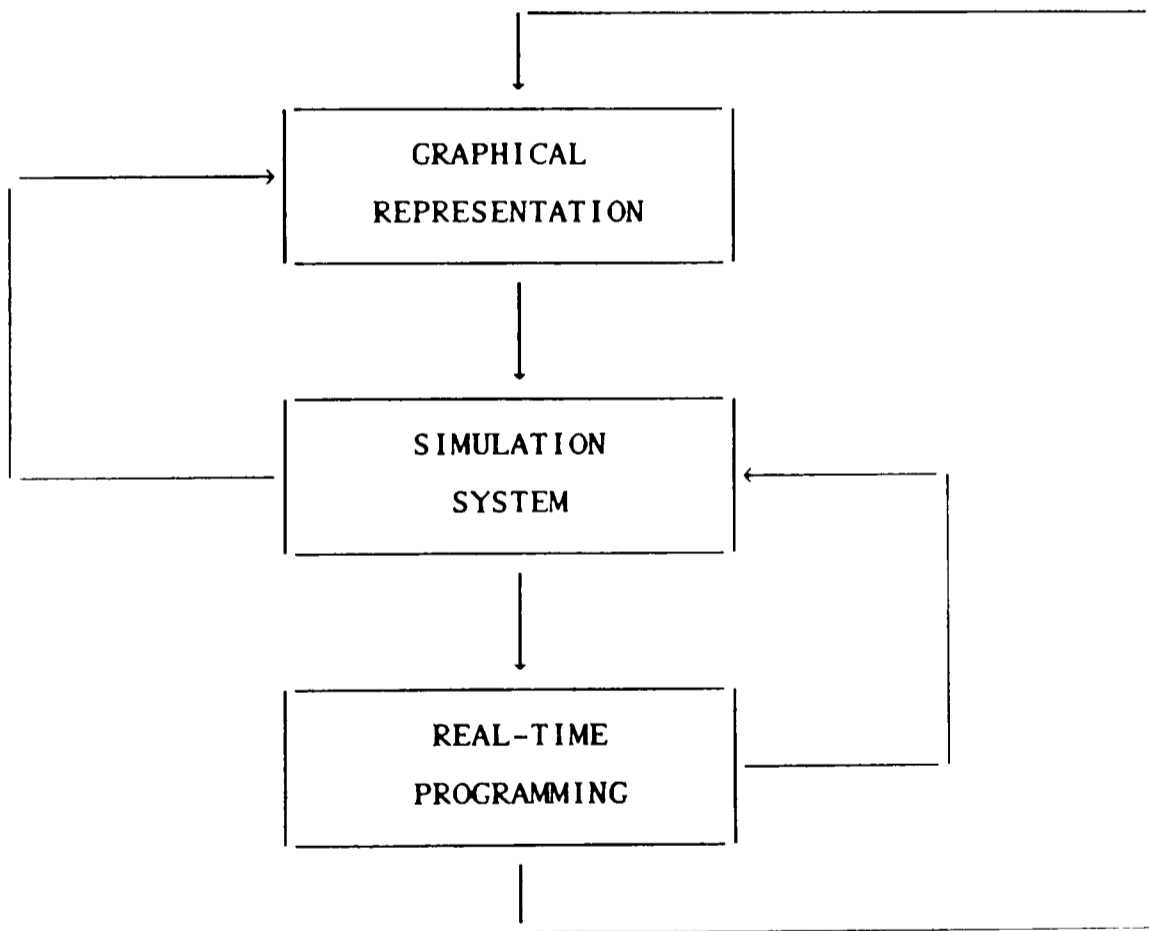


Figure 1.1 - Schematic for a complete system

The research is mainly concerned with the investigation of robot workcell planning, scheduling, control, programming and real-time execution. Particular attention is paid to aspects of error recovery. The physical feasibility of assembly tasks and their corresponding planning and optimisation has been studied in far greater depth elsewhere [Levi & Loeffler, 1986 ], [Frommherz & Hornberger, 1988] and will therefore not be pursued here.

Many reasons exist as to why error recovery in robot programming is gaining popularity. Foremost is the increasing need to have workcells operating completely autonomously for long periods with minimum operator intervention when things do go wrong. The finite degree to which engineering improvements can enhance a piece of hardware's operating efficiency, cost effectively, before a state of diminishing returns is experienced is becoming more relevant as automation is coerced into tackling tasks of ever increasing difficulty. One must never forget the old adage "the price for perfect reliability is infinite cost". This is of particular relevance to the handling of non-rigid materials where uncertainties in the objects physical parameters are considerably larger than those of the robot or end-effector. Such materials inhibit the ability of the designer to "engineer out" the possibility of error occurrence. Much of the research work carried out over the past decade at the University of Hull concerns the manipulation of non rigid objects [Taylor et al, 1990], and it is on the handling techniques developed for these and similar tasks that many of the examples provided in this work are based.

With regard to error recovery in robot workcells, the vast majority of research work done to date appears to be in the area of geometrical errors occurring due to some part defect, robot inaccuracy [Lee et al, 1984], or geometrical uncertainty in the robot environment [Donald, 1986]. However, the choice of sensor, its positioning and the programming which interacts with the sensors is usually implemented in some arbitrary manner. The programming of error recovery mechanisms appears to have been approached in two ways: From the sensor point of view [Milovanovic 1987] and the artificial intelligence angle [Gini 1987], [Kumpel & Rosa 1987].

Most discrete time simulation systems to date rely on the execution of a sequence of activities in the same manner in which the actions being simulated would be run by a real time process control program. This differs from the usual mathematical techniques used for the analysis of continuous time control systems where the model is treated as a complete algebraic entity rather than a set of discrete events [Denham, 1989]. Sequential event execution is ideal for real time task and object level programming, however it makes a very cumbersome simulation tool relying on much statistical analysis. This work attempts to address the use of mathematical techniques for discrete time network simulation, whilst retaining the sequential event execution methodology for real time robot programming.

In addition, an analysis system based on established network notations will be demonstrated, leading to a more usable method of object and task level robot programming than has hitherto been available. This is to include off-line simulation and analysis allowing some degree of system optimization as well as providing a means of assisting in the selection of the most appropriate sensing strategy for the required task.

Before any simulation or programming can be attempted, some form of notation leading to a task representation must exist. This usually takes the form of an algorithm. Many such notations are in use for describing algorithms and a selection of the relevant ones will be discussed in chapter 2. Their corresponding analysis is covered in chapter 3 and then some of them are expanded using matrix techniques in chapter 4.

The programming is at task and object levels only, though a thorough definition of all programming levels, including sensor parameters has been formulated and is presented in chapter 5. Graphical simulation depicting actual geometrical robot movements such as that provided by many simulation packages like SAMMIE [Heginbotham et al, 1979] is not considered part of this work.

Interaction with sensor systems is an essential part of any robot cell which includes error recovery. This aspect and its practical implementation forms the core of chapter 5. In chapter 6 these philosophies are combined with the techniques of the preceding chapters to yield the basis for an object level programming strategy based on Markov chains. Selected case studies are explored during chapter 7 using these, and other related, modelling and programming methods. Finally, during chapter 8, the results of this work are compared and contrasted with modelling techniques used in other related fields of simulation and programming.

The necessary physical parameters, including times of physical actions and probabilities of outcomes, are expected to be available from information compiled over a period of operational time on a computer database. This ties in with the work carried out by Song [Song, 1988], Ghis [Ghis, 1989] and Halloran on error recovery [Halloran, 1989].

Where texts are referred to, usually the one with the most easily readable explanation is listed, even though other descriptions of the same topic may exist in other texts given in the references. Only where a text is referred to many times for different purposes, or in the case of larger texts containing no index, are page numbers included.

## 2. NETWORKS

How complex or simple a data structure is depends critically upon the way in which we describe it. Most of the complex structures found in the world are enormously redundant, and we can use this redundancy to simplify their description. But to use it, to achieve the simplification, we must find the right representation.

Herbert A. Simon, *The Sciences of the Artificial*.

The term "network" is often used to describe any combination of intersecting paths whether they be in the form of subterranean passages or electrical wiring circuits. The only common attribute shared by all types of network is that of some kind of order as opposed to simple random chaos. However, to make use of any network, what is important are the factors governing this order.

The first thing which must be established with regard to any form of network is the notation to be used. This will depend heavily on the task to be represented and the corresponding analysis required. Many such notations are already established as standard. A few of these will be discussed in the rest of this chapter.

### 2.1 Flowcharts and Stategraphs

The flowchart, perhaps the oldest form of network notation, has been utilised by Lee [Lee et al, 1987] for asynchronous robot workcell process representation. The notations of flowcharts and their uses are well known and extensively documented elsewhere [Forsythe et al, 1975] and will not be discussed in any depth here.

Albus [Albus et al, 1982] uses a similar state graph representation to describe a hierarchical robot control system. This uses the structure shown in figure 2.1

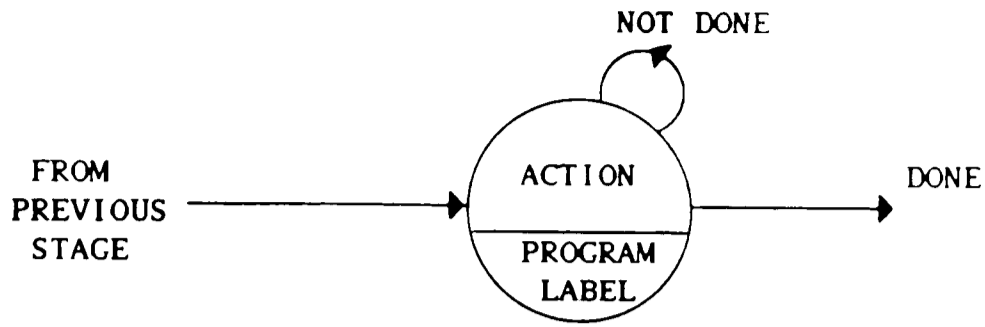
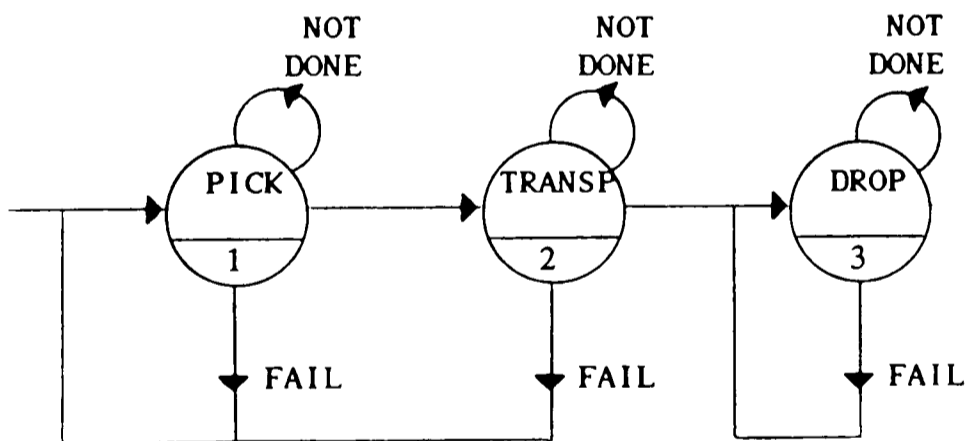


Figure 2.1. - State Graph Notation.

Like flowcharts, this notation has the advantage of being translatable straight into computer code for robot control using languages such as VAL [VAL II, 1984]. Figure 2.2 shows the state graph for a simple pick and place operation and its corresponding pseudo code representation.



```

1  PICK
  IF PICK - FAIL THEN GOTO 1
2  TRANSP
  IF TRANSP - FAIL THEN GOTO 1
3  DROP
  IF DROP - FAIL THEN GOTO 3

```

Figure 2.2. - Pick & Place Model with Pseudo Code.

In many cases the 'not done' loops are somewhat superfluous and so may be omitted from figure 2.2, if desired. Most processes have an outcome of 'successful' or 'failed' with 'not done' being otherwise assumed.

Though easy to read and convert into code, this kind of notation is somewhat clumsy and lacks the ease of mathematical translation and manipulation inherent in simpler flowgraph notations. Furthermore, as algorithms become more complex the relationship between the flowchart or state graph and the computer code becomes increasingly less obvious.

## 2.2 Signal Flowgraphs.

The particular kind of network most relevant to process analysis and control is the flowgraph, the simplest manifestation being the directed graph, or digraph [Wilson, 1979].

Once a notation has been established, some form of network analysis is required. In the case of the digraph, a number of texts provide methods of translating the network into either an adjacency matrix [Carre, 1979] or a connectivity matrix [Tutte, 1966], depending on the information required. This may seem straightforward enough for simple digraphs, but if nodes are to be allowed to have different characteristics then this form of notation is inadequate. Whitehouse [Whitehouse, 1969] suggests examples of the use of simple flowgraphs for a multitude of tasks. Then using GERT [Pritsker, 1966] shows how flowgraph nodes may have different logical properties (more of this in section 2.5). These and other notations will be discussed later. However, most use will be made of the digraph owing to its simplicity of notation and its ability to be expressed in concise mathematical form.



The digraph notation used throughout the rest of this work will be the standard notation found in queueing theory and operations research. This is because it reads in the same direction as the flow between nodes rather than the reverse as is often encountered in the electrical and control theory notation. So, the simple node to node path of figure 2.3 reads from node<sub>i</sub> to node<sub>j</sub> in the main direction of flow.

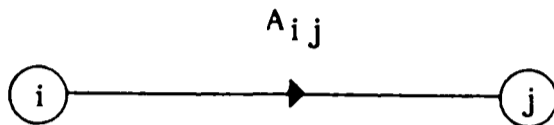


Figure 2.3 - Flowgraph Notation

The simple digraph version of the previous state graph pick and place example is shown in figure 2.4. Note that the main difference is in the juxtaposition of the activity labels from nodes to paths.

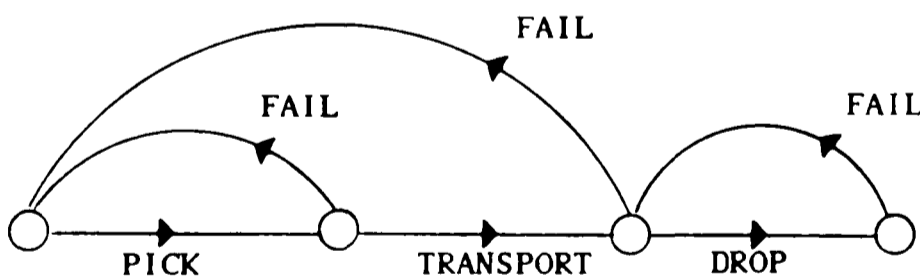


Figure 2.4. - Digraph Version of Figure 2.2

What the digraph models is not the physical movement of the object through the cell but that of the mover, i.e., the robot, itself. This is convenient in that it is impossible to program the object rather than the robot. It is perhaps an unfortunate choice of terminology that this level of robot programming is known as the 'object level'.

Due to the physical constraints of implementation of a robot workcell, the robot end effector can only be in one position along the digraph at any one time. It is impossible for any two paths of figure 2.4 entering the same node to be active simultaneously if the action of only one robot is being modelled. This means the digraph node inputs must behave in an exclusive OR manner.

### 2.2.1 Isomorphisms and Homomorphisms

At some stage in the modelling of a process we are likely to require some form of delimiting device as a means of isolating one part of a process from another.

Returning to the example given in figure 2.4. Here it will be noticed that if the DROP procedure fails, then after the recovery path has been completed, it is not only possible to re-execute the DROP procedure but also to return to the beginning of the sequence via the recovery path belonging to the TRANSPORT procedure. In the actual implementation of this program, this path may be possible due to physical implementation of the sensing at the time. In essence, occasionally the object might leave the gripper just after the sensor had been interrogated (a typical short coming of discrete sensing).

In reality this may not always be the case and under a different sensing regime, ie., if this sensor were not checked at the beginning of the DROP procedure, then this traversal of feedback paths would not be possible. In this case the true flowgraph would be that of fig. 2.5 where the path between nodes 3 and 4 acts as a buffer isolating the first part of the sequence from the last. The number of paths entering each of the nodes is now different with the buffer, ie., the two representations are not isomorphic.

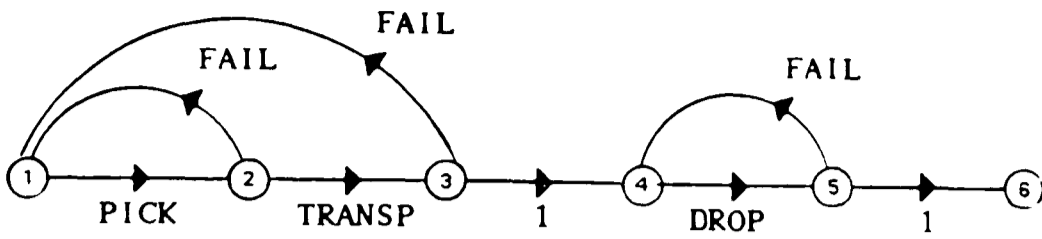


Figure 2.5 - Pick & Place example with Buffer path.

The electrical circuit equivalent is, of course, the familiar non-inverting unity gain amplifier. This kind of circuit is also used for the purpose of isolating two parts of a circuit, usually to prevent the effects of reflected impedances.

In other instances parallel forward paths may be required. This is not so easy to represent with numerically labelled nodes. For example, the path  $A_{1,2}$  connects nodes 1 and 2. Any other path directly connecting the same nodes in this order would also be named  $A_{1,2}$ . This problem can be overcome either by adding another subscript for each level of additional parallel paths, or by introducing redundant nodes in the same way as extra paths were added for buffering purposes.



Figure 2.6 - Homomorphic Graphs

Figure 2.6 shows the equivalent flowgraph to one with two direct parallel forward paths. In this case node 3 is redundant but its addition does not alter the operation of the network, the two flowgraphs are homomorphic.

### 2.2.2 Continuous Sensing Representation

To build a model for this type of sensing we must first describe all the robotic operations and their continuous sensing graphically. Using the notation proposed by Taylor [Taylor 1987] continuous sensing is depicted as in Figure 2.7.

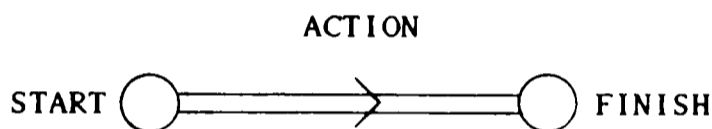


Figure 2.7. - A Continuous Sensing Representation.

With this notation the arrow denotes the next node to be approached in the event of some sensor transition, at which point the sensors are interrogated before the necessary recovery routine is executed. Naturally the signal flow represents the robot control parameters rather than the actual physical movement of the robot.

For example: If we were performing a robot operation which entailed releasing an object from the gripper, say by opening the gripper jaws at the appropriate destination (the START node for this operation), it would be sensible to continuously sense for the presence of the object and terminate the 'drop' action as soon as the task was complete. It would be considerably less clever to continue opening the gripper jaws, simply to complete the program segment (at the FINISH node), long after the object had left the gripper. Consequently, the end of the operation can be determined by sensor information. Clearly, the structure of continuous sensing is one of: 'On Sensor Transition GOTO...'

If we consider continuous sensing as the change in sensor parameters  $\partial S_i$  alone, then we must also incorporate some form of discrete sensing, ie. the monitoring of the absolute sensor state  $S_i$ . Otherwise, we will be unable to effect error recovery if an operation totally fails, because no sensor transition will occur. Fortunately we can use the end of a procedure as a sense parameter thereby effecting discrete sensing automatically.

### 2.3 Event Graphs

Yet another form of flowgraph is the 'Event Graph' [Schruben, 1983]. This consists of a set of preconditions, an event and a corresponding set of postconditions. With this we return to an "active node" notation, only this time the label denotes a discrete event such as the start or finish of some activity. Figure 2.8 shows the syntax for this form of flowgraph as given by Schruben.

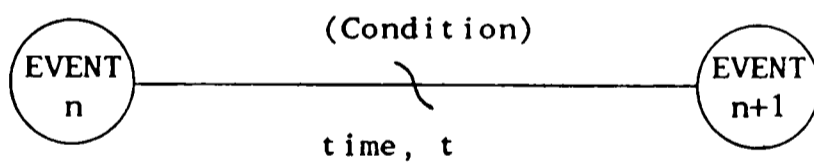


Figure 2.8. - Event Graph Notation.

This says that EVENT 2 will occur  $t$  seconds (or other time units) after the occurrence of EVENT 1 together with the fulfilment of the required condition(s). Unlike the digraph case of figure 2.4, an event is not an action in itself, rather it is a change in state giving rise to a beginning or end of an event. Consequently, the resulting event graph network is roughly twice the size of its digraph counterpart. An event graph is effectively a Petri net with exactly one input and one output transition [Valavanis, 1989].

## 2.4 Petri Nets

Another well established network notation is that of Petri nets. Petri nets have been used extensively in modelling space systems [Srinivas, 1976] for error recovery purposes, and where it can be used to detect deadlock easily in parallel situations [Cooke, 1987]. Petri nets are bi-partite directed graphs where particular states are depicted as circles and activities as bars. The paths are for interconnection only and unlike digraphs carry no intrinsic significance. Figure 2.9 shows a simple Petri net having two inputs and one output. The tokens, shown as dots, are event markers and are provided to help the user follow the operation of the network.

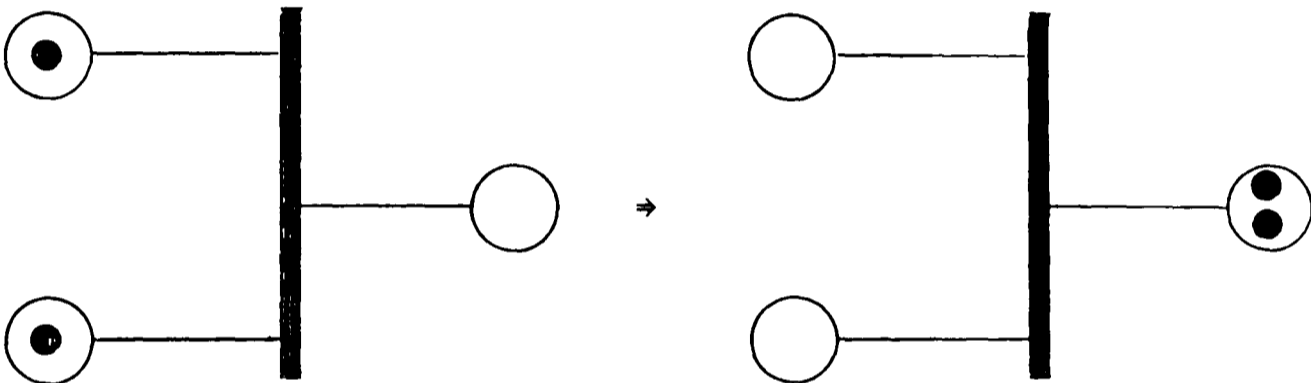


Figure 2.9 - Token flow through a Petri Net

Before any node output can occur, all input conditions must be satisfied and the output must be connected to a node which is available for occupation. [Reisig, 1985]. This gives the input characteristics an AND structure. Unfortunately Petri nets only model the flow of control and cannot be used to model the flow of resources [Valavanis, 1989].

For sequential situations Malcolm & Fothergill have investigated the modelling of the effect of sensors. One shortcoming is that it is more difficult to read when networks become large and complicated than is a simple digraph or GERT representation (see section 2.5). One advantage is that it can depict a required sequence explicitly as shown in Fig. 2.10. where the order is: "Do A, B & C in any order but B must not be last" [Malcolm & Fothergill, 1987]. It would be very difficult to ensure this kind of ordering using simple digraphs.

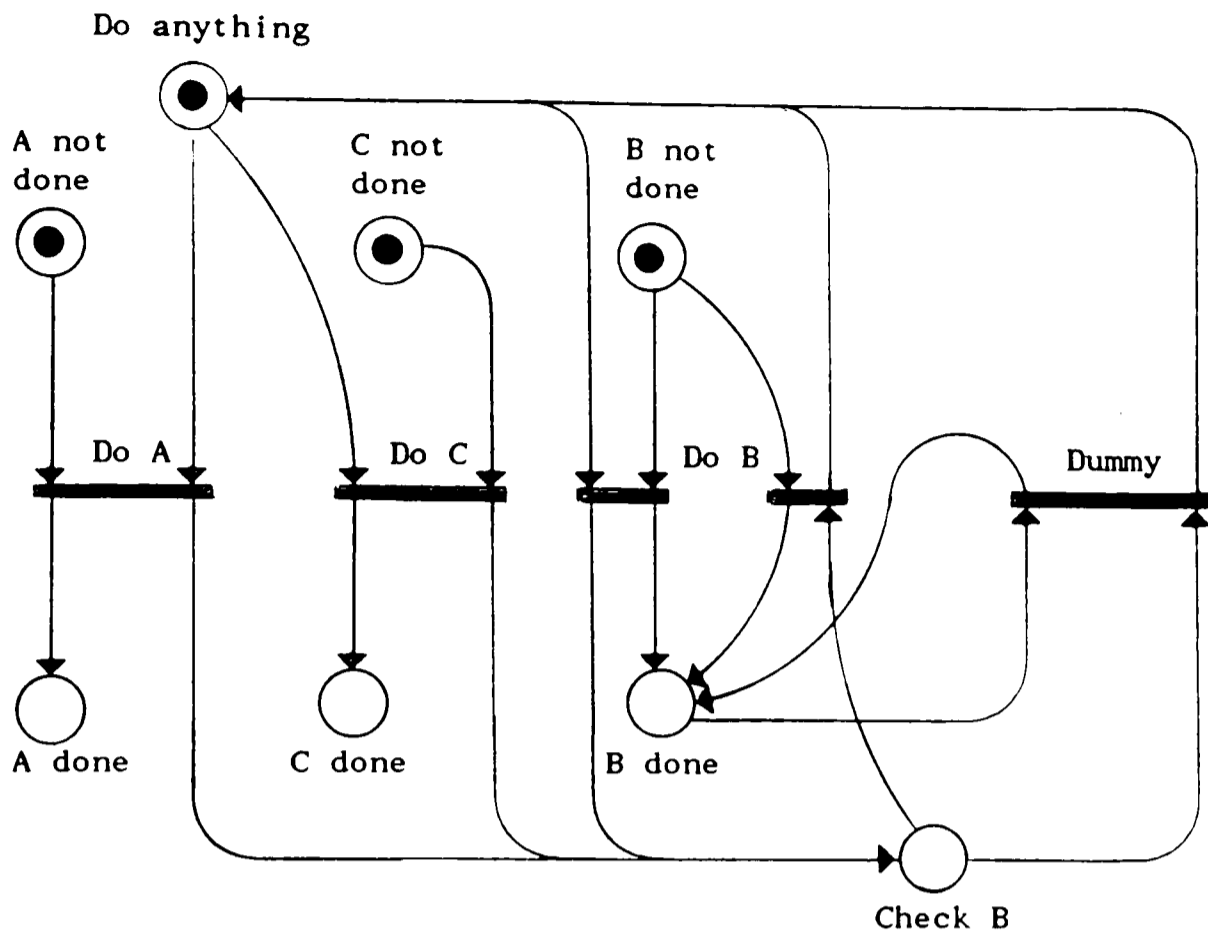


Figure 2.10. - Example of PETRI nets in sequencing.

### 2.4.1 The Token Machine

To make things easier to read Merlin [Merlin, 1974] uses an additional state graph representation which he calls a token machine. This is shown in figure 2.11 (b) alongside its corresponding Petri net of figure 2.11 (a).

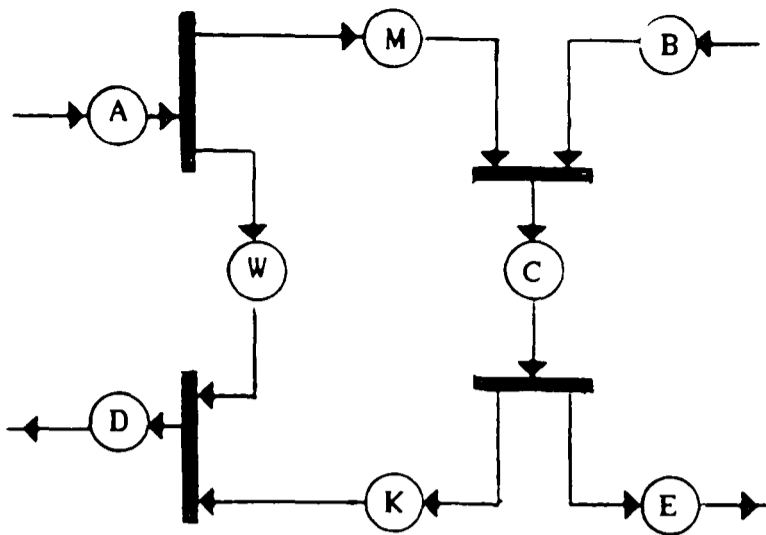


Figure 2.11 (a) - Petri Net

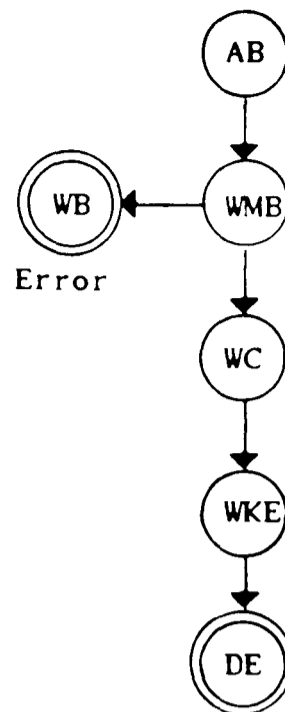


Figure 2.11 (b) - Token Machine

Despite the clumsiness of using two graphical representations simultaneously, Merlin states some useful points with regard to error recovery in computer systems:

1. A process P is recoverable from failure F if and only if in the Error Token Machine of P for failure F, all the directed paths through illegal states arrive at legal states in a finite number of steps.
2. A process is recoverable from failure F if and only if in the Error Token Machine of P for failures F: the number of states are finite, there are no final illegal states, and there are no directed loops containing only illegal states.

Statement 1 basically says that if an error occurs then a route must exist from which the process can recover. For example an error like 'bin empty' during a robot picking operation would constitute an illegal state from which recovery were impossible if no sensing were available to instruct the robot to cease attempting to pick out non-existent objects. A typical recovery strategy would be to allow a set number of unsuccessful attempts before requesting another bin.



Statement 2, on the other hand, concerns loops. In effect saying that there should be no infinite loops, or finite loops which contain only illegal states. This is because a loop containing only illegal states would mean that any form of recovery would lead to yet another illegal state.

These philosophies are of great importance when forward or backward chaining is used [Fielding et al, 1987] to return to a legal state in the event of an error causing the existence of an illegal state. This is true regardless of the graphical representation used.

#### 2.4.2 Stochastic and Time Petri Nets

Time Petri nets were introduced by Merlin as a modification to the standard Petri net representation intended to accommodate temporal constraints. The timing features are basically: a minimum time  $t_1$  which can elapse before the bar must fire after all the inputs are present, and a maximum time  $t_2$  for which all inputs may be enabled during which the bar does not fire.

Another time aspect is that of a variable time, rather like the continuous sensing flowgraph model. With regard to Petri nets this is a relatively new concept described as "Fuzzy-time" Petri nets [Valette et al, 1989]. However, the basic idea has been around for some time as will be seen when the GERT network is investigated.

Discrete time stochastic Petri nets (SPN) have been investigated to model Markov processes where node firing probabilities and delay times are used [Molloy, 1985]. Naturally, anything which can be used to represent a Markov process must map onto a transition matrix. Incidentally, a flowgraph does this far better than a Petri net.

Further variations on the Petri net theme include the "extended Petri net" which allows for several types of token and different classes of node [Ahuja & Valavanis, 1987]. Event graphs of the type covered in section 2.3 are considered as a special class of Petri net with deterministic output and exclusive-OR type input conditions. A more thorough notation involving a complete class of logic input and output conditions is GERT which will be discussed next.

## 2.5 PERT and GERT.

Simple signal flowgraphs of the type used so far have a number of serious shortcomings. The nodes inherently act as inclusive OR to all inputs, though as already mentioned, in most cases exclusive OR is required and even assumed. Whichever logic function is used it is impossible to distinguish easily between types. On the other hand Petri nets are restricted to an AND input characteristic. In either case no definition is given to the type of output from a node using SFG's.

Program Evaluation and Review Technique (PERT) was originally formulated to depict probabilistic networks. Unfortunately PERT will not handle feedback loops, and as this is the essence of any error recovery strategy such methods must be left to the overall modelling of the process as a whole. With this in mind we must construct and analyze the models of error recovery networks in which all feedback loops are constrained using other techniques. These may then form the building blocks of larger networks, without feedback loops which may fit into the modelling scenario for which PERT was conceived.

As mentioned previously Graphical Evaluation and Review Technique (GERT) employs a precise notation which overcomes the ambiguities of the SFG and Petri net notations. Tables 2.1 and 2.2 give a brief outline of the possible node constructions [Whitehouse, 1973]:

Table 2.1 - GERT Inputs






NODES INPUTS	
EXOR	
INC-OR	
AND	

Table 2.2 - GERT Outputs

NODE OUTPUTS	
DETERMINISTIC	
PROBABILISTIC	

This gives a combination of six possible node types:



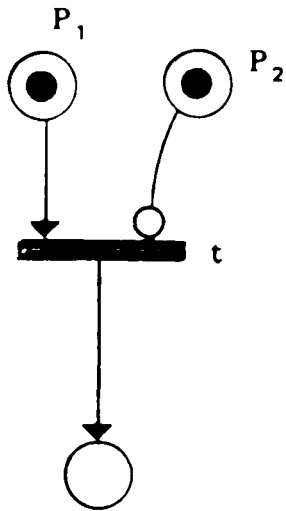
Whitehouse uses the Laplace operator to perform time analysis on networks containing EXOR nodes only (see chapter 3). As yet no one has developed satisfactory procedures for the analytic solution of GERT systems involving AND and INC-OR nodes.

## 2.6 Concurrency

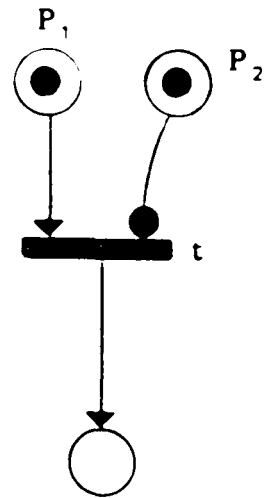
If, for instance, I say, "That the train arrives here at 7 o'clock", I really mean something like this: "The pointing of the small hand of my watch to 7 and the arrival of the train are simultaneous events".

A. Einstein.

This is an important aspect when more than one process is to be modelled or executed simultaneously. With its rigorous logic structure GERT possesses a high degree of precedence regulation, and as portrayed in section 2.4 Petri nets are ideal for representing precedence conditions. Further to this, Peterson [Peterson, 1981] considers the introduction of two further attributes to the Petri net model: the inhibitor and the activator nodes.



(a) Inhibitor arc



(b) Activator arc

Figure 2.12 - Control arcs

If the inhibitor arc of figure 2.12 (a) has a token at  $P_2$  then the transition  $t$  cannot fire and the token from  $P_1$  may not pass. Alternatively, for the activator arc of figure 2.12 (b), a token must be at  $P_2$  for  $t$  to fire and for the token at  $P_1$  to pass. With these control arcs, the token at  $P_2$  does not actually pass through when transition  $t$  fires, but is simply required to inhibit or activate the transition accordingly. This ambiguity between flow paths and information or control paths can lead to much confusion, particularly in larger networks.

This anomaly can be removed by using a different notation for the control paths as is done by Taylor in the extended flowgraph notation by the use of a dotted line for control signals [Taylor, 1987]. This feature is not available in conventional SFG theory and represents an extension for which as yet the mathematical analysis has not been fully resolved.

## 2.7 Logistics and Planning

When any form of network includes probabilities, as in the case of error recovery loops, at some stage parts of the network will behave like queues. Much has been written on queueing theory [Lee, 1966] and a brief resume of its nomenclature is given in appendix A.

Some multi-input closed queueing systems have been analysed from a timing (or cost) perspective [Grubbstrom & Lundquist, 1987], though they cannot all handle circuits (feedback loops). Koenigsberg deals specifically with cyclic queues [Koenigsberg, 1958] and shows an example used in the modelling of idle, mean cycle times etc. for machines working a coal face.

Many of the notations previously discussed are applicable to the modelling of queues, particularly the event graph. Many simulation packages, aimed mainly at queueing problems, exist. All of which contain a set of user selectable probability distributions for controlling the occurrence of events. These will be discussed in greater depth in chapter 6. A selection of useful probability distributions is provided in appendix D.

The scheduling of robot assembly tasks is often determined greatly by the ordering constraints of the individual components. Sometimes several alternative procedures are available to effect the same assembly. In such instances it may not necessarily be the most temporally economical order which is the most logistically viable. The order and manner in which each of the components are presented to the robot workcell may have a strong influence. Fox & Kempf [Fox & Kempf, 1985] present two strategies: 'fixed-buffer' where all the components are presented either in the order they are required for assembly, or as bins or stacks of one type of component only in each. Alternatively, 'fixed-build' is where all the components are presented at random, for example as a common bin of all component parts. It is then up to the robot system to either identify and select the components in the order they are required for assembly, or to pick them from the bin at

random and then identify and utilise them in the forming of smaller sub-assemblies which can be later connected together as the final assembly. Naturally, this final scenario relies on more than one possible order of assembly if it is to be used efficiently. If only one possible order exists, then the robot would simply remove all the parts from the bin, identifying and depositing them in a known position as it did so, only to re-pick them in the required order for assembly – not a very efficient approach.

Fortunately, the fixed-build strategy is often a non-problem as most component parts are manufactured individually and therefore could be presented to the robot in individual collections. For example nuts and bolts may be purchased in mixed packs, but they are never manufactured by the same machine in a totally random manner. Consequently, problems associated with bin picking are usually restricted to separating like parts from one another – a much more straightforward, if not always simpler, task.

As a result of this narrowing-down of the number of possible task order permutations to only those which are logistically viable, it is unlikely that any resulting single cell network will have more than a few possible forward paths. Only when many such cells are connected to form a larger network where parallel processes may exist do "Shortest Route" techniques [Boffey, 1982] concern us. Moreover, when parallel processes may be active simultaneously, then critical path methods (CPM) may be necessary. Bedworth & Bailey demonstrate a number of project planning techniques with case studies using CPM with PERT [Bedworth & Bailey, 1987], whilst Elsayed & Boucher [Elsayed & Boucher, 1985] investigate the use of CPM and PERT together with linear programming techniques to the same ends.

As many texts already exist covering the use of CPM, PERT, linear programming etc., many of which are standard text-book techniques [Taha, 1987], the rest of this work will concern only those networks containing feedback loops pertinent to sensing and error recovery strategies.

## 2.8 Summary.

Flowgraph techniques, appear to fall into two main groups: 'active path' and 'active node' networks. Signal flowgraphs, PERT & GERT etc are active path systems. That is to say, the information regarding the process(es) in hand are contained in the paths. Active node systems on the other hand, such as: Flowcharts and stategraphs use paths simply as connectors between the nodes which contain all the relevant process information. A comparison between flowcharts and their electrical and hydraulic counterparts reveals a close similarity, but not an exact correspondance [Kodres, 1978]. This is because active node and active path networks are not always homomorphic, ie. not exactly equivalent (see appendix B). Despite this, in most cases they can be used to model the same process, because they share an isomorphism, ie., the output of one representation is the same as that of the other even though the internal workings may differ. This equivalence has been shown to belong to a restricted set when flow languages and Petri net languages are compared [Araki et al, 1981].

Petri nets and event graphs differ from flowgraphs in that they depict transitions rather than activities, though they behave essentially as active node graphs. The fact that Petri nets can be used to represent Markov processes [Denham, 1989] means that they could be used in place of flowgraphs if so desired. However, their mathematical analysis is considerably more complex, if possible at all.

Signal flowgraphs have the distinct advantage of being easily manipulated in matrix form as will be shown in chapter 4. Unfortunately SFG's suffer from a slight ambiguity in their nodal logic properties not present in the GERT notation. Both notations are suitable for representing error recovery loops and will be used extensively for this purpose throughout the remainder of this work.

### 3. NETWORK ANALYSIS

Some methods of obtaining useful data from flowgraph representations of robot workcell networks are presented in this chapter. The use of some of the techniques normally found in electrical network analysis are discussed together with the results of some simple experiments.

The use of flowgraphs to represent actions of machinery, scheduling of parts flowing through a production line etc., has been a useful means of depicting a process or set of series/parallel processes for many years now, and as will be seen in chapter 6. many simulation techniques already exist.

With the emergence of robotics and its continuing expansion within industry, many of these methods have been used successfully in representing robot workcells [Cash, 1986]. The growing need to cope with problems autonomously within a cell has led to the investigation of error recovery within robot workcells [Lee, 1984] and its corresponding representation in flowgraph form [Taylor, 1987].

When used for representing simple electrical circuits, such as resistance networks, Kirchoffs laws may be used because all the inputs to each node are basically inclusive OR. Unfortunately, this is no longer true when modelling many sequential processes, such as a robot workcell, where the process can only be active at any one position along the path(s) of the flowgraph at any one time. In this case the inputs to nodes are usually treated as exclusive OR and the following sections provide the necessary tools for such an analysis.



### 3.1. Flowgraphs, Costs and Probabilities.

It is remarkable that a science which began with the consideration of games of chance should have become the most important object of human knowledge.

Marquis de Laplace.

When using signal flowgraphs with all EXOR inputs, the simplest representation of a very basic error recovery loop is shown in Fig. 3.1. Here, the cost or toll  $t_{1,2}$  is the time required to complete some action (say a robot pick-up operation). Node 1 is the starting point and node 2 the point at which the action should have finished and sensing is carried out to verify whether the action was successful or not.  $P_{2,3}$  is the probability of success whilst  $P_{2,1}$  is the probability of failure (ie. the pick-up operation failed or the object dropped off the gripper before node 2 was reached) with its corresponding recovery cost  $t_{2,1}$  being the time required to return to node 1 and start again.

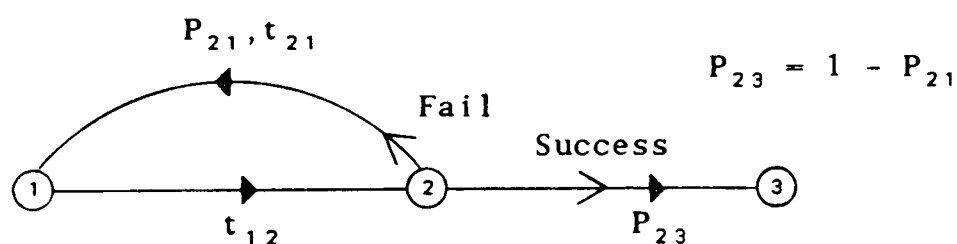


Figure 3.1. - Simple Error Recovery Loop.

Now, given one failure detected at node 2, and assuming there is the same probability of failure after recovery at node 2 again. If an operation is of a deterministic nature, then it is not possible for that procedure to only partly succeed. Since  $P_{2,1} < 1$ , then eventually (though it may take a long time if  $P_{2,1} \gg P_{2,3}$ ) a success must be achieved.

This may seem at first sight to be an over simplification of an error recovery strategy. However, the simple "repeat last operation" feedback path may also contain further routines such as "discard faulty part" etc. Furthermore, in a real production environment, the vast majority of error recovery routines are merely required to return the system to a normal running state as quickly as possible, usually by simply repeating the failed operation. Elaborate error recovery routines are rarely cost effective, even if practical.

One of the important requirements in scheduling a workcell is a knowledge of the expected cost of an operation given the appropriate probabilities of success and failure. The same information is of equal importance in assessing the throughput and hence cost effectiveness of the cell.

It is apparent from Fig. 3.1. that an expected average time to complete the operation of such a loop will be:

$$t_{av} = t_{12} + (1 - P_{23})(t_{12} + t_{21}) + (1 - P_{23})^2(t_{12} + t_{21}) + \dots$$

$$t_{av} = t_{12} + (t_{12} + t_{21}) \sum_{n=1}^{\infty} (1 - P_{23})^n$$

but  $P_{21} = 1 - P_{23}$

$$\therefore t_{av} = t_{12} + (t_{12} + t_{21}) \sum_{n=1}^{\infty} P_{21}^n$$

Now using the geometric progression [Spiegel, 1968].

$$\frac{a(1 - P^n)}{1 - P} = a(1 + P + P^2 + P^3 + P^4 + \dots + P^{n-1}) \quad (3.1)$$

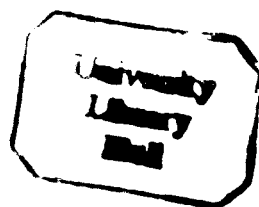
substituting  $P_{21}$  for  $P$  in (3.1)

as  $n \rightarrow \infty$ ,  $P_{21}^n \rightarrow 0$  (because  $P_{21} < 1$ )

and hence it can be shown that

$$\sum_{n=1}^{\infty} P_{21}^n = \frac{P_{21}}{1 - P_{21}} = \frac{P_{21}}{P_{23}}$$

thus  $t_{av} = \frac{t_{12} + t_{21} P_{21}}{P_{23}}$  (3.2)



Equation {3.2} is quite a simple expression but represents only a simple single loop. Due to the additive property of time and the multiplicative property of probabilities, the approach becomes very cumbersome in cases of multiple loops, such as that of Fig. 3.2.

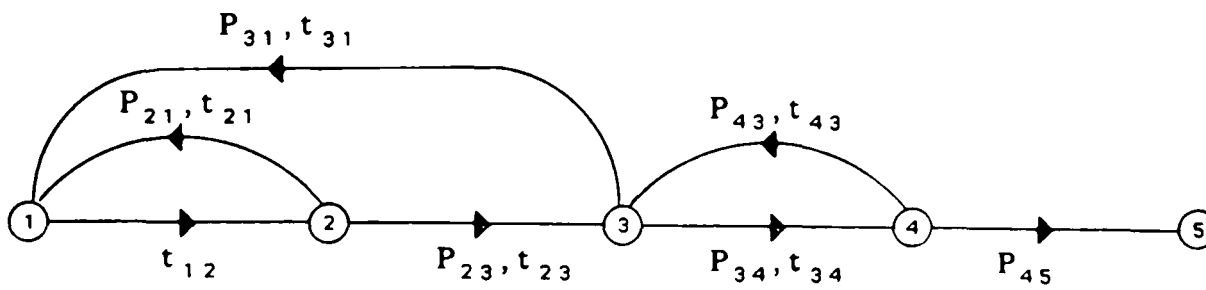


Figure 3.2. - Multiple Loop Flowgraph.

However, if it can be assumed that the probabilities of success and failure do not change, as in the case of fig. 3.1., that is to say each event is independent of the outcome of the previous one, then we can introduce some very powerful techniques.

### 3.2. Independence and the Markov Property.

The first of earthly blessings, independence.

Edward Gibbon.

Firstly, the property of independence of past events (known as the Markov property) must be defined [Cooper, 1981]. If the probability of success after  $k$  failures is  $P_f^k P_s$  and  $N$  is the number of failures occurring before a success during some operation, then:

$$P(N = k) = P_f^k P_s$$

and  $P(N > k) = P_f^k$

where  $P_s$  and  $P_f$  are the probabilities of success and failure respectively, as defined previously.

Assuming each attempt has the same probability of success  $P_s$ , and given that at least  $j$  failures have occurred, then the probability of success after  $j + k$  attempts can be found using Bayes theorem thus:

$$\begin{aligned}
 P(N = j + k | N \geq j) &= \frac{P(N = j + k) \cap P(N \geq j)}{P(N \geq j)} \\
 &= \frac{P_f^{j+k} \cdot P_s}{P_f^j} = P_f^k \cdot P_s
 \end{aligned}$$

$$\therefore P(N = j + k | N \geq j) = P(N = k)$$

$$\text{or } P(N \geq j + k | N \geq j) = P(N \geq k)$$

So, given that the probabilities being used have this Markov property, then over a large number of events these probabilities may be treated as constants. The more usual descriptive example is that of tossing a fair coin (each event is independent of the last) and over a large number of tosses of the coin there will tend towards an equal number of heads and tails giving a probability of each outcome as exactly one half. Even though tossing a coin say only ten times may yield a result of six heads and four tails.

It is this ability to treat probabilities as constants when considering a process over a large number of events which allows us to make use of the Laplace and Z-transforms.

### 3.3 The Z Transform

I am a great believer in running before you can walk, because by finding out how difficult it is to run, one takes greater interest in the problem of learning to walk.

Lt. General, Sir Brian Horrocks.

The Z-transform is defined as:

$$Z = e^{sT}$$

where T is the sampling interval and S the LaPlace operator.

Now, a robot picking items from a stack, just as objects arriving at a workcell for processing, can be thought of as a train of discrete events we can use the Z-transform to represent time delays through the process. For example  $Z^{-\alpha}$  represents a time delay of  $\alpha$  time units. Also, just as the gain (or attenuation) in a network can be represented by a multiplier, ie  $\beta Z^{-\alpha}$  where  $\beta$  is some gain factor, so can we use probabilities [Huggins, 1957].

It must be remembered that all probabilities are less than or equal to one and so act more like attenuations than gains. This is important from a view of dynamical stability.

Returning to Fig.3.1., this network can be expressed using Z-transforms as in Fig.3.3.

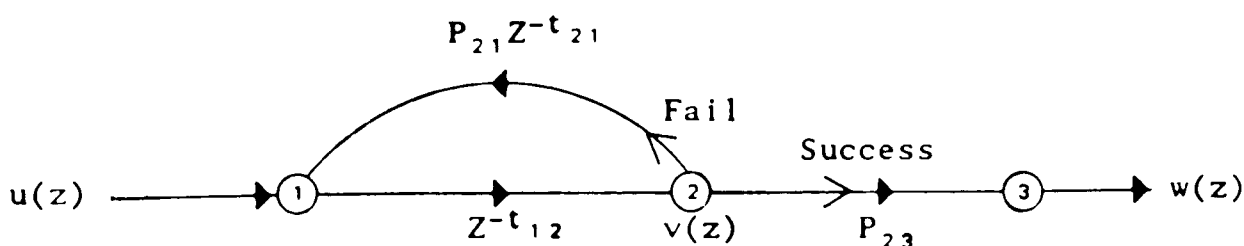


Figure 3.3. - Simple Loop Using Z-transforms.

Given a discrete sequence of unit events into the network  $u(z)$ , and a corresponding output of events  $w(z)$ , a transform function  $w(z)/u(z)$  can be derived to represent this network as follows:

$$v(z) = Z^{-t_{12}} (u(z) + P_{21} Z^{-t_{21}} v(z)) \quad (3.3)$$

$$v(z) (1 - P_{21} Z^{-t_{12}-t_{21}}) = Z^{-t_{12}} u(z)$$

$$w(z) = P_{23} \cdot v(z) \quad (3.4)$$

eliminating  $v(z)$  between equations (3.3) and (3.4) gives

$$\text{Transfer function, } TF(z) = \frac{w(z)}{u(z)} = \frac{P_{23} Z^{-t_{12}}}{1 - P_{21} Z^{-t_{12}-t_{21}}}$$

Differentiating  $TF(z)$  with respect to  $Z$  will give a rate of flow of events passing through the network. Elmaghraby [Elmaghraby, 1977] uses this technique and by setting  $Z = 1$  to give the steady state we have the time average for the network.

So.

$$\frac{\partial TF(z)}{\partial Z} = \frac{P_{23} Z^{-t_{12}} P_{21} (t_{12} + t_{21}) Z^{-t_{12}-t_{21}-1} + t_{12} P_{23} Z^{-t_{12}-1} (1 - P_{21} Z^{-t_{12}-t_{21}})}{(1 - P_{21} Z^{-t_{12}-t_{21}})^2}$$

$$\left. \frac{\partial TF(z)}{\partial Z} \right|_{Z=1} = \frac{P_{23} (t_{12} + t_{21}) P_{21} + t_{12} P_{23} (1 - P_{21})}{(1 - P_{21})^2}$$

$$\text{but } P_{23} = 1 - P_{21}$$

So substituting for  $P_{23}$  and simplifying gives the original expression

$$\therefore \left. \frac{\partial TF(z)}{\partial Z} \right|_{Z=1} = \frac{t_{12} + t_{21} P_{21}}{P_{23}} \dots \text{ which is equation (3.2)}$$

Using the Z-transform we overcome the difficulties of mixing variables with additive and multiplicative properties. This allows us to calculate transfer functions, time expectations etc., using the usual network techniques such as Mason's theorem [Mason, 1956]. In fact Pritsker [Pritsker, 1966] uses the Laplace transform (instead of the Z-transform) to obtain the same results by differentiating the network transfer function and setting  $s = 0$ . He then goes on to derive Mason's theorem by these techniques. Calculation of the variance can also be achieved from information given by the second derivative of the transfer function.

### 3.4. A Practical Example.

Table 3.1. contains data obtained from a large number of tests performed on a textile pick operation by a robot using a very basic electrostatic roller gripper [Monkman, 1987] fitted with only discrete sensing capabilities. The force of lifting is set to ensure that only one ply at a time may be lifted. In the event of two or more plies sticking together, the pick operation fails and the robot returns to the start of the pick operation. Similarly, on dropping the ply the operation is repeated until successful.

OPERATION	TIME REQUIRED	SUCCESS PROBABILITY
PICK	7.04	0.841
TRANSPORT	2.50	0.968
DROP	3.95	0.812

Table 3.1. Robot Operation Data.

The times quoted in table 3.1 are in seconds. These were timed for each operation within the program control (using computer internal clock) and so may be considered accurate.

The resulting flowgraph for these 3 operations and their respective feedback paths are shown in figure 3.4. The values of Probability and time for each path are expressed as Z-transforms.

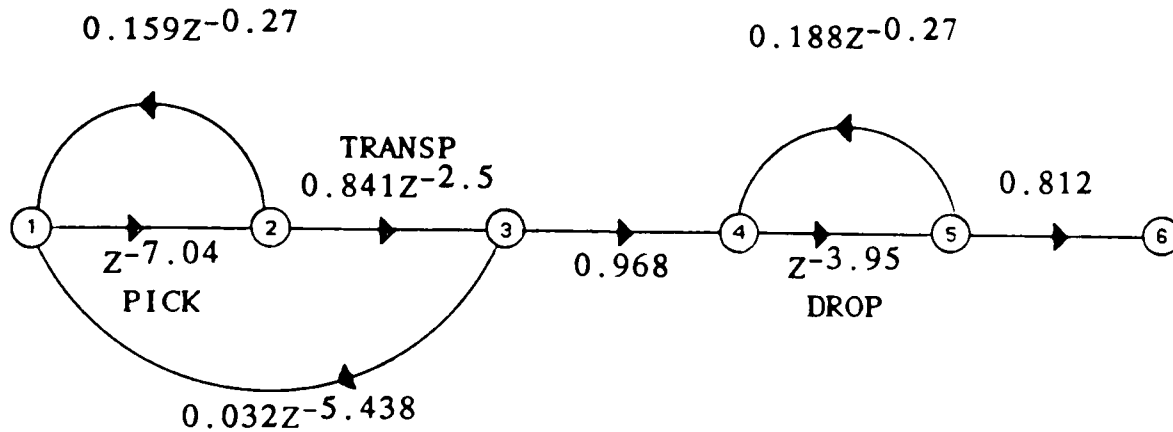


Figure 3.4. - Flowgraph of Working Model

Using Masons theorem we obtain the transfer function:

$$TF(Z) = \frac{Z^{-7.04} \cdot 0.841Z^{-2.5} \cdot 0.968Z^{-3.95} \cdot 0.812}{1 - 0.159Z^{-7.31} - 0.188Z^{-4.22} - 0.027Z^{-14.978} + 0.0299Z^{-11.53} - 0.0051Z^{-19.198}}$$

and thus:

$$\left. \frac{\partial TF(Z)}{\partial Z} \right|_{Z=1} = \frac{0.661 \times 1.9159 + 0.661 \times 8.917}{0.661^2} = 16.39$$

If all three operations were perfectly successful and no error recovery was required, then the total operation time would simply be the sum of the forward path times ( $t_{nor} = 13.49$  seconds).

So the expected time increase due to error occurrence and recovery is:

$$(t_{av} - t_{nor})/t_{nor} = (16.39 - 13.49)/13.49 = 21.5\% \text{ time increase.}$$



100 New destack operations were conducted. Each of these normally takes 13.49 seconds (plus 1.488 seconds for the robot to return to the start of the PICK operation) each time. The total time recorded was 30 minutes and 11 seconds for the 100 complete cycles (each cycle including the total number of repeated attempts). This gives an average cycle time of 22.38 seconds. So the forward path average time is:  $18.11 - 1.488 = 16.622$  seconds. The resulting increase is:

$$(t_{av} - t_{nor})/t_{nor} = (16.622 - 13.49)/13.49 = 23.2\% \text{ time increase.}$$

This is only very slightly longer than the calculated time increase. This demonstrates the practical usability of this technique, even with relatively small sample sizes. Of course such processes have a variance as well as an average completion time, and this will be discussed later.

### 3.5 Isochronic Curves.

It is not the business of the botanist to eradicate the weeds. Enough for him if he can tell us just how fast they grow.

C. Northcote Parkinson.

Following from considerations of reliability of Markov processes by Siegrist who conducts various sensitivity analyses by finding the partial derivative of the system equation with respect to the individual reliability of one parameter [Siegrist, 1988], we can take reliability to be analogous to performance in our model and construct a graphical representation to act as a tool in comparing system performance by various parameter adjustments.

Isochronic plots were first introduced during the course of this work to provide a means of estimating the improvement in overall operational performance of a particular task due to various parameter changes such as a replacement of end effector, alteration of error recovery strategy etc [Monkman et al, 1989]. These ideas can be extended to the consideration of parallel situations where it may be shown that several workcells operating slowly, but simultaneously, can be more cost effective than a single faster operating workcell [Hartley, 1986].

Given the simple error recovery loop we are now familiar with and having the appropriate expected efficiencies and measured action times for each process (or part process), we are now in a position to determine just how long our error recovery routines may be allowed to take without increasing the average time of the whole process, given the probabilities of success and failure. Alternatively, we can determine whether it would be more cost effective to make a process time shorter at the expense of a decrease in the success/fail ratio, or improve the process efficiency whilst incurring a corresponding time penalty. All this is made much easier if we have a set of plots of the desired parameters against one another for a constant  $t_{av}$  (Isochronic average time). Figure 3.5 shows a plot of  $t_{av}$  against  $P_{23}$  for a simple error recovery loop.

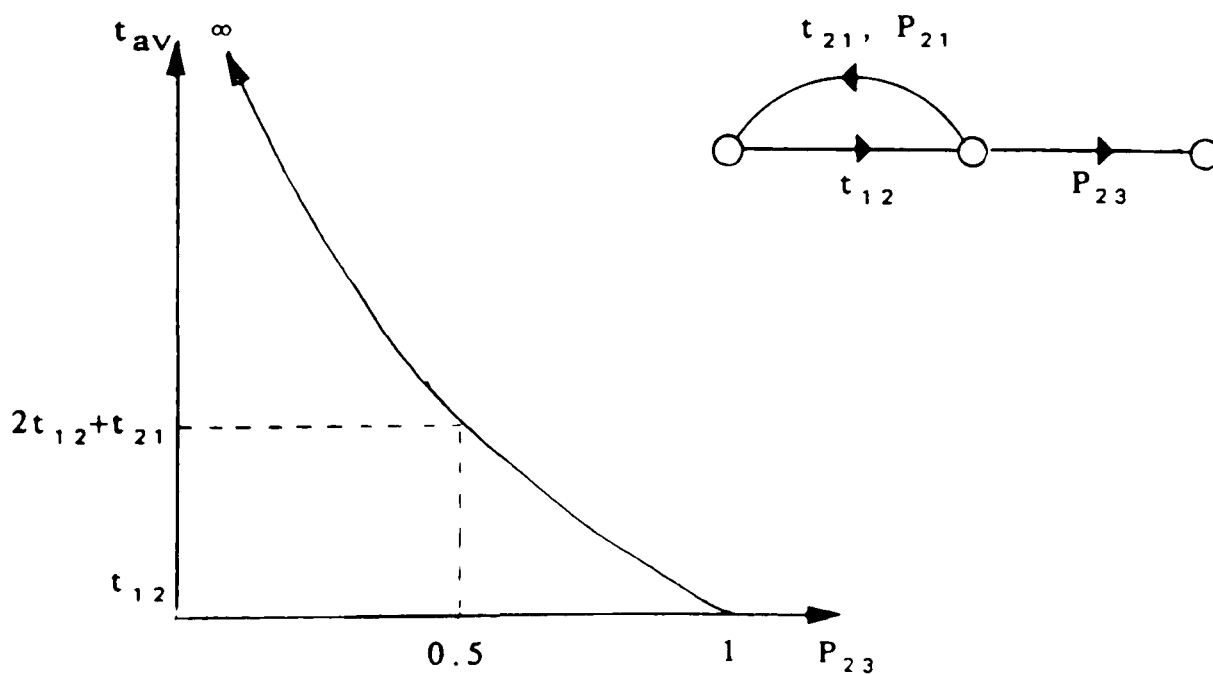


Figure 3.5 - Plot of  $t_{av}$  against  $P_{23}$

Figure 3.5 shows us how the average process time will increase dramatically as the process efficiency drops with the curve tending towards infinite average time as efficiency drops to zero.

Isochronic  $t_{av}$  plots for action time  $t_{1,2}$  and recovery time  $t_{2,1}$ , against success probability  $P_{2,3}$  are what is really needed. These are shown in figures 3.6 and 3.7 respectively.

Using equation (3.2) again:

$$t_{av} = \frac{t_{1,2} + t_{2,1}P_{2,3}}{P_{2,3}}$$

transposing gives:

$$t_{1,2} = P_{2,3}(t_{av} + t_{2,1}) - t_{2,1} \quad (3.5)$$

Plotting the success probability  $P_{2,3}$  against the forward path time  $t_{1,2}$  as in figure 3.6 we can see that if  $P_{2,3} = 1$  then success will be achieved every time. Hence the average time  $t_{av}$  will simply be  $t_{1,2}$ , as the recovery path will never be accessed making  $t_{2,1}$  irrelevant in this case. Now, if we let  $t_{2,1} = 0$ , then as  $P_{2,3}$  is reduced, the forward path time  $t_{1,2}$  must also be reduced if the same  $t_{av}$  is to be maintained.

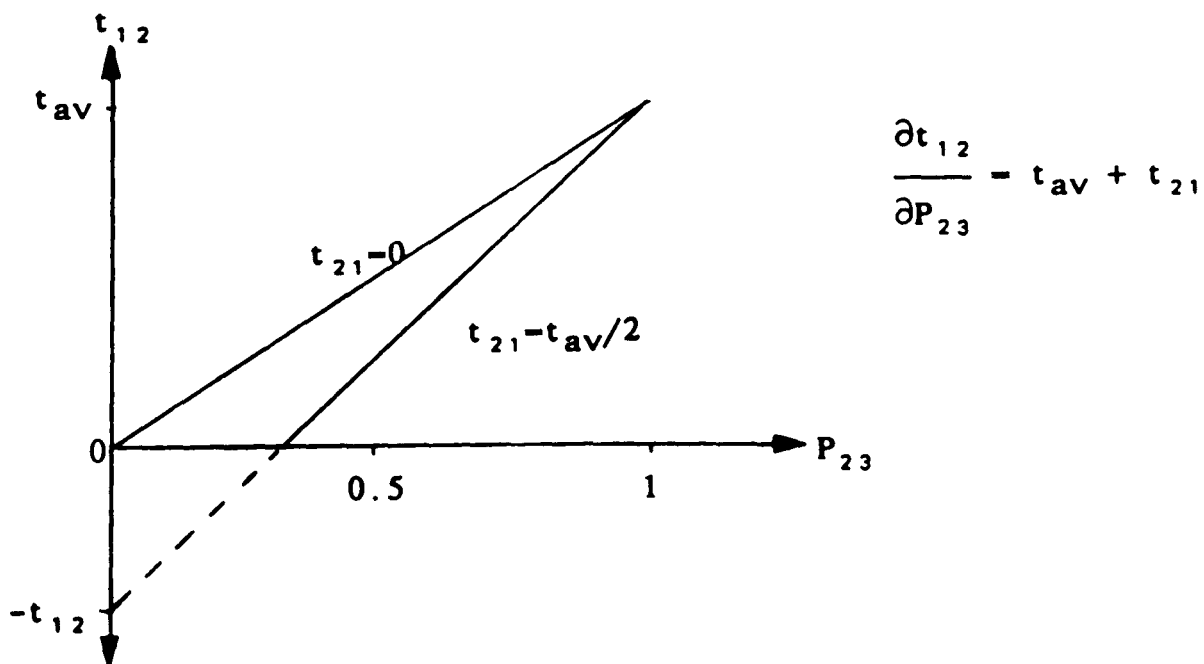
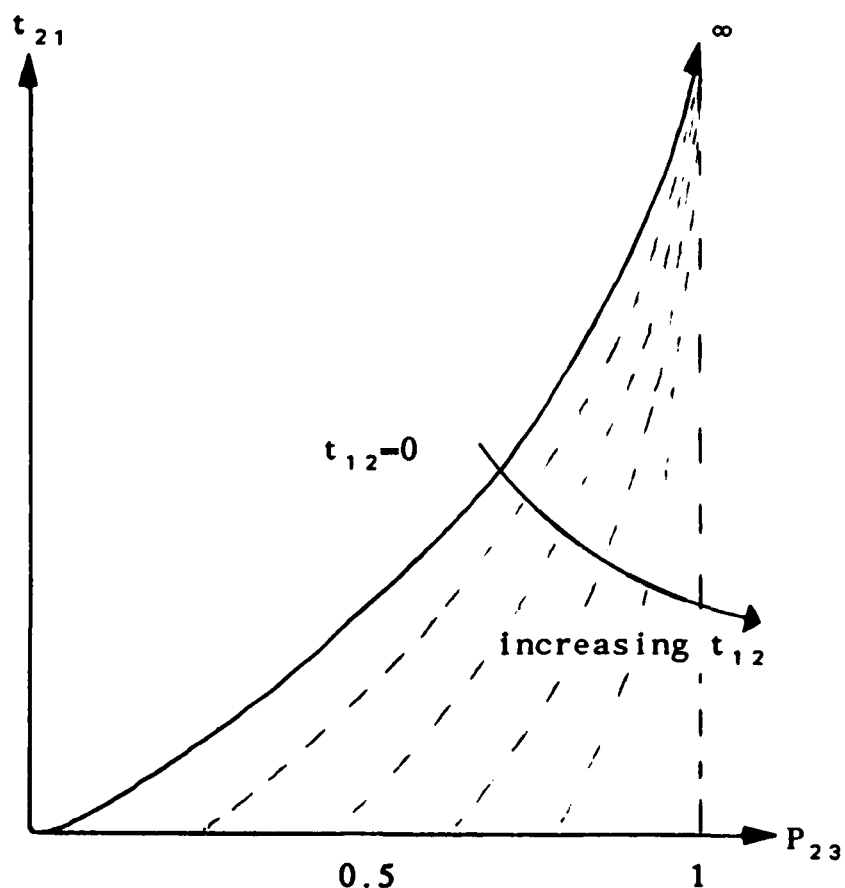


Figure 3.6 - Plot of  $t_{1,2}$  against  $P_{2,3}$  for isochronic  $t_{av}$ .

When  $t_{21}$  is equal to zero, doubling efficiency simply doubles the allowable action time  $t_{12}$ . However, as  $t_{21}$  becomes significant, the improvement becomes more pronounced. Also where  $t_{21}$  is greater than zero there exists a region where  $t_{12}$  becomes negative. Clearly this is practically impossible, though it does set a boundary as to how low the action efficiency may drop before the average process time must be increased regardless of how small  $t_{12}$  is kept.

Transposing equation (3.5):

$$t_{21} = \frac{P_{23}t_{av} - t_{12}}{1 - P_{23}} ; \quad t_{av} > t_{12} \quad (3.6)$$



$$\frac{\partial t_{21}}{\partial P_{23}} = \frac{t_{12} - t_{av}}{P_{23}^2}$$

Figure 3.7 - Plot of  $t_{21}$  against  $P_{23}$  for isochronic  $t_{av}$ .

Now, from figure 3.7, we can see that doubling the probability of success,  $P_{23}$  gives a considerable increase in the allowable recovery time, due to the slope of figure 3.7 following an inverse square relationship.

These techniques not only provide statistical expectations of the performance of a particular robot workcell configuration but also lend themselves to aiding the choice of sensor implementation with a view to improved efficiency of the overall process. Furthermore, changes in overall performance due to variations in time and efficiency parameters can be estimated using isochronic plots. This provides a useful tool in choosing processes and error recovery routines, as well as allowing an estimation of the improvements which may be expected for given system parameter changes.

### 3.6. Flowgraphs with Continuous Sensing

For any finite sequence of integers, I can always find a rule that tells me exactly how to construct the sequence. But the rule may be very complicated.

Heinz Pagels.

In the discrete sensing model, all actions must be completed before sensing and consequently, error recovery (if required) can be performed. However, under real robot control conditions much, if not all, sensing must be done continuously if maximum economy of time is to be achieved. For example, it would be a complete waste of time to continue a robot transport operation for an object which had accidentally fallen from the gripper. This, of course, is what would happen if only discrete sensing was employed.

As briefly discussed in section 2.2.2, a more efficient scheme is to monitor the state of the gripper/object relationship continuously and take whatever remedial action is required the instant an error occurs. It goes without saying that this holds true for virtually every possible robot action having a greater than zero probability of error occurrence.

In the discrete model, costs and probabilities of success and failure of an operation, or set of operations, can be used to calculate an average cost requirement. This is also true of the continuous model.

### 3.6.1. Tolls & Probabilities.

If it takes as long to state the rule, suitably transcribed into numbers, for the construction of the numerical sequence as the actual length of the sequence, then the sequence is 'random'.

Andrei Kolmogorov.

Now, in a simple single feedback loop, the feedback time  $t_f$  will usually be a function of the forward path time  $t_0$ . For example, if the robot had completed  $n$  seconds of a transport operation to the point at which a failure, resulting in the need for error recovery along the feedback path, occurred. Then the time of the feedback loop to return the robot to the point at which the procedure could be resumed would be  $f(n)$  seconds. The time  $t_f$  may be longer than the proportion of  $t_0$  achieved so far depending on the point of resumption and the path taken. The relationship is likely to be linear, though this may not necessarily be so. Fig. 3.8 shows the situation for successive executions of a simple feedback loop.

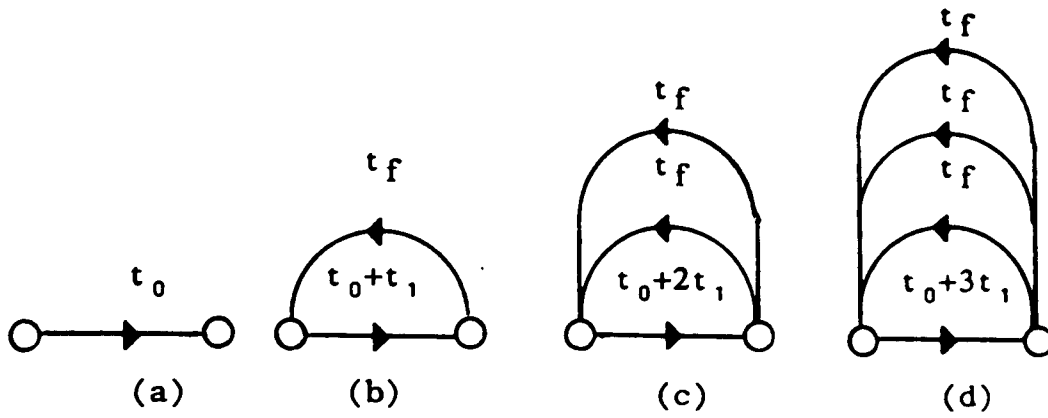


Figure 3.8. - Feedback with  $t_f$  as a function of  $t_0$ .

The flowgraph for a successful first attempt is given in Fig. 3.8 (a) with (b), (c) and (d) showing the result of repeated failures. The whole algorithm for a single loop may be expressed mathematically thus:

Collecting all the terms:

$$\begin{aligned}
 t_{av} = & P_0 t_0 + P_0(1-P_0)(t_0+t_1+t_f) + P_0(1-P_0)^2(t_0+2t_1+2t_f) \\
 & + P_0(1-P_0)^3(t_0+2t_1+3t_f) + \dots
 \end{aligned}$$

where  $t_0$  is the duration of a complete forward operation without error and  $t_1$  is the amount of time taken in an incomplete forward operation before an error occurs.

This reduces to:

$$\begin{aligned}
 t_{av} = & P_0 t_0 + P_0 t_0 \sum_{n=1}^{\infty} (1-P_0)^n + P_0 \frac{(t_1+t_f)}{2} \sum_{n=1}^{\infty} 2n(1-P_0)^n \\
 = & P_0(t_0(1 + \sum_{n=1}^{\infty} P_f^n) + \frac{t_1+t_f}{2} \sum_{n=1}^{\infty} 2nP_f^n) \quad (3.7)
 \end{aligned}$$

Now given that:

$$\sum_{n=1}^{\infty} P_f^n = \frac{P_f}{P_0} = \frac{1-P_0}{P_0}$$

and

$$\sum_{n=1}^{\infty} 2n P_f^n = \frac{2P_f}{(1-P_f)^2} = \frac{2(1-P_0)}{P_0^2}$$

Average value for continuous time becomes:

$$\begin{aligned} t_{av}(c) &= P_0 \left( t_0 \left( 1 + \frac{1-P_0}{P_0} \right) + \frac{(t_1+t_f)(1-P_0)}{P_0^2} \right) \\ &= P_0 t_0 + t_0(1-P_0) + (t_1+t_f)(1-P_0)/P_0 \\ &= t_0 + (t_1+t_f)P_f/P_0 \end{aligned} \quad (3.8)$$

Now, if  $t_1 = t_0$ , ie. error correction occurs only at the end of the forward path operation, then  $t_{av}(c)$  should be equal to the discrete model average time  $t_{av}(d)$ .

So using (3.8).

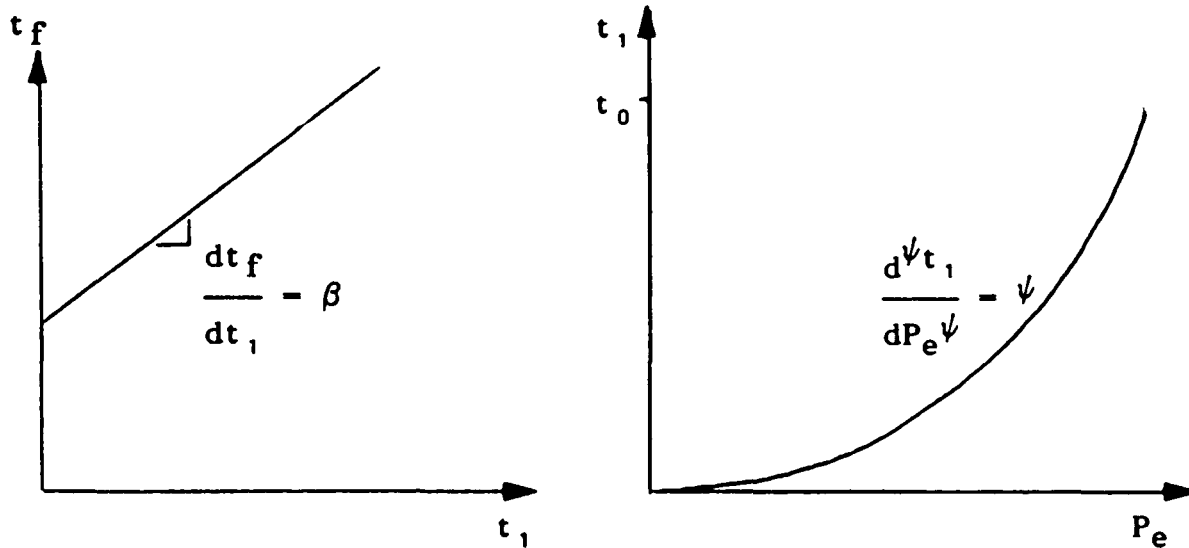
$$\begin{aligned} t_{av}(c) &= t_0 + (t_0+t_f)P_f/P_0 \\ &= \frac{t_0 P_0 + (t_0+t_f)(1-P_0)}{P_0} \\ &= \frac{t_0 + t_f(1-P_0)}{P_0} = t_{av}(d) \quad \dots \text{the discrete model.} \end{aligned}$$

This of course, is equation {3.2} derived in section 3.1.



### 3.6.2. Functions and Distributions.

As suggested previously, the recovery time  $t_f$  is a function of the forward path time  $t_0$ . Fig. 3.9 (a) shows a possible relationship, in this case assumed to be linear for simplicity. Likewise, the proportion,  $t_1$ , of the forward path used, is a function of the error position probability,  $P_e$  (ie., it is the amount of the forward time,  $t_0$  achieved before an error occurs). It is unlikely that this relationship would be linear. The one shown in Fig. 3.9 (b) is raised to some predetermined power,  $\psi$ .



(a)  $t_f = \alpha + \beta t_1$

(b)  $t_1 = t_0 P_e^\psi$

Figure 3.9. - Functions and Relationships.

$P_e \in [0,1]$

So,

$$\begin{aligned}
 t_{av(c)} &= \frac{t_0 P_0 + t_0 P_f (1+\beta) P_e^\psi + \alpha P_f}{P_0} \\
 &= t_0 + (t_0 (1+\beta) P_e^\psi + \alpha) P_f / P_0 \qquad (3.9)
 \end{aligned}$$

Now suppose  $\alpha = 0, \beta = 1$ .

$$\text{then } t_{av}(c) = t_0 + \frac{2(1-P_0)P_e^\psi t_0}{P_0}$$

Now,  $P_e$  is the error position probability, ie. if  $P_e = 0.5$  then the average position at which an error may occur during the forward path is half way through (at  $t_0/2$ ).

Let us once again compare the continuous model with the discrete model.

$$t_{av}(c) = \frac{t_0 P_0 + 2(1-P_0)P_e^\psi t_0}{P_0}$$

$$t_{av}(d) = \frac{t_0 + t_f(1-P_0)}{P_0}$$

Therefore

$$t_0 + t_f(1-P_0) = t_0 P_0 + 2(1-P_0)P_e^\psi t_0$$

subtracting  $P_0$  from both sides.

$$1 + (1-P_0)^\psi t_f/t_0 = P_0 + 2(1-P_0)P_e^\psi$$

$$(1-P_0) + (1-P_0)^\psi t_f/t_0 = 2(1-P_0)P_e^\psi$$

dividing by  $(1 - P_0)$  throughout

$$1 + t_f/t_0 = 2P_e^\psi$$

Now if  $t_f = t_0$  (as in the discrete model), then:

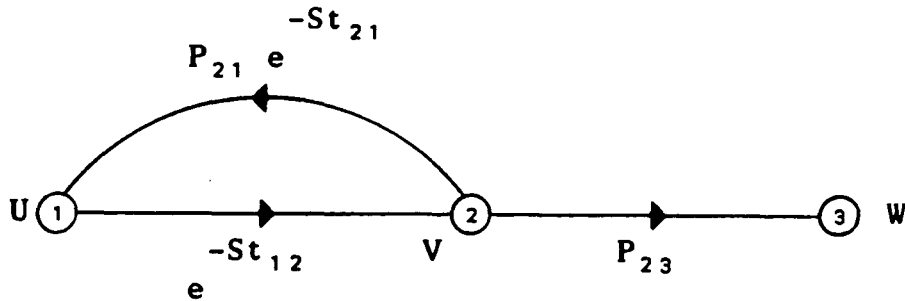
$$P_e^\psi = 1 \quad \text{ie. } \psi = 0$$

So as long as  $\psi > 0$  and  $P_e < 1$ , the continuous implementation will operate faster than that of the discrete. That is to say, in all cases except where the error correction does not take place until the end of the forward operation - which is by definition, the discrete model!

### 3.7 Laplace Transforms.

Similar to the Z transform introduced previously, the Laplace transform can be used to find the system transfer function of a network. Pritsker [Pritsker, 1966] uses this method exclusively rather than the Z transform, as it is more applicable to continuous time systems.

For example, returning to our original simple network, consisting of a single error recovery loop, but substituting the Laplace for the Z operator:



$$\text{transfer function TF}(S) = \frac{W(S)}{U(S)} = \frac{P_{23} e^{-St_{12}}}{1 - P_{21} e^{-S(t_{12}+t_{21})}}$$

differentiating with respect to S:

$$\frac{\partial \text{TF}(S)}{\partial S} = \frac{P_{23} e^{-St_{12}} (t_{12}+t_{21}) P_{21} e^{-S(t_{12}+t_{21})} - 1}{(1 - P_{21} e^{-S(t_{12}+t_{21})})^2} + \frac{(1 - P_{21} e^{-S(t_{12}+t_{21})}) t_{12} P_{23} e^{-St_{12}-1}}{(1 - P_{21} e^{-S(t_{12}+t_{21})})^2}$$

$$\left. \frac{\partial \text{TF}(S)}{\partial S} \right|_{S=0} = \frac{P_{12} P_{21} (t_{12} + t_{21}) + P_{23} t_{12} (1 - P_{21})}{(1 - P_{21})^2}$$

$$= \frac{t_{12} + t_{21}P_{21}}{P_{23}} \quad (3.10)$$

$$= t_{av} \quad - \quad \text{as before.}$$

The only real difference between using the Laplace and Z operators, up to this point, is that we set  $S = 0$  whereas using the Z transform we would set  $Z = 1$  to find the average network time.

### 3.7.1 Topology.

Using the Laplace transform we can call on the usual flowgraph techniques to find the network transfer function. ie., Mason's theorem:

$$TF(S) = \frac{\sum_i (PATH_i) [1 + \sum_m (-1)^m (\text{ORDER } m \text{ LOOPS NOT TOUCHING } PATH_i)]}{[1 + \sum_m (-1)^m (\text{LOOPS OF ORDER } m)]}$$

Now if we connect the output of a network directly to its input we get the topological equation  $H(S)$  [Pritsker, 1966].

$$H(S) = 1 - TF(S)W(S) = 0 \quad (3.11)$$

$$\text{where: } W(S) = 1/TF(S)$$

$$\text{and } TF(S) = \frac{1}{W(S)} = \frac{-\partial H(S)/\partial W}{H(S)|_{W=0}} \quad (3.12)$$

EXAMPLE:

Given the single error recovery loop of figure 3.10 where the output is joined to the input via path  $W(S)$ .

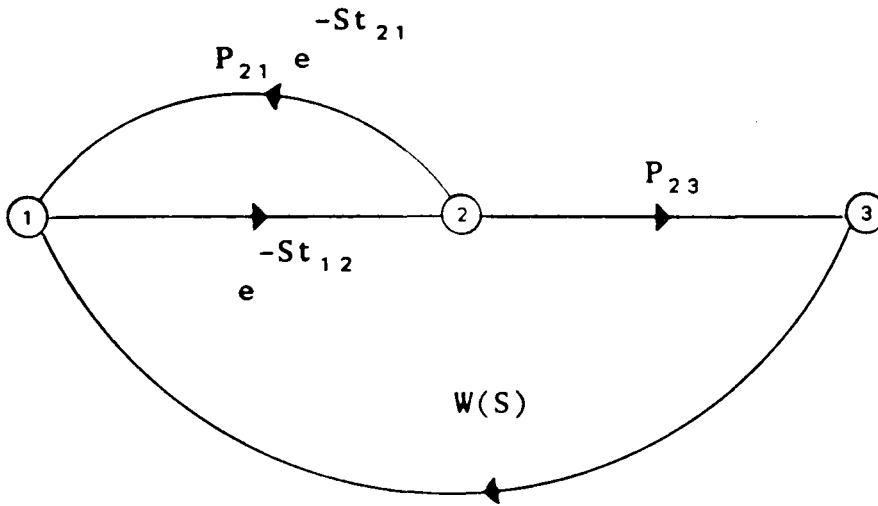


Figure 3.10 - Closed Network

Using (3.11), the topological equation:

$$H(S) - 1 - P_{21} e^{-S(t_{21}+t_{12})} - P_{23} e^{-St_{12}} W(S) = 0$$

$$\frac{\partial H(S)}{\partial W(S)} = - P_{23} e^{-St_{12}}$$

Now the transfer function can be found from (3.12)

$$-\frac{\partial H(S)}{\partial W(S)} \bigg|_{W(S)=0} = \frac{P_{23} e^{-St_{12}}}{1 - P_{21} e^{-S(t_{21}+t_{12})}} = TF(S)$$

### 3.7.2. Variance.

As shown in previous chapters, and repeated in the last section of this chapter, the average time (cost) can be found from the first derivative of the network transfer function. The higher derivatives yield equally useful results. The variance being the square of the first derivative subtracted from the second derivative, as in {3.13}.

$$\sigma^2 = U_2 - U_1^2 \quad (3.13)$$

where:

$$U_n = \frac{\partial^n}{\partial s^n} \left[ \frac{TF(s)}{TF(0)} \right]_{s=0} \quad (3.14)$$

#### EXAMPLE 1

The following simple example illustrates the use of equations {3.13} and {3.14} for finding the variance (and therefore standard deviation) of a simple network.

Figure 3.11 shows a very simple network consisting of two parallel forward paths. From inspection of figure 3.11, given say 10 items fed sequentially into the input, according to the respective probabilities, one would expect 7 items to take 3 time units and the other 3 items to take 4 time units.

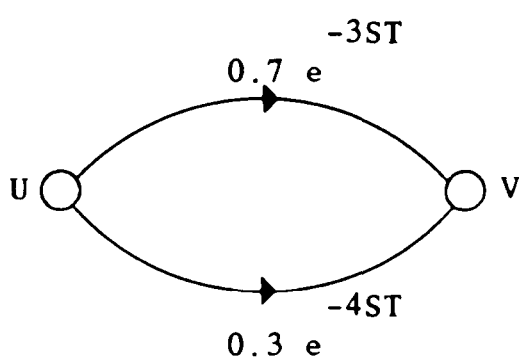


Figure 3.11 - Two Parallel Paths.

This gives a time average

$$t_{av} = \frac{7 \times 3 + 3 \times 4}{10} = 3.3$$

and variance

$$\sigma^2 = \frac{\sum(t_i - t_{av})^2}{n} = \frac{2.1}{10} = 0.21$$

The time average is found from the transfer function as usual.

$$TF(S) = 0.7 e^{-3ST-1} + 0.3 e^{-4ST-1}$$

$$TF(0) = 1$$

$$U_1 = \frac{\partial}{\partial S} \left[ \frac{TF(S)}{TF(0)} \right]_{S=0} = 2.1 e^{-3ST-1} + 1.2 e^{-4ST-1} = 3.3$$

$$U_2 = \frac{\partial^2}{\partial S^2} \left[ \frac{TF(S)}{TF(0)} \right]_{S=0} = 6.3 e^{-3ST-2} + 4.8 e^{-4ST-2} = 11.1$$

Now, using (3.13):

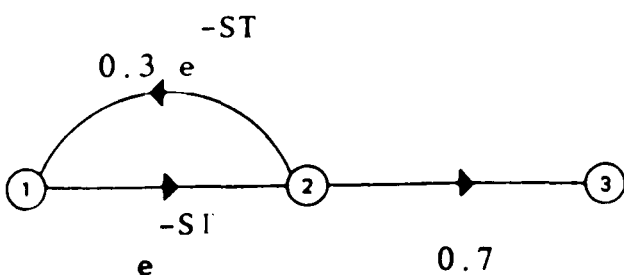
$$\text{Variance, } \sigma^2 = 11.1 - 3.3^2 = 0.21$$

Exactly as before.

If instead of the Laplace, the Z transform were used, the indices would be reduced on differentiating thereby giving an incorrect value for  $U_2$ . This is because the Z transform is a discrete operator whilst the Laplace is continuous. This is important if such results as variance are calculated using standard differential calculus.

### EXAMPLE 2

Consider once again, the basic error recovery loop:



$$U = \frac{0.7 e^{-ST}}{1 - 0.3 e^{-2ST}}$$

Using {3.14} to find the first and second derivatives of the transfer function U.

$$\frac{\partial U}{\partial s} = \frac{0.21 e^{-3sT-1} + 0.7 e^{-sT-1}}{(1 - 0.3 e^{-2sT})^2}$$

$$\left. \frac{\partial U}{\partial s} \right|_{s=0} = 1.857$$

$$\frac{\partial^2 U}{\partial s^2} = \frac{0.252 e^{-5sT-2} + 0.84 e^{-3sT-2}}{(1 - 0.3 e^{-2sT})^3} + \frac{0.63 e^{-3sT-2} + 0.7 e^{-sT-2}}{(1 - 0.3 e^{-2sT})^2}$$

$$\left. \frac{\partial^2 U}{\partial s^2} \right|_{s=0} = 5.898$$

According to (3.13)

$$\text{Variance, } \sigma^2 = \frac{\partial^2 U}{\partial s^2} - \left[ \frac{\partial U}{\partial s} \right]^2 = 2.448$$

This gives a standard deviation of approximately 1.56.

What this signifies is that, given a unit entering the flowgraph, the fastest time in which an output may physically occur is one time unit. However, the probability is that it will take an average of 1.86 time units, give or take 0.78.



### 3.8 AND Input Analysis

Everthing that is not forbidden is compulsory.

Murray Gell-Mann.

Petri nets have an AND input structure and so cannot be analysed in the same manner as signal flowgraphs which have EX-OR inputs. With GERT the choice of input logic functions is far greater than with other network representations and GERT networks can not only contain a combination of logic functions at the inputs but also deterministic and probabilistic output characteristics. This next section highlights some of the problems associated with AND type structures.

An example of what can happen when AND nodes are used is reproduced here [Whitehouse, 1973], though using the simpler Z transform, in figure 3.12.

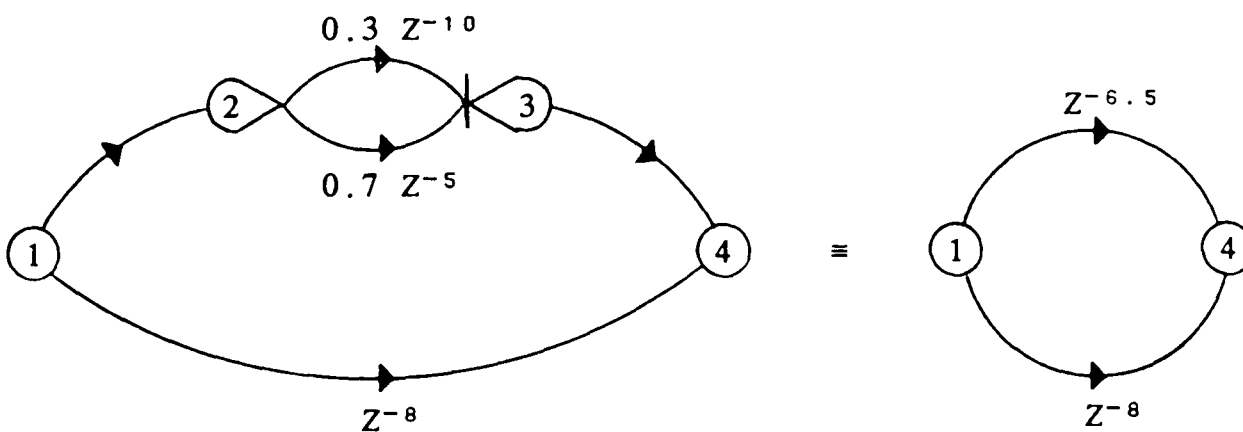


Figure 3.12 - GERT Flowgraph using EXOR and AND nodes.

The transfer function of Figure 3.12 reduces to:

$$TF(Z) = (0.3 Z^{-10} + 0.7 Z^{-5}) \cap Z^{-8}$$

$$\frac{\partial TF(Z)}{\partial Z} \Big|_{Z=1} = (3 + 3.5) \cap 8$$

$$= 6.5 \cap 8 = 8 \text{ time units.}$$

But one path on the upper side has a time of 10 units which is greater than the lower path of 8 units. Clearly we cannot use AND input nodes in this manner.

However, if we allow the output of node 3 to be probabilistic rather than deterministic as given in Figure 3.12, then:

$$TF(Z) = (Z^{-8} \cap (0.3 Z^{-10} \mid \cap 0.7 Z^{-5})) \tag{3.15}$$

where EXOR is given by the symbol  $\mid \cap$

from De Morgans theory [Bajpai et al, 1980 (p45)]:

$$\alpha \mid \cap \beta = \overline{(\alpha \cap \beta)} \cap (\alpha \cup \beta) \tag{3.16}$$

So using {3.16} on {3.15} we get:

$$TF(Z) = Z^{-8} \cap (\overline{(0.3 Z^{-10} \cap 0.7 Z^{-5})} \cap (0.3 Z^{-10} \cup 0.7 Z^{-5}))$$

$$\text{where } \overline{0.3 Z^{-10} \cap 0.7 Z^{-5}} = \overline{0.3 Z^{-10}} \cup \overline{0.7 Z^{-5}} = 0.3 Z^{-10}$$

$$\text{therefore } TF(Z) = Z^{-8} \cap 0.3 Z^{-10} \tag{3.17}$$

At first sight this may appear to give a transfer function with an overall probability of the final node being realised of greater than unity! But it must be remembered that both sides of the AND equation {3.17} must be realized simultaneously. Consequently there is a probability of 0.3 that the longer time (10 units) will be required, otherwise the shortest time (8 units) will be needed.

So  $TF(Z)$  becomes:  $0.7 Z^{-8} + 0.3 Z^{-10}$

Differentiating and setting  $Z = 1$  gives an average time of 8.6 time units.

From this exercise we can conclude that: nodes with deterministic outputs can be made to drive nodes with probabilistic outputs. However, the reverse is not possible. That is to say, more information must be provided in the output data than a single logic level if that result is derived from anything other than a single logic level.

### 3.8.1 Logic Functions & Distributions.

The fact that most network events behave like probability distributions rather than simple logic functions gives rise to some interesting results. Given that these events are time dependant, Bell [Bell, 1971 (p35)] defines the following AND and OR logic conventions:

$$X(\text{AND}) = \text{MAX}(X1, X2)$$

$$P(X(\text{AND}) < T) = P(X1 < T) P(X2 < T)$$

and

$$X(\text{OR}) = \text{MIN}(X1, X2)$$

$$P(X(\text{OR}) < T) = 1 - P(X1 > T) P(X2 > T)$$

$$= 1 - (1 - P(X1 < T)) (1 - P(X2 < T))$$

$$= P(X1 < T) + P(X2 < T) - P(X1 < T) P(X2 < T)$$

Now substituting density functions:

$$F(\text{AND}(T)) = F1(T) F2(T)$$

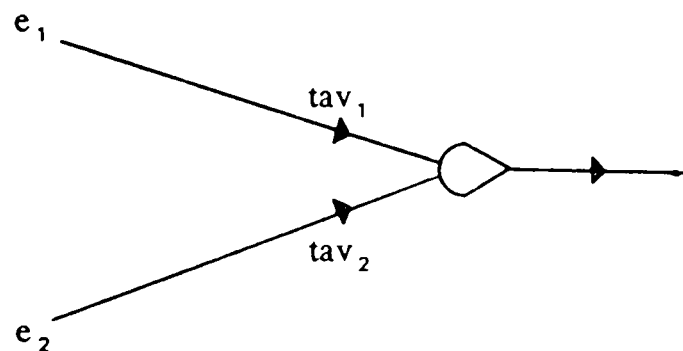
$$F(\text{OR}(T)) = F1(T) + F2(T) - F1(T) F2(T)$$

Which are the functions used if normal additive and multiplicative operations are carried out on simple binary logic levels as given in the truth table of table 3.2.

Table 3.2 - Logic Functions.

A	B	A+B	AB (AND)	A+B-AB (OR)
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	2	1	1

Now, if we have two events  $e_1$  and  $e_2$ , triggered simultaneously:



Their corresponding time averages  $tav_1$  and  $tav_2$ , will have distributions  $F(tav_1)$  and  $F(tav_2)$  respectively. These functions are shown graphically in figure 3.13.

$tav_2 > tav_1$  always.

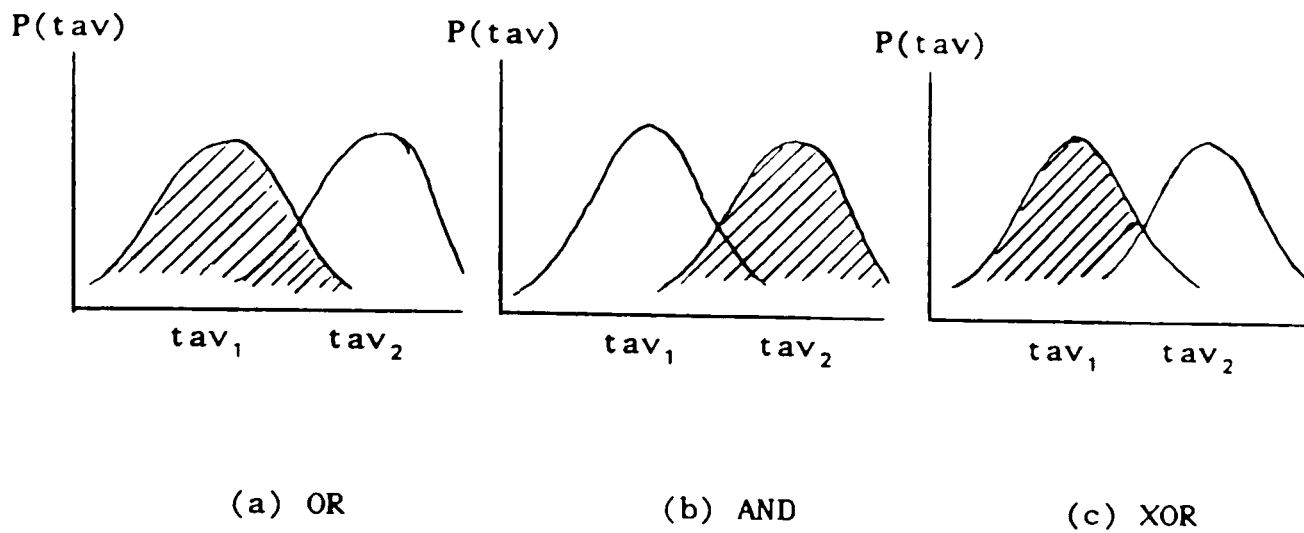


Figure 3.13 – Graphical Representation of Logic Functions.

Note, that unlike electronic logic gates, GERT nodes, like Petri nets, operate on a trigger basis. That is to say, even if the distribution shown in figure 3.13 are completely separated in time, once the node has been triggered by event 1 then all it requires is the receipt of event 2 to complete the required conditions for realising that node.

### 3.9 Summary

Clearly, continuous sensing will result in improvements in operating times compared with systems employing only discrete sensing. In fact, as this chapter shows, the worst implementation of continuous sensing can only be as poor as its discrete counterpart – it cannot be worse, and with correct attention to its installation should show some improvement in the economy of operating time. This naturally assumes that the practical configuration does not introduce superfluous delays not present in the discrete model, ie. continuous or too frequent interrogating of sensors instead of interrupt mechanisms or properly scheduled sensing.

If an interrupt mechanism is to be used to provide the sensor output data directly, that is to say the sensor output is the same signal which causes the interrupt, then the sensor must be perfectly reliable. A safer configuration is to restrict the interrupt signal to instigating the execution of an interrogation routine. This way other sensors, as well as the one causing the interrupt, can be checked. The need to check a single sensor many times to achieve a consensus is rather cumbersome and suggests either the need for a more reliable sensor or a greater number of sensors.

Using the formulae presented here, expected time averages for robot workcell operations may be accurately calculated. Alternatively, given a collection of operating statistics gathered over a period of time, the relevant distribution parameters can be calculated.

An insight into some of the pitfalls of using nodes with mixed logic characteristics has been discussed and the relevance of the corresponding probability distribution, as well as the absolute probability itself, highlighted. This has provided at least a partial solution to the GERT AND analysis problem.

## 4. MATRIX ANALYSIS

Science is nothing but developed perception, interpreted intent, common-sense rounded out and minutely articulated. It is therefore as much an instinctive product, as much a stepping forth of human courage in the dark, as is any inevitable dream or impulse action.

George Santayana, *Relativity of Science*.

This chapter deals exclusively with the different forms of matrix representation of flowgraphs and their relevant properties with regard to robot workcell modelling and analysis.

Where discrepancies exist between previously presented notations given in the references, definitions are provided which are most relevant to this particular work.

### 4.1 Adjacency Matrices.

Sometimes called the vertex matrix, though the matrix elements represent arcs rather than vertices, the adjacency matrix is really only applicable to digraphs. In the case of simple (non directed) graphs the connectivity matrix is used [Tutte, 1966].

With regard to digraphs there appears to be some ambiguity about the term 'strongly connected'. Harary & Palmer [Harary and Palmer, 1973 (p126)] define a digraph as strongly connected 'if every pair of points is mutually reachable by directed paths'. Busacker and Saaty [Busacker & Saaty, 1965 (p28)] give a similar definition. However, Wilson [Wilson, 1979 (p101)] extends the definition to include certain cases of one way paths between nodes as do Berman & Plemmons when discussing the reducibility of a digraph [Berman & Plemmons, 1979 (p31)]. For the purposes of this work a strongly connected digraph is defined as: that whose adjacency matrix is square and irreducible, ie. it has all non-zero elements with the optional exception of the main diagonal (reflexive or irreflexive). In either case, all nodes are mutually reachable from all other nodes by no more than one path traversal.

Again using the convention that the element  $A_{ij}$  represents the path from node  $i$  to node  $j$ , we are now in a position to make a few statements with regard to the general properties of adjacency matrices.

- a) All finite elements along the main diagonal represent self loops (ie. loops whose single path is both incident to and incident from the same node).
- b) An adjacency matrix  $A$  in strictly upper triangular form represents an acyclic digraph with only forward paths.  $A^n = 0$  for some  $n > 2$ .
- c) An adjacency matrix in strictly lower triangular form represents a digraph with only feedback paths.  $A^n = 0$  for some  $n > 2$ .



#### 4.1.1 Boolean Adjacency Matrices.

The Boolean adjacency matrix is the same as the adjacency matrix mentioned in section 4.1 except that all elements are Boolean (have value 1 or 0). It is sometimes known as a permutation matrix [Rice, 1981].

Carre gives us the following definition for a Boolean adjacency matrix  $A$  representing a digraph  $G = (x, U)$ :

$$a_{i,j} = \begin{cases} 1 & \text{if } (x_i, x_j) \in U \\ 0 & \text{if } (x_i, x_j) \notin U \end{cases}$$

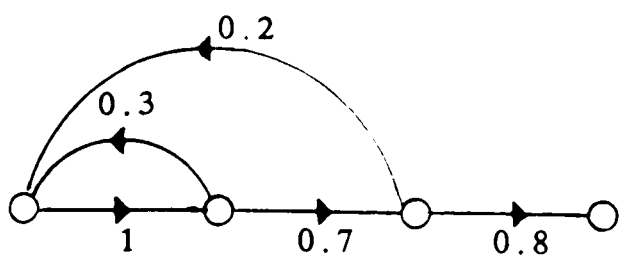
The Boolean adjacency matrix is used by Hsu [Hsu, 1975] to find the minimum equivalent graph (MEG) of an acyclic flowgraph. Unfortunately this technique is not usable in the same manner with non-Boolean adjacency matrices and so is of limited use.

#### 4.1.2 Transition Matrices.

This is yet another manifestation of the adjacency matrix. Only this time, unlike the Boolean adjacency matrix, each element represents the probability of transition from one node to the next. Using Carre's notation, we can define a transition matrix  $P$  representing a digraph  $G = (x, U)$  as:

$$P_{i,j} = \begin{cases} p & \text{if } (x_i, x_j) \in U \quad ; \quad p \in ]0, 1] \\ 0 & \text{if } (x_i, x_j) \notin U \end{cases}$$

An example of a transition matrix is given in Figure 4.1 where a simple digraph having two feedback paths is depicted together with its corresponding transition matrix.

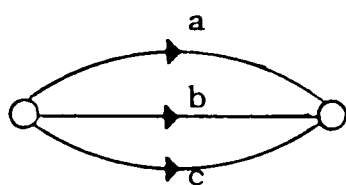


$$P_t = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0.3 & 0 & 0.7 & 0 \\ 0.2 & 0 & 0 & 0.8 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 4.1 - The Transition Matrix.

The transition matrix  $P_t$  is always shown as a square matrix even though there are no non-zero elements in the fourth row. The reasons for this will become apparent later.

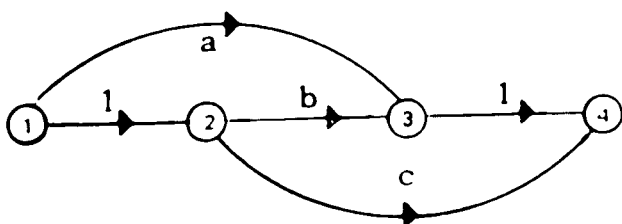
The parallel paths between two nodes, first mentioned in section 2.2.1, would be most difficult to implement in transition matrix form as they stand. For example, figure 4.2 shows three parallel paths between two nodes and the corresponding 2 by 2 transition matrix.



$$\begin{bmatrix} 0 & f(a,b,c) \\ 0 & 0 \end{bmatrix}$$

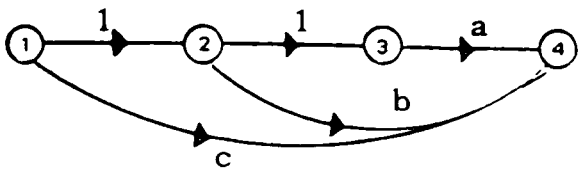
Figure 4.2 - Three Parallel Paths.

Clearly, this is impossible to implement as a 2 by 2 transition matrix without explicitly defining the function  $f(a,b,c)$ . The solution is to put the flowgraph into a different form. Figures 4.3 and 4.4 show the flowgraphs and transition matrices for the cascade and canonical forms respectively.



$$\begin{bmatrix} 0 & 1 & a & 0 \\ 0 & 0 & b & c \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

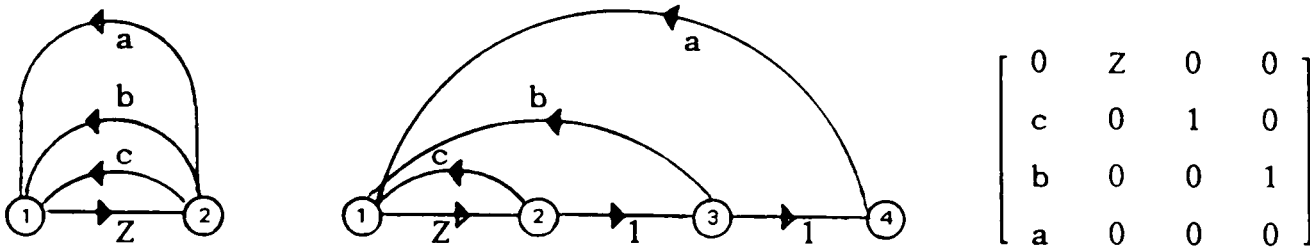
Figure 4.3 - Cascade Form.



$$\begin{bmatrix} 0 & 1 & 0 & c \\ 0 & 0 & 1 & b \\ 0 & 0 & 0 & a \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Figure 4.4 - Canonical Form

Similarly, figure 4.5 shows the canonical form for multiple feedback loops between two nodes.



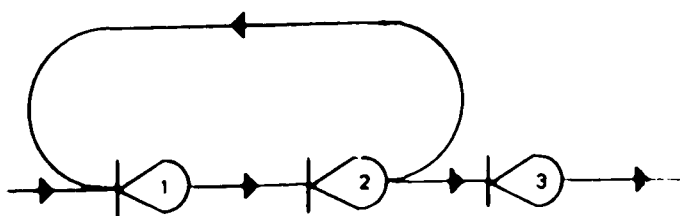
$$\begin{bmatrix} 0 & Z & 0 & 0 \\ c & 0 & 1 & 0 \\ b & 0 & 0 & 1 \\ a & 0 & 0 & 0 \end{bmatrix}$$

Figure 4.5 - Canonical form for feedback loops

Needless to say, this introduces another two nodes and two further paths which may appear superfluous. However, it will become clear later why this is necessary when any form of mathematical analysis is to be performed on transition matrices.

#### 4.2 The Realization Matrix.

Using the GERT notation (chapter 2) Bell [Bell, 1971 (p13)] introduces the concept of the realization matrix. This is a form of Boolean matrix which not only contains the reachable states but also contains information about their reachability. Referring back to the simple error recovery loop example of chapter 2, figure 4.5 shows the realization matrix, which is somewhat like a truth table, for this digraph.



$\begin{array}{c cccccccc} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 2 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 3 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ \hline 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$	$\begin{array}{c cccc} 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 \\ 3 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \end{array}$	$\begin{array}{c cccccccc} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 2 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 3 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ \hline 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 \end{array}$
--	--	--

a)

b)

c)

Figure 4.5 - 3 Stages of the Realization Matrix.

Figure 4.5 (a) is the realization matrix assuming single entry only, ie., only one item in the system at any one time. From this it can be seen that two or more nodes cannot be active at the same time therefore all states other than 000, 100, 010 and 001 are impossible, giving the reduced matrix of 4.5 (b). This shows clearly that exactly two states can be reached from any state. The multiple entry matrix of 4.5 (c) is somewhat more complex. From some states up to four other states can be reached.

Comparing the realization matrices of figure 4.5 (b) and (c), those states which exist in (c) but not in (b) represent those which are capable of resulting in simultaneous realization. This is important in practical multiple robot workcell implementations where simultaneous realization may mean collision!

In a GERT system, if only EXOR nodes are employed then the network represents a flowgraph in which simultaneous realization is impossible - assuming that the nodes are practically (as well as theoretically) EXOR.

### 4.3 The Incidence Matrix.

The incidence matrix provides a convenient method of representing flows in and out of the nodes of a digraph. If a path is incident from a node its corresponding incidence matrix element has value 1, if incident to the node it has value -1, and if no incidence it has value 0.

Adopting Carre's notation again, for a digraph  $G = (X, U)$  the incidence matrix  $S$  is defined as:

$$S_{ij} = \begin{cases} +1 & \text{if } u_j \text{ is incident from } x_i \\ -1 & \text{if } u_j \text{ is incident to } x_i \\ 0 & \text{if } u_j \text{ is not incident with } x_i \end{cases}$$

This method of representation of a digraph requires the labelling of both arcs and vertices. Consequently the resulting incidence matrix will not be square. Attempts have been made to use the incidence matrix to represent Petri nets [Harhalakis et al, 1989] where  $U$  denotes the set of states (circles in figure 2.10) and  $X$  denotes the set of transitions (bars in figure 2.10). However, closer inspection of figure 2.10 reveals a 'dummy' transition which would result in an element of any incidence matrix  $S$  having both values of +1 and -1 simultaneously. Clearly, this severely limits the usage of such representations.

#### 4.4. Matrix Analysis

If God has made the world a perfect mechanism, He has at least conceded so much to our imperfect intellect that in order to predict little parts of it, we need not solve innumerable differential equations, but can use dice with fair success.

Max Born.

Having introduced a selection of useful matrix representations of flowgraphs, we now conduct an analysis with particular regard to stochastic transition matrices.

##### 4.4.1 Flows.

It is not the significant event which ought to be wondered at, but rather the frequent recurrence of similar instances.

Gerolamo Cardano (1501-1576). From *De Vita Propria Liber* (the book of my life) 1576.

In a multiple node flowgraph each node may be represented by a discrete sense point. In the case of cyclic graphs, some nodes are going to experience a greater degree of activity (flow or transmission) than others. It then follows that, in a multiple entry system (as mentioned in 4.5), it is often more efficient to interrogate sensors at a high flow node more frequently than those at a low flow node. This is illustrated in figure 4.6.

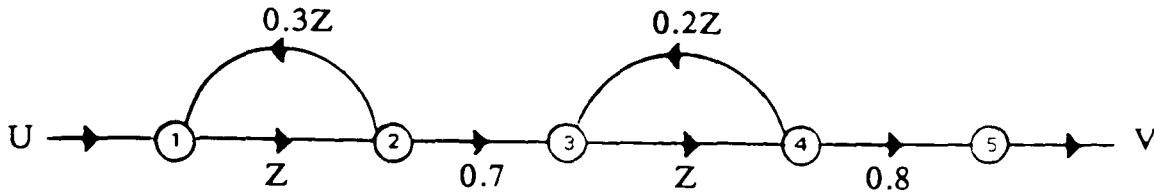


Figure 4.6 - Flowgraph showing Node Flows.

For the flowgraph of figure 4.6 the flows at each node can easily be determined:

The first node,  $n_1 = U + 0.3Z^2n_1$

Let  $U = 1$ ;

then  $n_1 = 1/(1 - 0.3Z^2)$

Setting  $Z = 1$  gives:  $n_1 \Big|_{Z=1} = \frac{1}{1 - 0.3} = 1.43$

Similarly for node  $n_2$ :

$$n_2 = Zn_1 = Z/(1 - 0.3Z^2) \quad \text{and} \quad n_2 \Big|_{Z=1} = 1.43$$

Node  $n_3 = 0.7n_2 + 0.2Z^2n_3$

hence  $n_3 = \frac{0.7n_2}{1 - 0.2Z^2} = \frac{0.7Z}{(1-0.2Z^2)(1-0.3Z^2)} \quad \text{and} \quad n_3 \Big|_{Z=1} = 1.25$

Likewise:

Node  $n_4 = Zn_3 = \frac{0.7Z^2}{(1-0.2Z^2)(1-0.3Z^2)} \quad \text{and} \quad n_4 \Big|_{Z=1} = 1.25$

and finally;

Node  $n_5 = 0.8n_4 \quad \text{hence} \quad n_5 \Big|_{Z=1} = 0.8 \times 1.25 = 1$

So, the flows for nodes 1 to 5 are 1.43, 1.43, 1.25, 1.25 and 1 respectively for a normalized input value of  $U = 1$ . By inspection of the digraph of figure 4.6 it can be seen that these values are of an order one would expect given the relative tolls and probabilities. Clearly, what passes through node  $n_1$  must also at some time pass through node  $n_2$  as these both share the same flowgraph loop, consequently the flows through each of these nodes are equal. This also holds for nodes  $n_3$  and  $n_4$ , though of course the value of the flows are lower owing to the smaller probability of repeat at this loop. Node  $n_5$  has a flow of unity which is not surprising as the normalised input value of  $U$  is unity. The laws of conservation must apply to all stochastic digraphs which do not have absorbing nodes, but more of this when we deal with Markov chains later.

#### 4.4.2 Sensor Implementation.

What is most interesting is the fact that the flow at each node expressed as a fraction of the sum of the flows at all the nodes gives us the relative activity at that node. That is to say if we divide the flow at each node by the sum of all the flows we can find the percentage activity at each node. For the example of figure 4.5 these values are approximately 22.5% for nodes  $n_1$  and  $n_2$ , 19.7% for nodes  $n_3$  and  $n_4$  and 15.6% for node  $n_5$ .

If we have a sensor at each of the nodes in the above example, then the sensors at nodes  $n_1$  and  $n_2$  will be in use more often than those at nodes  $n_3$  and  $n_4$  which will in turn be used more than that at node  $n_5$ . The fact that the flow at node  $n_1$  is equal to that at  $n_2$  suggests that only one sensor is required between the two. In fact as no decision is made at node  $n_1$ , only  $n_2$  requires a sensor. Consequently we can reduce the number of sensors which would appear to be required from 5 to 3. This may seem obvious as it is not usual to sense for errors before an operation. However, as work cells, and consequently flowgraphs, become larger this may not be so immediately clear.



The usual method of signal I/O employed by most robot controllers is polling. This is a timeshare technique which involves the interrogation of each of the input lines, connected to the sensors, in turn. According to Shan "polling is a source of unnecessary performance loss". Polling loops must always be active and polling can miss state changes if they are not present at the right place [Shan, 1989]. Consequently, if we are to interrogate such sensors from a single computer, or robot cell controller, where each of a number of I/O lines are polled in turn, then it would be most economic to distribute the polling times in accordance with the relative activity at each node. Now in most computer I/O systems it is not practicable to alter the amount of time spent at each line. However, it is very often the case that more I/O lines are available than are actually required. For example in the case of a Puma 500 [Unimation, 1985] there can be up to 64 input lines. If we are using only three sensors then it would be sensible to distribute the lines according to the ratios of nodal activity.

Given 32 such lines and 3 sensors with the corresponding activity percentages of 45%, 39.4% and 15.6% for each of the three nodes it would be most efficient to share the lines accordingly, ie., 28 lines for the sensor  $s_1$  at node  $n_2$ , 26 for sensor  $s_2$  at node  $n_4$  and 10 for  $s_3$  at  $n_5$ . This gives a greater probability of being able to act as soon as a sensor change occurs. The alternative simple method would be to allocate one line to each of the three sensors thereby utilising only 4.7% of the available I/O time. This means that for 95.3% of the time the computer is looking at nothing!

Perhaps more important over a long period of usage is the 'wear and tear' to various parts of the work cell. Those used more often are more likely to be subject to greater strain and hence more frequent failure. The expectation of such 'high stress' points, together with possible bottlenecks, is emphasised by the network flows.

### 4.4.3 Matrix Formulation.

In chapter 3 we found the transfer function of a flowgraph by using Mason's theorem or other flowgraph reduction techniques. This was then differentiated with respect to  $Z$  and the resulting time average calculated from this expression with  $Z$  set to unity. Likewise, but with  $S$  set to zero if the Laplace transform of chapter 3 were used.

Here the flowgraph will be put into transition matrix form before performing the necessary algebraic manipulations. We will start with our usual single error recovery loop of figure 4.7.

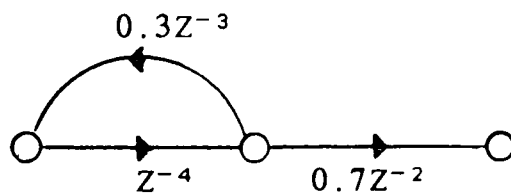


Figure 4.7 - Simple Error Recovery Loop.

$$\text{The transition matrix } A = \begin{bmatrix} 0 & Z^{-4} & 0 \\ 0.3Z^{-3} & 0 & 0.7Z^{-2} \\ 0 & 0 & 0 \end{bmatrix} \quad (4.1)$$

Returning to the use of the power series of {3.1}, the matrix  $A$  can be dealt with in the same manner [Howard, 1971].

$$B = [I - A]^{-1} = \frac{1}{1 - 0.3Z^{-7}} \begin{bmatrix} 1 & Z^{-4} & 0.7Z^{-6} \\ 0.3Z^{-3} & 1 & 0.7Z^{-2} \\ 0 & 0 & 1 - 0.3Z^{-7} \end{bmatrix} \quad (4.2)$$

$$\text{Transfer function } B_{1,3} = \frac{0.7Z^{-6}}{1 - 0.3Z^{-7}} \quad (4.3)$$

This process is relatively straight-forward when done manually. Even large matrices can be inverted without too much hard work by using recently devised techniques [Jeter et al, 1987]. Many forms of computer algorithm exist as standard methods for normal constant matrix inversion [Monro, 1983]. Unfortunately the matrix of {4.2}, like most of the matrices we will be using, is not a matrix of constant values but rather one of functions in Z. It is therefore essential that some other method be found for dealing with matrices containing functions.

#### 4.4.4 Matrix Derivatives.

We shall first state the general problem, and then solve it for a particular special case. In so doing we hope to illustrate the underlying concepts while simultaneously avoiding inessential detail. The success of the method in the special case then suggests its own generalization. It is interesting to observe that in the literature, for reasons of economy and mathematical "elegance", such problems are usually presented in the most general form that the authors can handle. Such presentation often obscures the process by which the solution was reached, leaving the reader with only the answers and undue respect for the intelligence of the author.

R.B. Cooper (Introduction to Queueing theory).

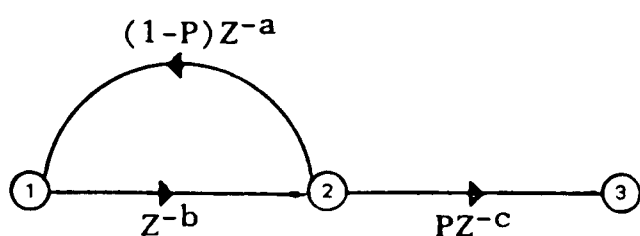
Normally the differentiation of a matrix of the form of {4.2} is quite straight forward. Brickell states:

For a general matrix  $\mathbf{B} = [b_{ij}]$  whose elements are functions of t, the derivative

$$\frac{d\mathbf{B}}{dt} = \left[ \frac{db_{ij}}{dt} \right] \quad (4.4)$$

can be found by simply differentiating each element with respect to  $t$  [Brickell, 1972]. For our purposes we simply substitute  $Z$  for  $t$ .

However, as pointed out in 4.4.3, solving this type of problem by digital computer is not so easy. To introduce a non-differential calculus method for achieving the same result, we must now take a closer look at the situation by returning to the example of our simple building block, the single feedback loop.



$$TF = \frac{PZ^{-(b+c)}}{1 - (1-P)Z^{-(a+b)}}$$

$$\left. \frac{dTF}{dZ} \right|_{Z=1} = \frac{P(1-P)(a+b) + P(1-(1-P))(b+c)}{(1-(1-P))^2}$$

$$= \frac{P(1-P)(a+b) + P^2(b+c)}{P^2}$$

$$= \frac{a+b}{P} - a + c \quad (4.5)$$

We must now split our  $Z$ -function matrix into two separate transition matrices so as to achieve the same result as {4.5} without actually differentiating the matrix using standard calculus.

The stochastic transition matrix:

$$A_P = \begin{bmatrix} 0 & 1 & 0 \\ 1-P & 0 & P \\ 0 & 0 & 0 \end{bmatrix}$$

and the toll matrix:

$$A_T = \begin{bmatrix} 0 & b & 0 \\ a & 0 & c \\ 0 & 0 & 0 \end{bmatrix}$$

Note that the toll matrix  $A_T$  is not a transition matrix in the same sense as the stochastic matrix  $A_P$ . This is because in  $A_T$  a path having no toll is represented as a zero in the same way as the absence of a path is denoted. The toll matrix is often simply a representation of inter-nodal time delay (or other cost).

Once again using (4.2)

$$[I - A_P]^{-1} = \frac{1}{P} \begin{bmatrix} 1 & 1 & P \\ 1-P & 1 & P \\ 0 & 0 & P \end{bmatrix}$$

Now introducing the technique of congruent matrix multiplication [Howard, 1971], which is the result of the multiplication of corresponding matrix elements, for which the operator  $\square$  is used.

$$\begin{aligned} A_T \square [I - A_P]^{-1} &= \begin{bmatrix} 0 & b & 0 \\ a & 0 & c \\ 0 & 0 & 0 \end{bmatrix} \square \begin{bmatrix} 1/P & 1/P & 1 \\ 1/P-1 & 1/P & 1 \\ 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & b/P & 0 \\ a/P-a & 0 & c \\ 0 & 0 & 0 \end{bmatrix} \end{aligned} \quad (4.6)$$

Now the sum of all the elements of {4.6} gives us the same result as {4.5}

$$\sum_j \sum_i (A_T \square [I - A_P]^{-1}) = \frac{a+b}{P} - a + c$$

Returning to the example given previously with the probability and toll values of {4.3}.

Differentiating according to the form of {4.4}

$$\frac{dB_{1,3}}{dZ} = \frac{4.2Z^{-7} + 0.21Z^{-14}}{(1 - 0.3Z^{-7})^2}$$

$$\left. \frac{dB_{1,3}}{dZ} \right|_{Z=1} = \frac{4.2 + 0.21}{0.49} = 9 = t_{av}$$

Again using the matrix method of {4.5}

$$A_T = \begin{bmatrix} 0 & 4 & 0 \\ 3 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix} \quad A_P = \begin{bmatrix} 0 & 1 & 0 \\ 0.3 & 0 & 0.7 \\ 0 & 0 & 0 \end{bmatrix}$$

$$A_T \square [I - A_P]^{-1} = \begin{bmatrix} 0 & 4 & 0 \\ 3 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix} \square \begin{bmatrix} 1.429 & 1.429 & 1 \\ 0.429 & 1.429 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\hat{=} \begin{bmatrix} 0 & 5.71 & 0 \\ 1.29 & 0 & 2 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\sum_j \sum_i (A_T \square [I - A_P]^{-1})_{ji} = 5.71 + 1.29 + 2 = 9 = t_{av}$$

Now this provides us with a new tool for achieving the numerical equivalent of differentiation of an error recovery loop transition matrix. Such a method is ideally suited to implementation on a digital computer, in fact most matrix manipulations including congruent matrix multiplication are available within many modern mathematical software packages such as MATLAB [Moler et al, 1986].

Care must be taken when using matrices of greater dimension than 3. To ensure the correct topology, buffer paths must often be included. This is investigated more thoroughly, with examples, in section 6.2.4.

#### 4.5 Markov Processes.

The Markov property was first introduced in chapter 3.2 where it was defined as the property of independence of past events. In the light of the other philosophies discussed so far we now return to the Markov property in investigating the matrix representation of digraphs. It should soon become apparent that the Markov property is one of the most powerful tools for dealing with problems concerning chains of stochastic events. Many texts exist on the principles of Markov processes and a basic knowledge would be useful as a prerequisite to the rest of this work. However, it should be possible to follow the main ideas without difficulty by reference to the Markov process nomenclature given in appendix C, or to one of the texts listed in the references, particularly those by Howard.

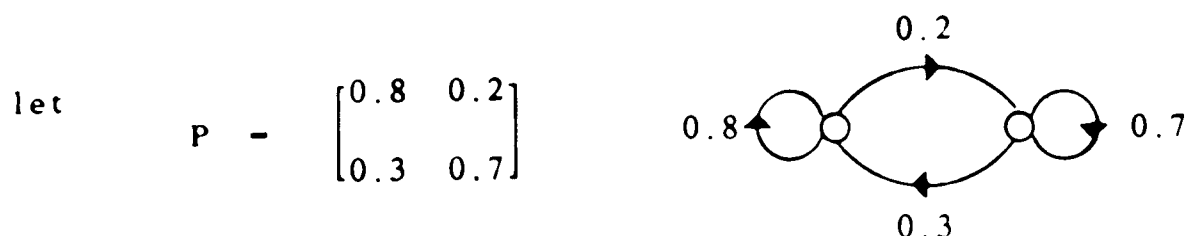
##### 4.5.1 The Limiting Matrix.

Given a transition matrix  $P$  representing a digraph, and a starting vector  $V_0$ , the next state in a process is given by:

$$V_1 = V_0 P \quad \text{or} \quad V_{i+1} = V_i P \quad \{4.7\}$$

and this is true for all  $i$  provided the Markov property holds.

For example;



if we start at node 1,  $V_0 = [1 \ 0]$

then the next state is  $V_1 = [1 \ 0] \begin{bmatrix} 0.8 & 0.2 \\ 0.3 & 0.7 \end{bmatrix} = [0.8 \ 0.2]$

and  $V_2 = V_1 P = [0.8 \ 0.2] \begin{bmatrix} 0.8 & 0.2 \\ 0.3 & 0.7 \end{bmatrix} = [0.7 \ 0.3]$

$V_3 = V_2 P = [0.65 \ 0.35]$

$V_4 = V_3 P = [0.625 \ 0.375]$  ...and so on.

alternatively, we could say:

$$V_n = V_0 P^n \quad (4.8)$$

After four iterations we have  $V_4 = [0.625 \ 0.375]$ . Clearly, the probability vector is converging. Howard [Howard (Vol 1), 1971] uses this example to show that the roots (Eigenvalues) of the transition matrix  $P$  are the probabilities given by  $V_n$  when  $n$  approaches infinity (the columns of  $P^\infty$ ).

$$\text{ie., } V = V P \quad (4.9)$$

$$\text{or } [v_1 \ v_2] = [v_1 \ v_2] \begin{bmatrix} 0.8 & 0.2 \\ 0.3 & 0.7 \end{bmatrix}$$

which reduces to  $v_1 = 0.6, v_2 = 0.4$

hence  $V_\infty = [0.6 \ 0.4]$

$$\text{or } P^\infty = \begin{bmatrix} 0.6 & 0.4 \\ 0.6 & 0.4 \end{bmatrix}$$

which is what {4.2} would eventually converge to with the above  $P$  and starting vector  $V_0$ .

$P^\infty$  is known as the limiting matrix of a Markov process, and represents the probabilities a process will converge to over a long period of time. In the well known example of 'tossing a fair coin', the limiting values would be 0.5 and 0.5.



Returning to our initial example of figure 4.1. Completing the loop with a path from nodes  $n_5$  to  $n_1$ , to produce a closed system (the reasons for this will become apparent in 4.5.3) and putting this flowgraph into transition matrix form:

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0.3 & 0 & 0.7 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0.2 & 0 & 0.8 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The limiting vector

$$V_0 P^\infty = [0.2247 \ 0.2247 \ 0.1966 \ 0.1966 \ 0.1574]$$

which is exactly our percentage flows found in 4.4.1 by simple (but lengthy) algebra.

The rate at which convergence occurs is determined by these probabilities. For two nodes connected by paths of unity probability, convergence will never be achieved, instead the process will continually oscillate. This is the only occasion a stochastic matrix can represent a system which is conditionally unstable. In all cases where the probabilities causing a cycle to exist are less than unity, eventual stability is guaranteed. Where such probabilities are relatively large, damped oscillation may occur and convergence may be slow but it will always take place. Howard gives a thorough analysis of the factors governing convergence in a markov process [Howard, 1971]. Another interpretation is the definition: the maximal eigenvalue  $e$ , of a stochastic matrix  $A$  is one, ie., if and only if  $e$  is an eigenvector of  $A$  corresponding to the eigenvalue one [Berman & Plemmons, 1979 (p49)].

Convergence is only possible if the transition matrix is regular. In fact, the definition of a regular stochastic matrix  $P$  is that all the entries of some power  $P^m$  are positive [Lipschutz, 1966]. For non-zero convergence to be possible  $P$  must be closed so as to represent a process which repeats itself infinitely.

This act of closing the loop does not effect the actual operation of the process as the feedback path is of unity probability and zero toll.

Calculating  $P^\infty$  is a simple task for small equations where the roots can be found easily or where convergence is rapid thereby allowing successive multiplication by computer. The alternatives usually involve finding the inverse of some matrix. ie.,

$$\text{Given that } P + P^2 + P^3 + \dots + P^\infty = P[I - P]^{-1}$$

multiplying through by  $P^{-1}$

$$I + P + P^2 + P^3 + \dots + P^{\infty-1} = [I - P]^{-1}$$

$$\begin{aligned} \therefore P^\infty &= P[I - P]^{-1} - [I - P]^{-1} + I \\ &= -[I - P][I - P]^{-1} + I = 0 \end{aligned}$$

Fortunately for a stochastic matrix  $P$ , this is only possible where  $[I - P]^{-1}$  exists - in all cases where  $P^\infty$  is not zero,  $[I - P]$  is singular! This can be seen from the above if one considers  $P^\infty$  (and hence all  $P^n$ , where  $n < \infty$ ) to be finite. In which case  $I + P + P^2 + P^3 + \dots$  will approach infinity. Consequently,  $P^\infty$  must be made to approach zero, ie.,  $P$  must be regular.

If the normalised Eigenvector matrix  $T$  of transition matrix  $P$  can be found, then:

$$D^n = \prod_{n=1}^{\infty} (T^{-1} P T) = T^{-1} P^n T$$

and

$$P^n = T D^n T^{-1}$$

A proof of this is given by Eisenman [Eisenman, 1963] together with an example. Also given is the following simple result:

For a two event Markov chain

$$P = \begin{bmatrix} a & 1-a \\ b & 1-b \end{bmatrix}$$

where:  $a < 1$ ;  $b < 1$ .

thus: 
$$\lim_{n \rightarrow \infty} \begin{bmatrix} a & 1-a \\ b & 1-b \end{bmatrix}^n = \frac{1}{b + 1 - a} \begin{bmatrix} b & 1-a \\ b & 1-a \end{bmatrix}$$

The situation becomes increasingly complicated for Markov chains representing 3 or more events.

#### 4.5.2 Ergodic Chains.

The definition of an ergodic state of a Markov chain is defined as a state which is both persistent and aperiodic. Wilson [Wilson, 1979] gives the following example:  $n$  people sat round a table throwing a dice. If the dice is 1, 3 or 5 the player must pass the dice one place to the left. If 2 or 4 then two places to the right. If a six is thrown, then the dice must be thrown by the same player again. This produces the flowgraph of figure 4.8 (a) with three players ( $n = 3$ ), and figure 4.8 (b) for four players ( $n = 4$ ), together with their resultant transition matrices.

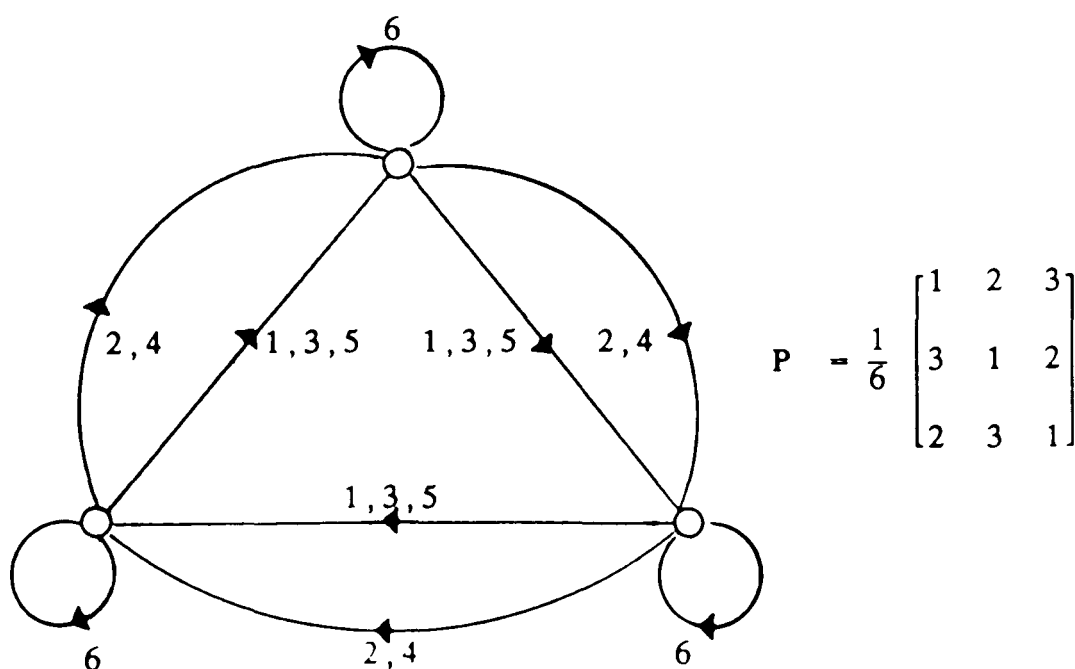


Figure 4.8 (a) - Process for  $n = 3$ .

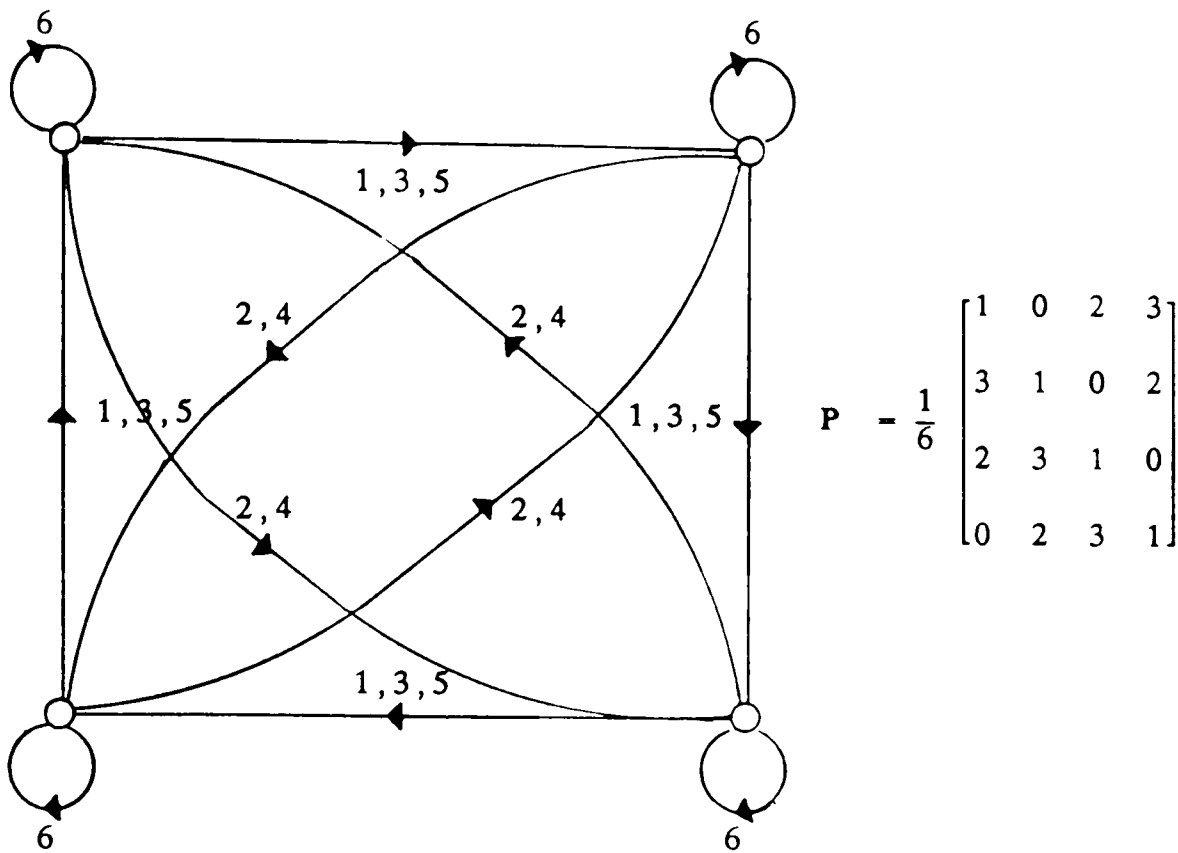


Figure 4.8 (b) - Process for n = 4.

Note that each diagonal of the transition matrix  $P$  contains the same  $P_{ij}$  values. According to Kelly, if the limiting matrix  $P^n$  converges as  $n$  approaches infinity, then the process is ergodic [Kelly, 1979]. If  $P^\infty$  has no zero entries, then this is only true for ergodic processes containing only one cyclic class, ie., regular Markov chains [Kemeny & Snell, 1965].

#### 4.5.3 Regular Chains.

A regular matrix is defined as: a non-negative square matrix  $P$  where there exists a natural number  $k$  such that  $P^k$  is a positive matrix [Iosifescu, 1980].

Romanovsky [Romanovsky, 1970 (p45)] with regard to the elements of the matrix  $P^n$  states: "when all of them are non-zero, the chain  $C_n$  is called POSITIVELY REGULAR, and COMPLETELY REGULAR, if, in addition, they are all equal to each other, and, consequently, equal to  $1/n$ ". Such a transition matrix  $P$  is said to be POSITIVELY REGULAR or COMPLETELY REGULAR accordingly.

The limiting matrix  $P^\infty$  of figure 4.8 (a) converges to zero and is therefore not regular. That of figure 4.8 (b), on the other hand, converges to a constant matrix with all elements equal to one quarter and is consequently completely regular.

In chapter 3 an example in section 3.7.1 was given in which the output of the network was fed directly back to the input to enable us to find the variance of the system. This was necessary to make the transition matrix REGULAR. It will now be made apparent why!

Using the example of fig. 4.6 we have the transition matrix  $P$ .

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0.3 & 0 & 0.7 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0.2 & 0 & 0.8 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The characteristic matrix

$$[I - P]^{-1} = \begin{bmatrix} 1.4286 & 1.4286 & 1.25 & 1.25 & 1 \\ 0.4286 & 1.4286 & 1.25 & 1.25 & 1 \\ 0 & 0 & 1.25 & 1.25 & 1 \\ 0 & 0 & 0.25 & 1.25 & 1 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

can be found easily.

However, the limiting matrix  $P^\infty \rightarrow 0$ . This is because the transition matrix  $P$  is not regular. To make  $P$  regular we must close the loop as was done in the example of 3.7.1 shown here in fig. 4.9.

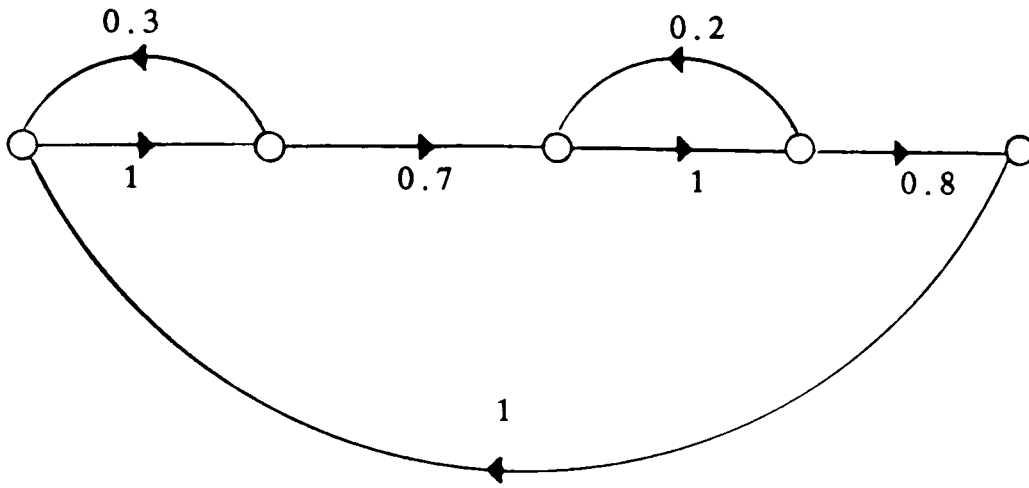


Figure 4.9 - Closed Double Loop Flowgraph.

Consequently

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0.3 & 0 & 0.7 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0.2 & 0 & 0.8 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

giving a limiting matrix

$$P^\infty = \begin{bmatrix} 0.2247 & 0.2247 & 0.1966 & 0.1966 & 0.1573 \\ 0.2247 & 0.2247 & 0.1966 & 0.1966 & 0.1573 \\ 0.2247 & 0.2247 & 0.1966 & 0.1966 & 0.1573 \\ 0.2247 & 0.2247 & 0.1966 & 0.1966 & 0.1573 \\ 0.2247 & 0.2247 & 0.1966 & 0.1966 & 0.1573 \end{bmatrix}$$

Unfortunately, the characteristic matrix  $[I - P]^{-1}$  is now singular.

To conclude: if we wish to find the limiting matrix of some Markov process, the transition matrix must be regular and hence the flowgraph must be of a closed system so as to represent an infinitely repeated process. On the other hand, if the system transfer function is required, then the flowgraph must not be closed otherwise  $[I - P]$  will be singular. It is the open system we will use next to find the variance of a Markov process without recourse to the use of differential calculus by the methods introduced in section 3.7.1.

#### 4.5.4 Variance.

As implied in section 4.1.2, all transition matrices can be expressed in canonical form. By putting a Markov process transition matrix into canonical form Kemeny & Snell provide a means of separating the parts of the process which influence the variance from those which do not.

In canonical form, the partitioned transition matrix

$$P = \begin{bmatrix} I & | & 0 \\ \hline - & - & | & - & - \\ R & | & P \end{bmatrix} \quad (4.10)$$

Where  $I$  represents the absorbing states,  $R$  the transient states leading to absorption,  $P$  the transient states not leading to absorption and  $0$  is null.

$$\text{The characteristic matrix } F = [I - P]^{-1} \quad (4.11)$$

$$\text{and the variance matrix } V = F[2 \text{ DIAG}(F) - I] - (F \square F) \quad (4.12)$$

Note the similarity to the non-matrix form of variance given by {3.13} and {3.14}.

Performing the same analysis on the stochastic transition matrix  $P$ , of example 2 (see section 3.7.2) gives the following results.

$$P = \begin{bmatrix} 0 & 1 & 0 \\ 0.3 & 0 & 0.7 \\ 0 & 0 & 0 \end{bmatrix}$$

Using (4.11)

$$F = \begin{bmatrix} 1.43 & 0.43 & 1 \\ 0.43 & 0.43 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

and from (4.12) the variance

$$V = \begin{bmatrix} 0.612 & 0.612 & 0 \\ 0.612 & 0.612 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Now the sum of all the elements of  $V$  is  $\sum_{i=1}^n \sum_{j=1}^n V(i,j) = 2.448$

which is exactly the variance calculated in example 2 of section 3.7.2 using differential calculus.

#### 4.5.5 The Reverse Markov Chain.

According to the Kolmogorov criterion [Kelley, 1979] a Markov chain is reversible if and only if its transition probabilities satisfy:

$$P(1,2) P(2,3) \dots P(n-1,n) P(n,1) = P(1,n) P(n,n-1) \dots P(3,2) P(2,1)$$

for nodes  $1 \dots n$ ,  $n$  finite.



A reverse form of the index matrix I, which for want of a better name I will call the mirror matrix M, is useful in cases where the above applies.

$$M = \begin{bmatrix} 0 & \dots & \dots & 0 & 1 \\ \cdot & & & \cdot & 0 \\ \cdot & & & \cdot & \cdot \\ \cdot & & & \cdot & \cdot \\ \cdot & & & \cdot & \cdot \\ 0 & \cdot & & \cdot & \cdot \\ 1 & 0 & \dots & \dots & 0 \end{bmatrix} \quad (4.13)$$

M simply performs a rotation of whatever matrix it operates on. To reverse a matrix R,

$$R(\text{reverse}) = M R^T M \quad (4.14)$$

#### 4.6 Summary

In this chapter a selection of matrix techniques useful for obtaining statistical data from the flowgraphs of workcell networks have been discussed. The list is by no means complete and a whole range of matrix operations exist in other texts (with varying relevance to flowgraph theory), too numerous to cover here [Gonnet, 1984].

A new method of achieving the same result, but without the use of differential calculus has been introduced for finding the derivative, and hence the flows and average tolls of a network in matrix form. This is most useful when the necessary calculations are to be performed by digital computer, an almost certain criterion when the networks, and resulting matrices become large.

These techniques not only give us the necessary tools for carrying out simulation of robot workcells but also, as will be shown in the next chapter, the ability to program at object level taking account of sensor data in real-time.

## 5. INTERACTIVE SENSORY SYSTEMS

Robotics is that field concerned with the intelligent connection of perception to action.

Mike Brady.

All robot systems incorporating any form of error recovery must interact with sensors interfacing with the physical environment. However, sense parameters manifest themselves in different forms at different levels of programming, just in the same way a program on a conventional computer has a different representation to the user at the terminal than do the corresponding logic signals experienced by the CPU.

Nilsson separates errors into two distinct types: failures and surprises [Nilsson, 1973], and uses demons to watch for the surprises. This assumes that failures are expected errors and surprises are unexpected errors. A demon is a software surveillance device which 'wakes up' on the occurrence of a particular action or statement, then watches out for further information to verify what action to be taken [Le Beux, 1984]. Similarly, Singh and Hindi call error occurrences "scheduled" if they occur as part of a planned sequence, and "conditional" if they are triggered by events [Singh & Hindi, 1989]. For example the usual PICK, TRANSP and DROP routines used in the examples of chapters 2 and 3 are "scheduled" whereas, the error recovery paths are "conditional". Further categorization of errors is attempted by Srinivas (one of the original researchers in the field of robotic error recovery) into four basic types: operational errors (position, servoing etc); information errors (manifestation of operational errors); precondition errors (eg. non-existence of object); constraint errors (eg. object faulty) [Srinivas, 1976]. Using this classification a failure tree can be used to represent all possible explanations of failure. "Failure reason analysis" can then be used to limit this set to only those which are possible or likely to occur [Srinivas, 1978].

Kamel introduces an interesting philosophy in comparing the games theoretic approach to the concept of error recovery. Here, the planned task performed by one player to achieve a specified goal is hindered by the action of an opponent in the form of unpredictable errors [Kamel, 1988]. This predictive/reactive compromise can be modelled by a system containing a finite set of possible states and is thus ideal for the simulation of error recovery strategies.

## 5.1 Robot Programming Levels

I hate definitions.

Benjamin Disraeli.

Over the past few years a number of attempts have been made to formulate robot programming levels [Gini, 1987], [Rodighiero & Canciani, 1987] with varying degrees of success. In most cases TASK, OBJECT, ROBOT (or manipulator) and JOINT are agreed as the four levels of robot programming. All the above authors agree that task level programming is still a research topic and that no commercially available task level programming system yet exists.

Attempts at specifying such task orientated models have been recently made [Kumpel & Rosa, 1987], whilst Rodighiero & Canciani define such a language called IPL (Implicit Programming Language). This does not however, appear to be a true task level programming language, but lies somewhere between Gini's "object level" and the conventional robot level. Lyons [Lyons, 1987] considers most conventional robot programming languages to be "just cosmetic alterations on a general purpose programming language" and introduces 'RS', a system computationally equivalent to PETRI nets. However, in all cases the definition of the forms in which the sensor parameters are to appear at these different levels have been omitted.

What must be done is to determine at what levels programming can be implemented practically and then choose that level most suitable for the given requirements and physical constraints. The following definitions have been formulated to include both the robot programming levels and the corresponding sensor representations at these levels. The sensor level descriptions are included within parentheses.

### TASK level (CAUSE level)

Complete task routines, ie., 'Assemble Part'

(Reasoned error causes, ie., 'Bin empty')

### OBJECT level (DECISION level)

Program segment routines, ie., Robot program subroutines such as PICK, TRANSP, DROP etc.

(Results of sensor merging and fusion, ie.,  $A \cap B \cup (A \cap C)$ , which appear as Boolean decisions.)

### MANIPULATOR level (INFORMATION level)

Robot programming language commands, ie., VAL II commands such as MOVES POS1, APPRO POS2 etc

(Actual sensor outputs at programming language level, ie., SIG1, SIG2 etc)

### JOINT level (DATA level)

Lower level programming, ie., Geometrical translations as would be carried out with a lower level programming language such as PASCAL or C. Simple high level robot programming language commands such as DRIVE 1,20,50.

(Direct sensor outputs, ie., Binary sensor data, analog output levels etc.)

At object level, sensor data is in the form of decisions which may be the result of sensor fusion, knowledge based reasoning etc. This allows object level programming to be conducted without any consideration of actual sensor implementations. It is not certain how much transparency may be needed. From a systems view point, each level should be completely independent [Martin, 1965]. However, from an engineering angle, the ability to see through to details of lower levels from higher ones, or even to access them is often desirable. For example, the ability to program at assembly level from within a BASIC program [Coll, 1984] can be a very useful attribute.

Nevertheless, assuming the former, a process implemented at the object level must be completely independent of the robot level commands in the same way that a conventional programming language is independent of (as far as the user is concerned) the processor assembly code into which the programming commands are compiled. Similarly, sensor data at the object level must be in the form of actual decisions as the result of an overall sensing requirement, not the outputs of individual sensors. For example, there may be several sensors all of whose outputs must be combined to give a decision as to the success or failure of some operation. The object level is concerned only with this final decision and not the individual measurands themselves.

Malcolm & Fothergill use the analogy of machine code for joint level, assembly language for manipulator level and that of a high level language such as Pascal or C for object level programming [Malcolm & Fothergill, 1987]. Task level differs in that it contains no direct reference to spatial relations and as a consequence may vary in degree from a set of operations such as 'Pick Object', 'Move Object' and 'Place Object' to a complete macro such as 'Build Subassembly'. The one common factor is that it contains no explicit reference to actual geometric or sensor data. This makes task and object level problems ideal for expressing in flowgraph form.

At this stage it should be noted that the term "object level" is sometimes used to refer to a level of computer programming (object oriented programming) rather than one pertaining to the physical robot world of objects. For example, Stefik and Bobrow define an object as an entity within a program which combines both procedures and data [Stefik & Bobrow, 1986]. Whereas in Gini's "real world", an object is a physical device and an object level command consists of a procedure carried out on a physical object using its corresponding geometrical description. This is in many respects simply a different interpretation of the same thing.

## 5.2 Sense Parameters.

The child does not know for a long time how to distinguish a cat from a dog; only when he happens to see them both side by side or when their images become customary can he distinguish between them.

A. Bogdanov (1873-1928).

Apart from the choice of the language for controlling the robot actions there is the manner in which the system is driven by the chosen algorithms. It is usual to have a set of procedures which are called by a main program in a set sequence to achieve a number of tasks in a particular order. Sensing and decision making is done between the calling of procedures in the main program. This is quite satisfactory where the number of sensors and/or levels of error recovery are small, but for more complex systems it becomes clumsy.

In this chapter continuous sensing will be distinguished from discrete by the fact that continuous means that the sensor(s) in question are monitored continuously (though in practice this will usually mean they are interrogated at regular intervals) rather than at the beginning and/or end of a program step to determine the next program step. This must not be confused with the definitions of discrete and continuous as given in some other texts [Johnson, 1986] where discrete sensing is that required to determine when an actuator has reached a certain point and continuous sensing that where the sensor would be used to maintain a specified trajectory or continuous path. The difference in definitions, though subtle, is important and should not be overlooked.

In discrete sensing the required parameter may be sensed continuously, but interrogated only at the required program stage. In a continuously sensed system, the workcell/robot controller is informed the instance the change in the sense parameter occurs, allowing the controller to act immediately.



As pointed out in section 2.2.2, under these conditions, the sense parameter is no longer the sensor measurand itself but the existence of change in the measurand, either with respect to time or some other pre-determined reference. This is known as sensor transition driven programming. [Monkman, 1989].

Given a binary sensor, the sense parameter under continuous sense conditions is the change in logic level. If used correctly, this attribute can have considerable advantages. A rather inefficient approach would be to sequentially interrogate the sensors continuously in an attempt to detect changes in sensor state (as was considered in 4.4.2). A more astute idea is to let the sensors interrupt the controller in the event of a change in any of the sense parameters. The controller can then interrogate all the relevant sensors once and act accordingly.

### 5.3 Interrupts

These are available on all computers but usually accessible to the user only for very drastic actions like 'reset', though lower priority, or maskable, interrupts are sometimes available to the user via some form of input/output mechanism, ie. the 'break' key on a keyboard. In the case of robot control computers, higher level interrupts are rarely (if ever) accessible to the user. One possible exception is the ASEA controller which incorporates up to 6 interrupts [ABB, 1987] though this is a robot system which is PLC driven and has no high level language facility. Fortunately most languages have some form of software capability which has the same effect (though without the instant response of hardware interrupts) available through the I/O system, for example the 'REACTI' command in VAL [VAL II, 1984], or MONITOR in AML [IBM, 1981]. In both cases a window time of approximately 20 mS is required, which is not particularly fast when compared with normal robot joint operating velocities.

In the early days of process control languages CORAL66 was developed for real-time applications, whilst at the same time providing a structured programming environment with all the usual high level features [Woodward, 1970]. However, it was not until the advent of the real-time language PEARL that a process control language capable of handling interrupts directly became available [Werum & Windauer, 1982]. PEARL also has parallel capability. That is to say, tasks may be executed concurrently, either synchronously or asynchronously. Task priorities and interrupts regulate the co-ordination between task blocks (processes). Moreover, interrupts are simply included as variables in the source code and require no other special handling features.

It is interesting that many of the features regarding real-time operation and concurrency which have been part of process control languages are just now starting to be included in robot programming languages. Some standard high level languages also possess these features, resulting in recent suggestions as to their use within robotic programming.

Modula-2, unlike its predecessor 'Concurrent Pascal', provides facilities for handling priority interrupts. At joint level there is the need for matrix multiplication where the independence of the result of each row by column multiplication mean that the operations must be done simultaneously [Dyer, 1985]. This is equally applicable to object level programming. Also, unlike most implementations of Ada, Modula-2 is intended for real time program execution interactively with peripheral devices. This together with its parallel ability makes Modula-2 possibly the only viable non specialist high level language suitable for robot programming [Zwarico, 1985].

A new system called DM2 (distributed modula-2) has recently been developed for use with networked, heterogeneous, multiprocessor applications [Mellor, 1987]. This has the advantage of not requiring shared memory or shared variables between processors, i.e., configurations in which each workcell has its own (possibly different) processor.

In describing 'non-synchronous' sensing Milovanovic [Milovanovic, 1987] introduces the concept of using interrupts directly with the main CPU. Non-synchronous sensing basically means that the process in hand can be interrupted whilst it is in progress without an explicit I/O request being made first (continuous sensing).

All interrupts have a set priority as do many of the computer's executable tasks, for example the memory refresh routines will have a higher priority than most interrupts. Some routines may have a higher priority after execution has commenced (but before it is finished) than when they are awaiting execution. Returning to the concepts of queueing theory we have what are known as 'locking systems' [Baccelli, 1987]. In a queue system most simple customers are served by each server in turn (a serial process). However, there is another type of customer: that which requires the service of all servers simultaneously (a parallel process). Such a system is known as a locking system. This is similar to a maskable interrupt in that it takes priority over any new customers but has a lower priority than those customers already being served and therefore must wait until all the servers are free.

Since the advent of so called "intelligent" sensors such as line scan cameras, multiple axis force sensors and other sensors which include a high degree of signal processing (and possibly such features as self-calibration) within the sensor device, the use of serial lines has become common. This levies a severe time penalty between an observed event taking place and the controller receiving the observation data. Fortunately, such sensing systems are usually required to be interrogated after the receipt of a signal from a more basic (and somewhat faster operating) binary sensor. However, not all such devices are constrained to using serial lines for communication. Direct connection of sensing devices to processor memory (DMA devices) provides a more elegant and faster means of data acquisition [Milovanovic, 1987].

## 5.4 Sensor Driven Programming.

He who runs the information runs the show.

Joseph Goebbels.

Simple sensor based control has been around for many years now, especially where only local sensing, feedback and correction are required. For example, during the insertion of electrical components into holes during printed circuit board assembly [Karkkainen et al, 1988].

Considerable problems exist in implementing sensor driven programming using more complex non-binary sensors such as vision systems [Williams et al, 1986]. This problem is eliminated at object level if the definitions of 5.1 are observed. This is because, at object level, decisions made at nodes of the operational flowgraph are based on Boolean data only. Whether this comes from sensor outputs which have been fused or merged to produce simple binary decisions, or from an expert system or AI and knowledge base is irrelevant when programming at object level.

On a more global scale, sensor driven programming implies the total control, or even generation, of programming strategies by sensor output data. This is referred to by Chapman & Agre as "reactive planning" [Chapman & Agre, 1986]. Taken to its extreme, reactive planning would appear to obviate the need for a goal plan. Unfortunately, the inherent inability of reactive planning systems to "think-ahead" renders this approach unreliable. [Firby, 1987]. Where an interrupt (or similar reaction) facility is available, sensor transition driven programming can be implemented [Monkman, 1989].

## 5.5 Object Driven Programming.

This is a rather obvious but little used concept in robot programming which concerns sets of autonomous workcells rather than the operation of individual cells themselves. It should not be confused with the term "Object level programming" which pertains to a level of software implementation as described in 5.1. We shall start with a rather simple hypothesis:

"All geometric information concerning an object is contained wholly within the physical frame containing that object"

This may seem fairly obvious but its usefulness is often overlooked. For example, where one autonomous workcell passes an object directly to another autonomous workcell, provided the timing constraints are correct, the only required communication between cells is the object itself. Only under error conditions may other communications be required, like a 'wait' instruction from the second workcell to the first if either an error occurs at the second or the timing is incorrect allowing the first workcell to run faster than the second. Otherwise no other information is needed. The second cell does not usually need to know what the first is doing. The only information the second (or subsequent) cell requires is to know that an object has arrived and its relevant geometric data, and then only when its ready to act. If another object arrives before the present operation is complete then it must wait until it is required. Once again we're back to the philosophies of queueing theory.

With regard to geometric information on the object, ie., orientation, position etc., this is usually only required at the position of entry into the cell. There are basically two methods by which this information may be acquired.

- 1) By multiple sensor analysis. For example a vision system and computer analysis to determine the required geometrical data. This has the disadvantage of time and cost, not to mention the problems associated with lighting and other environmental effects.
- 2) By mechanical compliance. If the position and orientation are known to be incorrect by an unknown amount but within a definite range (ie., by being purposely deposited some distance from the desired point) then the object may be moved by the next operation so that it will always end up in the correct position and orientation.

This second technique is especially applicable to solid objects. One good example of this is in the use of vibratory feeders, where objects are taken from completely random positions and orientations and then forced to comply with a set of constraints causing them to be repositioned as desired [Boothroyd et al, 1978]. Another is the manipulation of objects without prehension [Mason & Salisbury, 1985], ie., the movement of rigid objects by means of compliant or non-rigid grippers. Conversely, the principle has also been applied to non-rigid objects [SERC, 1988] where fabric panels were made to fall into a rigid robot gripper in the form of a shovel or scoop. As the fabric slips onto the gripper one axis is aligned against the gripper rear, the second axis being aligned by running the object against a compliance so as to slide it into the correct position verified by a single sensor.

## 5.6 Summary.

Choice of programming structure is highly dependant on a number of factors. The number of sensors available limits the degree to which an error recovery implementation can be acheived. The method by which the sensors can be interrogated influences the manner in which the execution of an algorithm which includes error recovery can be implemented. For example, it is not possible to implement a sensor transition driven philosophy with only simple discrete sensing.

Continuous sensing, particularly where interrupts are available, provides a faster response in detecting any error occurrence. However, how the error is dealt with thereafter, and within what timescale, depends largely on both the available programming structure and the physical configuration of the cell.

Before any serious programming can be considered a hierarchical system of levels must be employed. This not only has all the usual advantages of portability and readability but also provides a framework from which any number of individual work cells may be controlled. This chapter has seen the formulation of a new description of robot programming levels, which also defines the type of sensor data to be used at each level. Careful use of the object driven techniques outlined will reduce the necessary communication overhead between levels.



## 6. PROGRAMMING AND SIMULATION.

Coming events cast their shadow before.

Goethe.

This section is not intended to give an in-depth analysis of present day robot programming languages. Such studies already exist [Bonner & Shin, 1982], [Gruver, 1983], [Gini & Gini, 1984] and more recently [Blume & Jacob, 1986], also with interest pertaining to parallel processing languages [Zwarico, 1985]. Moreover, the intention is to provide a view of the present "state of the art" of robotic systems programming. Many of the shortcomings associated with current programming techniques are illustrated and the ground prepared for fresh approaches based on network planning, sensing and error recovery.

Critical path analysis will not be considered in any depth as its relevance to our task, though strong, is strictly limited. For acyclic networks critical path analysis is examined quite thoroughly by Martin [Martin, 1965] who gives a number of useful references in this respect. More recently, optimisation of PETRI nets with the object of finding the time optimal path, is considered by Freedman & Malowany using PROLOG [Freedman & Malowany, 1988]. With regard to digraphs, Grimaldi dedicates a complete chapter to the subject of shortest path algorithms using maximum flow/minimum cut techniques. These are included along with a number of other optimization topics [Grimaldi, 1989].

As a result of the work done on queueing theory, a considerable number of computer simulation packages already exist such as HOCUS [HOCUS], SIMSCRIPT [Markowitz, 1979] and SIMIAN (based on IBM's GPSS) [Open University, 1982] to name but a few. Taha [Taha, 1987] gives an extensive resumé of simulation languages and their uses, including the above and many more. In most cases a set of resources (entities) with characteristics (attributes), such as time, are introduced into the model and various statistical calculations made, such as correlation of data. Most packages have a number of statistical probability distributions such as Poisson, Normal, Binomial etc., according to which entities can enter and leave the model. A selection of relevant probability distributions are given in appendix D. GERT has a suite of programs written in FORTRAN for simulation purposes [Pritsker, 1968].

Means by which Petri nets can be converted into executable code for simulation have been around for some time. One method uses a simple character string language APN to translate into a procedural language called XL/1 which was developed specifically for the purpose of directing the activities of a collection of automata engaged in the firing of annotated Petri nets [Nelson et al, 1983]. This is then translated into PL/I and PL/S for compilation and execution. Though XL/1 holds the parallel processing group delimiters, these are not compatible with PL/I and PL/S (FORTRAN type language constructions) so only serial execution is possible. This long-winded set of computing processes has largely been replaced by more succinct and user-friendly languages such as GSPN [Chiola, 1987], but it demonstrates the complexity of converting mathematically intractable graphical descriptions into executable computer programs.

More sophisticated simulation packages are presently becoming available, the most recent being STEM [Popplewell & Jiao, 1989]. This contains a number of options allowing the user to specify Petri nets, flowgraphs etc., with the ability to break down the events into trees, visualise the flow of tokens etc. Like most such systems, STEM is a straight simulation algorithm rather than a mathematical analysis package.

Looking to the future, ideas for the full integration of simulation packages with large databases are beginning to appear. SIMPRO is a computer simulation and planning system which is intended to be integrated with a factory database when finally implemented (sometime in 1990 according to INPRO). The graphic language (GBS) consists of a net editor and a net simulator to handle advanced Petri net models. GBS consists of several modules of modula-2 which may be called for each Petri net firing process. This provides the emulatory simulation for which several stochastic distributions are available. [Weissenborn, 1989].

A contrast is drawn between two approaches to simulation of robot factory automation systems by Esposito & Vento [Esposito & Vento, 1987]. On the one hand specialist simulation languages exist, as already mentioned, which require skilled operators. On the other hand more sophisticated simulation packages are emerging which tend to de-skill the operation to some extent, but at the expense of severely limiting the field of operation.

## 6.1 Conventional Structures

Language is the most human thing about us: In a sense, the invention of language made us human: but language, perhaps for the same reason, is the greatest expression of human fidelity, or if you like, original sin.

C.P. Snow, 1970.

With the advent of computer languages specifically designed for use with robots, such as VAL, much of the work required for general robot control, ie. the calculation of trajectories, driving of joints, storing and retrieving of locations etc, has been taken care of within the primitives available to the user.

However, this does not mean that conventional languages, such as PASCAL, FORTH etc, are unsuitable. Their use though, is usually restricted to the less expensive breed of robots where higher level language versions either do not exist or are too expensive to implement. The main application of conventional languages, with regard to robotics, is in simulation where something more general purpose than a dedicated simulation language is desirable. Boucher [Boucher, 1986] uses BASIC for simulation in robotic assembly feasibility tests. Attempts have recently been made in adapting the general programming and simulation language SMALLTALK for robotics [LaLonde et al, 1987]. This combines both the facilities of simulation and robot control within one programming domain. As outlined in chapter 1, this is essential for any complete planning and execution system.

Choice of programming structure is highly dependant on a number of factors. The number of sensors available limits the degree to which error recovery implementation can be acheived. The method by which the sensors can be interrogated influences the manner in which the execution of an algorithm which includes error recovery can be implemented. For example, it is not possible to implement a sensor transition driven philosophy with only discrete sensing.

The flowgraph notation used may remain essentially the same whether discrete or continuous sensing is to be employed. However, the choice of use of notation (active path or active node) is dependant on a number of factors (as pointed out in chapter 2) including the sensing configuration used. Flowgraphs easily map into matrices, and for those familiar with the computer programming language APL [Katzan, 1970], programming with matrices will seem an obvious step forward.

The advent of parallel processing has provided methodologies capable of expediting the manipulation of matrices. Furthermore, this allows the simultaneous calculation of all matrix elements yielding a whole matrix as a single result instead of as a string of serial values emerging at different times. This has a considerable bearing on systems which are to operate quickly and in real time. This will be discussed in greater depth in section 6.3 of this chapter.

## 6.2 Simulation and Modelling.

If a man take no thought about what is distant,  
he will find sorrow near at hand.

Confucius.

So far we have discussed the use of Markov chains in producing statistical data on the operation of a given set of robot operations using known times and probabilities. To enable a proper simulation strategy to be implemented we must first define the variables to be used:

- 1.) An  $m$  by  $m$  toll matrix  $T$ , where  $T_{ij}$  is an operation routine toll.
- 2.) An  $m$  by  $m$  stochastic matrix  $P$ , where  $P_{ij}$  is a sensor based decision probability.
- 3.) A flow matrix  $F$ , such that  $F = [I - P]^{-1}$
- 4.) A time average matrix  $A$ , such that  $A = T \square F$
- 5.) A variance matrix  $V$ , such that  $V = F (2 \text{DIAG}(F) - I) - F \square F$
- 6.) An  $m$  by  $m$  routine matrix  $R$ , where  $R_{ij}$  is a program operation routine.
- 7.) An  $m$  by  $m$  sensor matrix  $S$ , where  $S_{ij}$  is a sensor based boolean decision.
- 8.) A process vector  $U_0$ , such that  $U_{n+1} = \text{INT}(\sum_{j=1}^m (U_n (R \square S)))$

9.) A Mirror matrix  $M$ , such that a Markov chain matrix  $R$  of executable functions, may be reversed by the operation  $M R^T M$

In all cases  $i$  denotes the matrix row, whilst  $j$  denotes the column.

For the purposes of the software the operation of  $\square$  between a string and a boolean variable yields a boolean. ie., the syntax is:  $\langle \text{string} \rangle \square \langle \text{boolean} \rangle \rightarrow \langle \text{boolean} \rangle$

It will next be shown how the Markov chain can be used, not only for simulation purposes, but also as an object level programming technique.

### 6.2.1 Object and Task Level Programming.

As mentioned in chapter 1, the combining of simulation and robot control functions within the same programming environment has some distinct advantages over the use of separate software packages. This section deals with a new object and task level programming philosophy which is inherently usable for both off-line simulation and on-line real time robot control.

Taking the same variables defined in section 6.2 for the purposes of off-line simulation, we now have the routine matrix  $R$  as a matrix containing actual robot control program routines instead of operation tolls, and the sensor matrix  $S$ , as one containing the actual Boolean decision values (either directly or as a result of some sensor fusion done at manipulator level) rather than their respective outcome probabilities.

Using exactly the same matrix manipulation techniques as for simulation, we can now actually control our robot simply by multiplying the appropriate vectors and matrices. Returning to a simple example of the kind first introduced in chapter 2 and its corresponding flowgraph as shown in figure 6.1. The boolean decisions are denoted by POG and its complement, where POG is a simple anagram for "Part On Gripper".

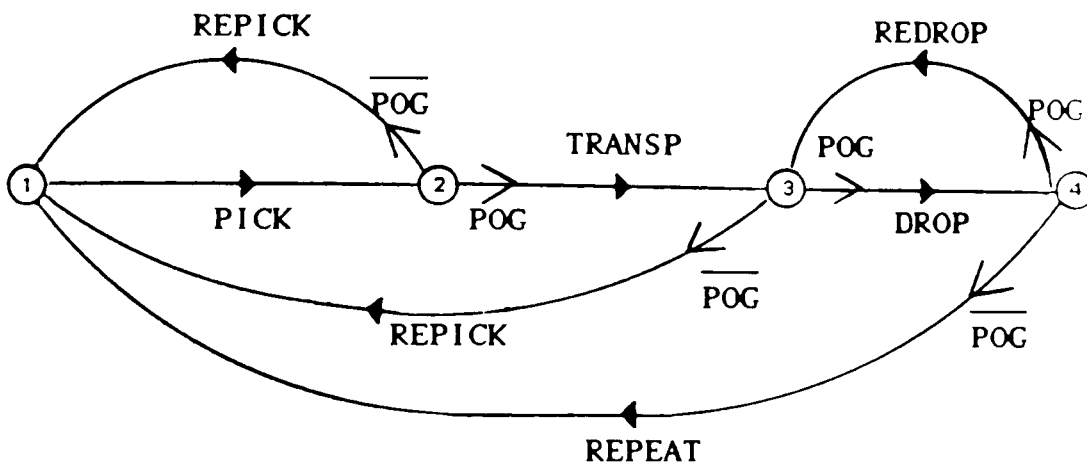


Figure 6.1 - Simple Pick and Drop Example.

From this flowgraph the usual matrices can be derived for times and probabilities, or in this case for routines and sensor values.

$$R = \begin{bmatrix} 0 & \text{PICK} & 0 & 0 \\ \text{REPICK} & 0 & \text{TRANSP} & 0 \\ \text{REPICK} & 0 & 0 & \text{DROP} \\ \text{REPEAT} & 0 & \text{REDROP} & 0 \end{bmatrix}$$

$$S = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \overline{\text{POG}} & 0 & \text{POG} & 0 \\ \overline{\text{POG}} & 0 & 0 & \text{POG} \\ \overline{\text{POG}} & 0 & \text{POG} & 0 \end{bmatrix}$$

Given a starting vector  $U_0 = [ 1 \ 0 \ 0 \ 0 ]$

the next vectors:  $U_1 = U_0 (R \square S)$

$U_2 = U_1 (R \square S) \dots$

and so on.

ie.  $U_{n+1} = U_n (R \square S)$



During program operation, the Routine matrix  $R$  will remain the same and only the Boolean  $S_{ij}$  values in the sensor matrix will change in accordance with the corresponding sensor outputs. It is these sensor values which will determine the run time operation of the program.

The use of this method in both simulation and robot programming is not restricted to single robot operation but can be extended to multiple parallel processes as the next examples show.

Using the flowgraph of figure 6.1 again, but this time assume we have two robots operating within the same cell, one just about to start a PICK operation and the other commencing a DROP operation. hence the starting vector is:

$$U_0 = [ 1 \ 0 \ 1 \ 0 ]$$

Slight modification is required to  $S$  to make the sensor results unique. These are likely to be a result of sensor fusion and are therefore given relative to a position in the flowgraph rather than being robot specific.

$$R = \begin{bmatrix} 0 & \text{PICK} & 0 & 0 \\ \text{REPICK} & 0 & \text{TRANSP} & 0 \\ \text{REPICK} & 0 & 0 & \text{DROP} \\ \text{REPEAT} & 0 & \text{REDROP} & 0 \end{bmatrix}$$

$$S = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \overline{\text{POG}}_1 & 0 & \text{POG}_1 & 0 \\ \overline{\text{POG}}_2 & 0 & 0 & \text{POG}_2 \\ \overline{\text{POG}}_3 & 0 & \text{POG}_3 & 0 \end{bmatrix}$$

$$U_1 = U_0(R \square S)$$

Suppose the PICK operation performed by the first robot is successful, but the DROP operation carried out by the second robot fails. All the POG elements of S will have Boolean value 1, hence,

$$\begin{aligned}
 U_1 &= [ 1 \ 0 \ 1 \ 0 ] \begin{bmatrix} 0 & \text{PICK} & 0 & 0 \\ 0 & 0 & \text{TRANSP} & 0 \\ 0 & 0 & 0 & \text{DROP} \\ 0 & 0 & \text{REDROP} & 0 \end{bmatrix} \\
 &- \begin{bmatrix} 0 & \text{PICK} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \text{DROP} \\ 0 & 0 & 0 & 0 \end{bmatrix}
 \end{aligned}$$

$$\text{So } U_1 = [ 0 \ 1 \ 0 \ 1 ]$$

Now if a second failure occurs for robot 2 while the first robot is successful again, then robot 1 is likely to catch up with the second robot. This can be seen if we take all the POG values as 1 once again to find  $U_2$ .

$$\begin{aligned}
 U_2 &= [ 0 \ 1 \ 0 \ 1 ] \begin{bmatrix} 0 & \text{PICK} & 0 & 0 \\ 0 & 0 & \text{TRANSP} & 0 \\ 0 & 0 & 0 & \text{DROP} \\ 0 & 0 & \text{REDROP} & 0 \end{bmatrix} \\
 &- \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & \text{TRANSP} & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \text{REDROP} & 0 \end{bmatrix}
 \end{aligned}$$

Now we have two operations in one column of U. This denotes simultaneous realization of the same node in the system flowgraph giving a resultant  $U_2$  of:

$$U_2 = [ 0 \ 0 \ 1 \ 0 ]$$

Where more than one robot is operating this can mean collision! Of course, this assumes that each of the operation times of figure 6.1 are all equal and are executed simultaneously. Where this is not the case, the process is no longer strictly Markovian but is known as a semi-Markov process. That is to say, the times for each action are no longer all equal and unity but are dependant on the action being executed, even though the probabilities of execution may remain constant and independant (forming an embedded Markov chain). Now, each robots own set of matrix operations must be performed independantly and continuously compared to detect simultaneous realization as the following example illustrates.

Adding the relative times for each operation, in brackets after the routine name, to give the flowgraph in figure 6.2.

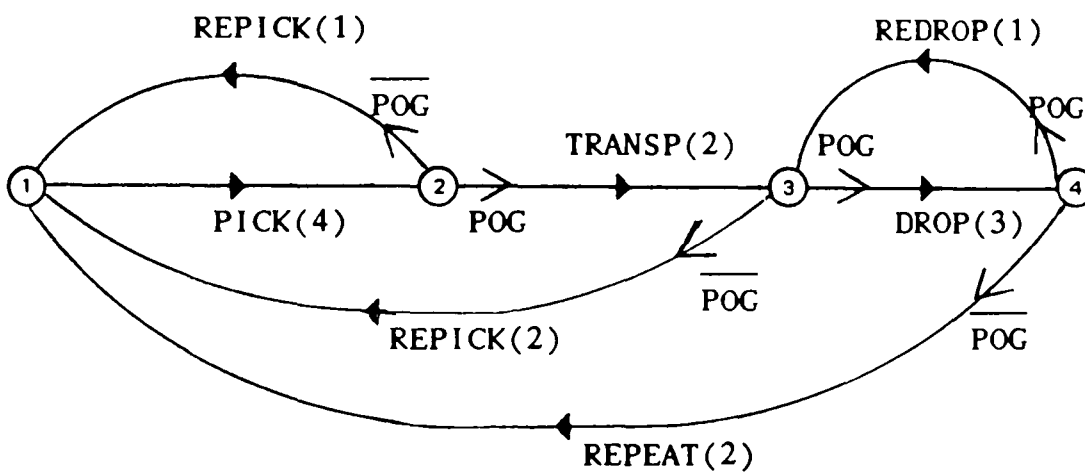


Figure 6.2 - Simple Pick and Drop Example with operation times.

For the first robot we start with the PICK operation again;

$$U_1(1) = [ 1 \ 0 \ 0 \ 0 ] \begin{bmatrix} 0 & \text{PICK}(4) & 0 & 0 \\ 0 & 0 & \text{TRANSP}(2) & 0 \\ 0 & 0 & 0 & \text{DROP}(3) \\ 0 & 0 & \text{REDROP}(1) & 0 \end{bmatrix}$$

$$- \begin{bmatrix} 0 & \text{PICK}(4) & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

and for the second we commence with a DROP once more;

$$U_1(2) = [0 \ 0 \ 1 \ 0] \begin{bmatrix} 0 & \text{PICK}(4) & 0 & 0 \\ 0 & 0 & \text{TRANSP}(2) & 0 \\ 0 & 0 & 0 & \text{DROP}(3) \\ 0 & 0 & \text{REDROP}(1) & 0 \end{bmatrix}$$

$$- \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \text{DROP}(3) \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

So  $U_1(1) = [0 \ 1 \ 0 \ 0]$

and  $U_1(2) = [0 \ 0 \ 0 \ 1]$

Now supposing as before, the first robot experiences no errors but the second has to repeat the DROP operation. Table 6.1 shows the sequence of events for each step in time.

Table 6.1 - Time & Vectors for two Robots.

START TIME	$U_n(1)$	NEXT ROUTINE	END TIME	START TIME	$U_n(2)$	NEXT ROUTINE	END TIME
0	1 0 0 0	PICK	4	0	0 0 1 0	DROP	3
				3	0 0 0 1	REDROP	4
4	0 1 0 0	TRANSP	6				
6	0 0 1 0	DROP	9	6	0 0 1 0	DROP	7

What table 6.1 reveals is that the DROP procedure is being executed by both robots simultaneously where the start and finish times overlap, ie., when  $U_n(1) = U_n(2)$ .

which is when time,  $t \in [4,7] \cap t \in [6,9]$

ie., between 6 and 7 time units from the start.

This result can be simulated as has been done above. However, recourse to section 4.5.2 and the definition of an Ergodic Markov chain should convince the reader that if physical contact is at all possible, then simultaneous realization is an eventual inevitability in a stochastic system containing two or more independantly operating robots. The simulation merely provides a means of estimating how, when and where.

### 6.2.2 Readability

And we should keep in mind that a program is worthless, unless it exists in some form in which a human can understand it and gain confidence in its design.

Niklaus Wirth, Programming in Modula-2.

The use of a matrix to represent an algorithm will now be compared to the more familiar format of a string of computer program statements. Figure 6.3 shows the flowgraph for a simple pick and place routine with an inspection operation.

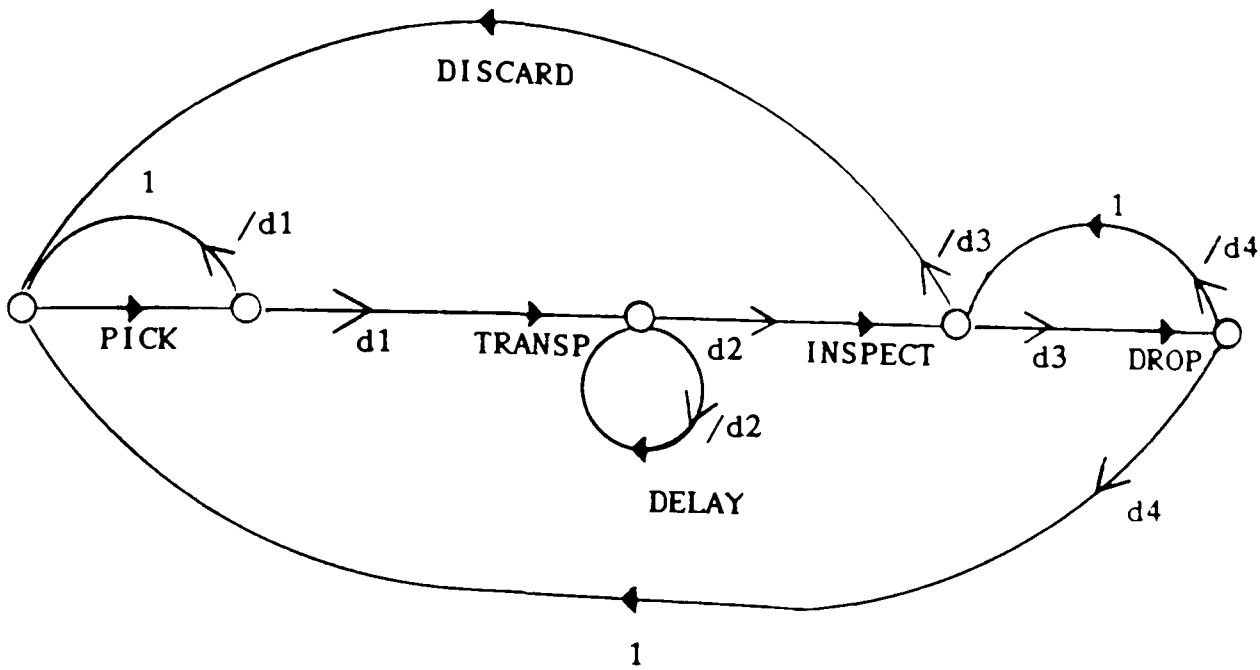


Figure 6.3 - Pick & Place with Inspect Process

The resulting executable matrix, where decision variables are in square parentheses, is:

$$(R \square S) = \begin{bmatrix} 0 & \text{PICK} & 0 & 0 & 0 \\ [\bar{d1}] & 0 & \text{TRANSP} [d1] & 0 & 0 \\ 0 & 0 & \text{DELAY} [\bar{d2}] & \text{INSPECT} [d2] & 0 \\ \text{DISCARD} [\bar{d3}] & 0 & 0 & 0 & \text{DROP} [d3] \\ [d4] & 0 & 0 & [\bar{d4}] & 0 \end{bmatrix}$$

By comparing the relative position of the elements in the above matrix to those of the flowgraph of figure 6.3 it should be obvious what is taking place as the vector U is multiplied and modified each time. In fact, by mentally following the elements of the main off diagonal and then noting the proximity of the feedback paths, the reader should be able to visualise the flowgraph directly from the topology of the matrix. This is not so easy to do with the computer listing for which the equivalent algorithm in pseudo-code is shown in figure 6.4.

```

1   IF d4 THEN
      CALL PICK
   ELSE
      STOP
   IF NOT d1 THEN GOTO 1
   CALL TRANSP
2   IF NOT d2 THEN CALL DELAY
   IF NOT d2 THEN GOTO 2
   CALL INSPECT
   IF NOT d3 THEN
      CALL DISCARD
      GOTO 1
   ENDIF
3   CALL DROP
   IF NOT d4 THEN GOTO 3
   GOTO 1

```

Figure 6.4 - Pseudo code representation

This is far more difficult to read than the matrix representation. Furthermore, as the algorithms become larger the conventional notation becomes increasingly more difficult to read.

### 6.2.3 Extensions to General Programming

So far the techniques used have concentrated on robot programming. However, almost all algorithms are Markov chain representations as the following example will show.

Most programmers are familiar with the Bubble sort of the kind given in the flow chart of figure 6.5 [Forsythe et al, 1975 (p259)]. The corresponding flowgraph version is shown in figure 6.6.

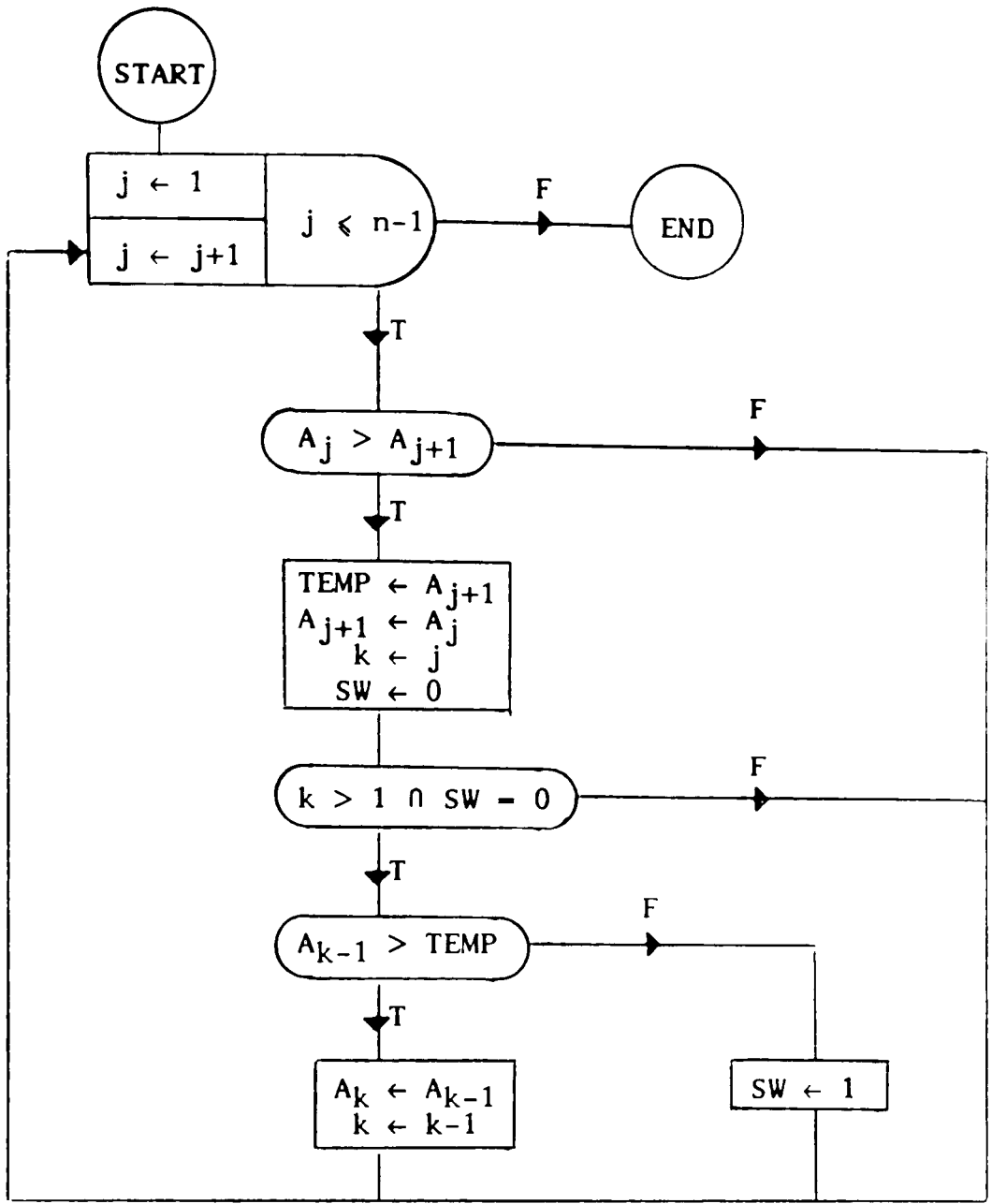


Figure 6.5 - Bubble Sort Flowchart

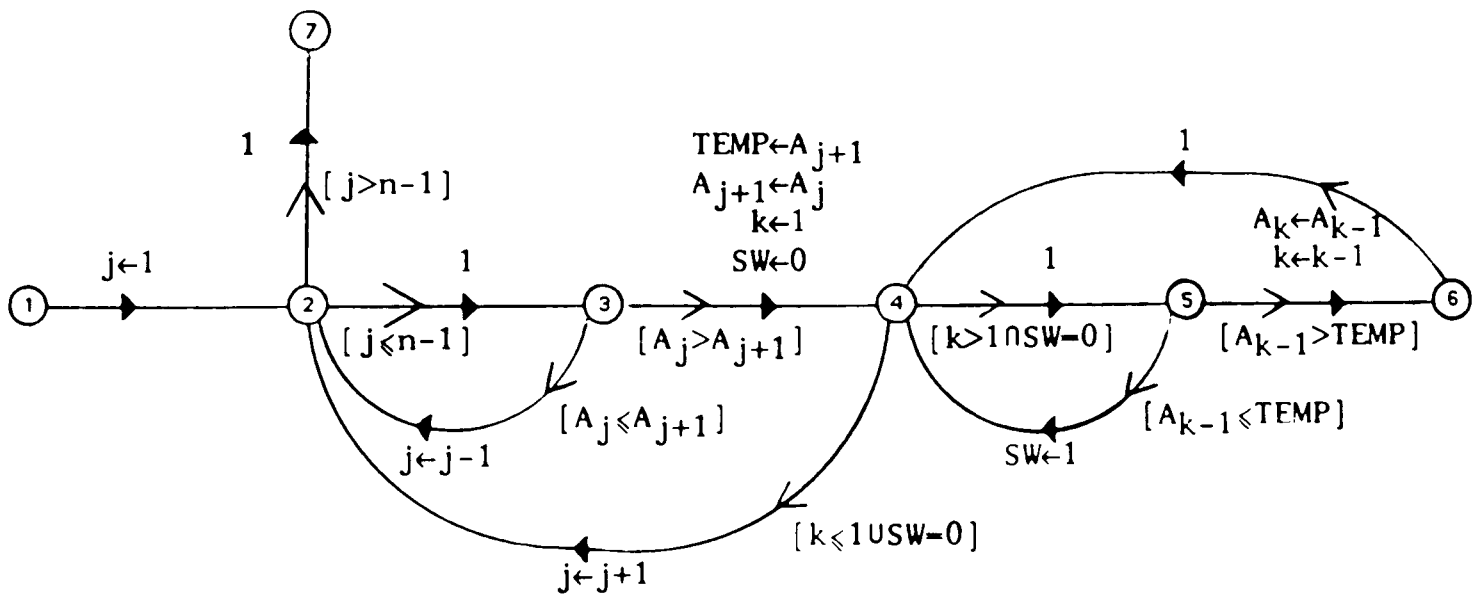


Figure 6.6 - Flowgraph of Bubble Sort



Putting this into matrix form we get the executable matrix of figure 6.7. The equivalent program in pseudo code is shown in figure 6.8.

0	$j \leftarrow 1$	0	0	0	0	0
0	0	$[j \leq n-1]$ 1	0	0	0	$[j > n-1]$ 1
0	$[A_j \leq A_{j+1}]$ $j \leftarrow j+1$	0	$[A_j > A_{j+1}]$ TEMP $\leftarrow$ $A_{j+1}$ $A_{j+1} \leftarrow A_j$ $k \leftarrow 1$ SW $\leftarrow$ 0	0	0	0
0	$[k < 1 \vee SW \neq 0]$ $j \leftarrow j+1$	0	0	$[k > 1 \wedge SW = 0]$ 1	0	0
0	0	0	$[A_{k-1} \leq TEMP]$ SW $\leftarrow$ 1	0	$[A_{k-1} > TEMP]$ $A_k \leftarrow A_{k+1}$	0
0	0	0	1	0	0	0
0	0	0	0	0	0	0

Figure 6.7 - Matrix of Bubble Sort Algorithm

```

j:=-1
1  IF j ≤ n-1 THEN
    IF Aj > Aj+1 THEN
        TEMP ← Aj+1
        Aj+1 ← Aj
        k ← j
        SW ← 0
        IF k > 1 AND SW = 0 THEN
            IF Ak-1 > TEMP THEN
                Ak ← Ak-1
                k ← k-1
            ELSE
                SW ← 1
            ENDIF
        ELSE
            j ← j+1
            GOTO 1
        ENDIF
    ELSE
        j ← j+1
        GOTO 1
    ENDIF
ELSE
    END

```

Figure 6.8 - Pseudo code Bubble Sort

There should be little doubt about which method is the most readable by now, even when several loops are included as with the bubble sort. In fact, the loops show up in the matrix as the elements below the diagonal. The only element above the main off-diagonal is the final state on completion of the algorithm. This is not so clear for the pseudo code representation, and even though the loops are nested and offset from the left of the page it is not so easy to see which conditions determine the order of execution. This is because flowcharts do not map readily into computer code, whereas flowgraphs and their representative transition matrices are effectively homomorphic.

#### 6.2.4 Buffering and Partitioning

As was shown in section 2.2.1, the use of buffering between sections of a flowgraph provides natural delimiters allowing partitioning at these points. So far it has usually been unnecessary to use buffer paths owing to the small size of the flowgraphs. However, as models become larger connectivity becomes more complex. This can result in hereto unforeseen problems.

Returning to the flowgraph of figure 4.1 in section 4.1.2, repeated here in figure 6.9, and its resulting stochastic transition matrix {6.1}.

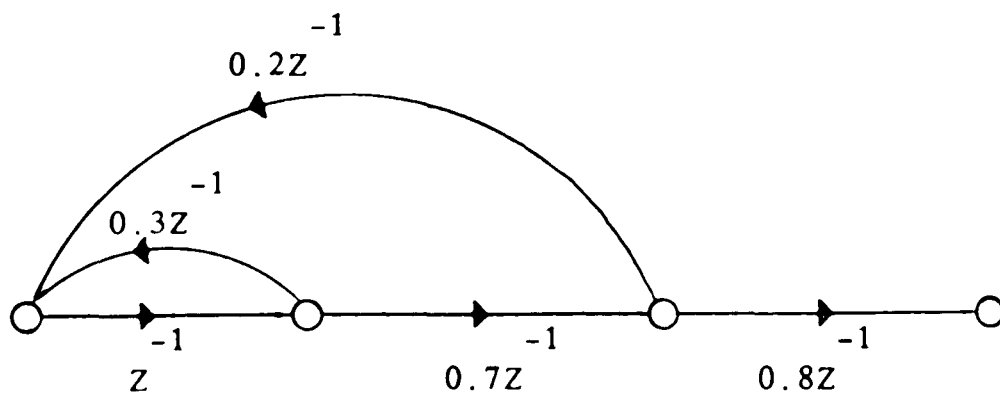


Figure 6.9 - Double Loop Flowgraph

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0.3 & 0 & 0.7 & 0 \\ 0.2 & 0 & 0 & 0.8 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad (6.1)$$

Now finding the flows from  $P^\infty$  is quite simple. However, if the method of the definition of 3.) in section 6.2 is to be used to calculate the inter-node flows and 4.) to find the toll average from the toll matrix T, then the result will be incorrect for this particular model (and any other with a similar topology), as will be demonstrated:

$$T = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad (6.2)$$

$$F = \begin{bmatrix} 1.79E+0 & 1.79E+0 & 1.25E+0 & 1.00E+0 \\ 7.86E-1 & 1.79E+0 & 1.25E+0 & 1.00E+0 \\ 3.57E-1 & 3.57E-1 & 1.25E+0 & 1.00E+0 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 & 1.00E+0 \end{bmatrix} \quad (6.3)$$

$$V = \begin{bmatrix} 1.40E+0 & 1.40E+0 & 3.13E-1 & 0.00E+0 \\ 1.40E+0 & 1.40E+0 & 3.13E-1 & 0.00E+0 \\ 7.91E-1 & 7.91E-1 & 3.13E-1 & -2.22E-16 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 \end{bmatrix} \quad (6.4)$$

$$A = \begin{bmatrix} 0.00E+0 & 1.79E+0 & 0.00E+0 & 0.00E+0 \\ 7.86E-1 & 0.00E+0 & 1.25E+0 & 0.00E+0 \\ 3.57E-1 & 0.00E+0 & 0.00E+0 & 1.00E+0 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 \end{bmatrix} \quad (6.5)$$

mean toll value is: 5.1785714E+0

But using Masons theorem, or simple decomposition, on figure 6.9 gives a toll average of:

$$t_{av} = \frac{2.857 + 0.2}{0.8} + 1 = 4.821 \quad (6.6)$$

The result of {6.6} is clearly different to that given by the sum of the elements of {6.5}.

Closer inspection of {6.2} reveals the fact that T is not upper triangular. This is not a problem with stochastic transition matrices like P because as we know, probabilities are multiplicative, whereas tolls are additive. Obviously we cannot treat them in the same way. The answer is to use the buffering techniques introduced in section 2.2.1. The buffered homomorphism of figure 6.9 is shown in figure 6.10 along with its corresponding toll matrix {6.7}

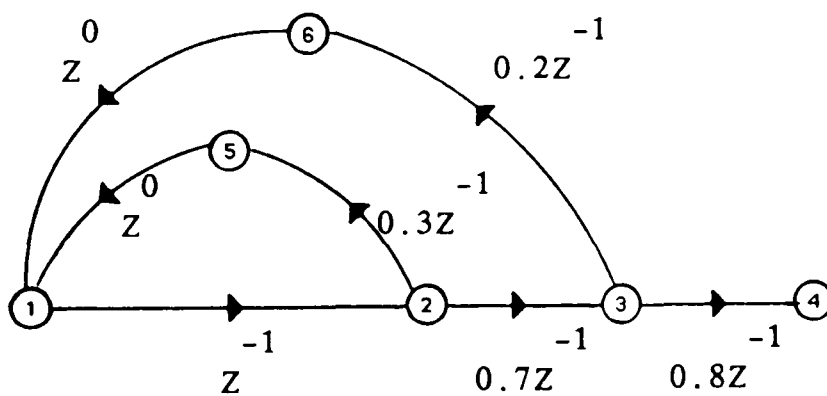


Figure 6.10 - Buffered Homomorphism of Figure 6.9

$$T = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (6.7)$$

Now the matrices are much larger but {6.7} is in upper triangular form. Note the sparseness of {6.7} compared with the stochastic matrix of {6.8}. This is because the buffer paths have a zero toll, despite a transition probability of unity.

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.7 & 0 & 0.3 & 0 \\ 0 & 0 & 0 & 0.8 & 0 & 0.2 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (6.8)$$

$$F = \begin{bmatrix} 1.79E+0 & 1.79E+0 & 1.25E+0 & 1.00E+0 & 5.36E-1 & 2.50E-1 \\ 7.86E-1 & 1.79E+0 & 1.25E+0 & 1.00E+0 & 5.36E-1 & 2.50E-1 \\ 3.57E-1 & 3.57E-1 & 1.25E+0 & 1.00E+0 & 1.07E-1 & 2.50E-1 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 & 1.00E+0 & 0.00E+0 & 0.00E+0 \\ 1.79E+0 & 1.79E+0 & 1.25E+0 & 1.00E+0 & 1.54E+0 & 2.50E-1 \\ 1.79E+0 & 1.79E+0 & 1.25E+0 & 1.00E+0 & 5.36E-1 & 1.25E+0 \end{bmatrix} \quad (6.9)$$

$$V = \begin{bmatrix} 1.40E+0 & 1.40E+0 & 3.13E-1 & 1.11E-16 & 8.23E-1 & 3.13E-1 \\ 1.40E+0 & 1.40E+0 & 3.13E-1 & 1.11E-16 & 8.23E-1 & 3.13E-1 \\ 7.91E-1 & 7.91E-1 & 3.13E-1 & 0.00E+0 & 2.10E-1 & 3.13E-1 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 \\ 1.40E+0 & 1.40E+0 & 3.13E-1 & 1.11E-16 & 8.23E-1 & 3.13E-1 \\ 1.40E+0 & 1.40E+0 & 3.13E-1 & 0.00E+0 & 8.23E-1 & 3.13E-1 \end{bmatrix} \quad (6.10)$$

$$A = \begin{bmatrix} 0.00E+0 & 1.79E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 \\ 0.00E+0 & 0.00E+0 & 1.25E+0 & 0.00E+0 & 5.36E-1 & 0.00E+0 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 & 1.00E+0 & 0.00E+0 & 2.50E-1 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 \end{bmatrix} \quad (6.11)$$

mean toll value is: 4.8214286E+0

This is exactly the toll average given by {6.6}.

This leads to the simple rule that: no two or more paths carrying non-zero tolls may be allowed to enter the same node. The validity of a flowgraph in this respect may be verified by ensuring that the corresponding toll matrix is in upper triangular form.

Note that the flow matrix of {6.3} is the same as the first 4 by 4 minor of {6.9}, and similarly the variance matrices of {6.4} and {6.10} to within the scope of mathematical rounding errors. The only difference lies in the additional elements due to the buffer paths.

Partitioning of a matrix was used in chapter 4 for separating the various states of a Markov process. Its usefulness when used to segregate the different sections of an overall process will now be considered.

As networks become large, their resulting transition matrices not only also become large but they also have a tendency to become sparse, particularly where a great deal of buffering is necessary. This can often result in large amounts of wasted computer memory for matrix parameter storage, not to mention the additional computational overhead required to conduct mathematical manipulations on matrices containing an inordinate amount of zeros! Both for purposes of economy and readability it is often necessary to partition matrices. Flowgraph representations offer some advantages in making this possible.

Using the flowgraph of figure 2.4 from the example in chapter 2, and its buffered counterpart in figure 6.10, their corresponding routine matrices are {6.12} and {6.13} respectively.

$$\begin{bmatrix}
 0 & \text{PICK} & 0 & 0 & 0 \\
 \text{FAIL} & 0 & \text{TRANSP} & 0 & 0 \\
 \text{FAIL} & 0 & 0 & \text{DROP} & 0 \\
 0 & 0 & \text{FAIL} & 0 & 1 \\
 0 & 0 & 0 & 0 & 0
 \end{bmatrix} \quad (6.12)$$

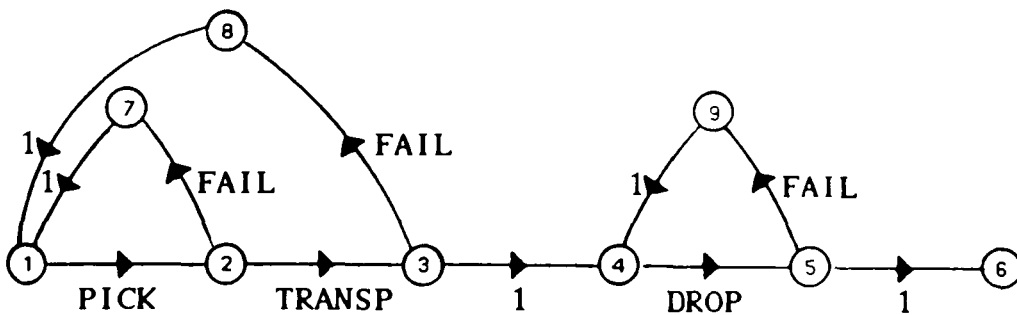


Figure 6.10 - Buffered version of Figure 2.4

For simple program execution, buffering is not necessary as there is only one parameter (the executable routine) being considered, unlike during the analysis where both time and probability are being dealt with simultaneously. However, if the matrix already exists in a buffered state so as to conform with the analysis flowgraph, then we have something like {6.13} representing figure 6.10.

$$\begin{bmatrix}
 0 & \text{PICK} & 0 & 0 & 0 & 0 & | & 0 & 0 & 0 \\
 0 & 0 & \text{TRANSP} & 0 & 0 & 0 & | & \text{FAIL} & 0 & 0 \\
 0 & 0 & 0 & 1 & 0 & 0 & | & 0 & \text{FAIL} & 0 \\
 0 & 0 & 0 & 0 & \text{DROP} & 0 & | & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 1 & | & 0 & 0 & \text{FAIL} \\
 \hline
 0 & 0 & 0 & 0 & 0 & 0 & | & 0 & 0 & 0 \\
 1 & 0 & 0 & 0 & 0 & 0 & | & 0 & 0 & 0 \\
 1 & 0 & 0 & 0 & 0 & 0 & | & 0 & 0 & 0 \\
 0 & 0 & 0 & 1 & 0 & 0 & | & 0 & 0 & 0
 \end{bmatrix} \quad (6.13)$$

As shown above {6.13} may now be partitioned into four separate parts, according to the form of {6.14}, where each partition represents a particular subset of the digraph.

$$\left[ \begin{array}{c|c} \mathbf{A} & \mathbf{B} \\ \hline \mathbf{C} & \mathbf{0} \end{array} \right] \quad (6.14)$$

The partitioned subsets are:

**A:** Forward paths.

**B:** Recovery routines.

**C:** Feedback buffer paths.

**0:** Null.

Now a **A** can be stored as a single 5 element vector, **B** as a 5 by 3 matrix and **C** as a 4 by 6 matrix. This reduces the storage of 81 elements, of what was previously a 9 by 9 matrix, down to 44 elements.



### 6.3 Aspects of Parallel Processing

One of the main functions of task level programming is the necessity to execute several tasks simultaneously. It should not be necessary to consider this at object level, where all such program routines should be capable of running autonomously. Where several such object level routines may need to be executed simultaneously, the decision as to when and where they are to be run is made at the task level.

This distinction is best illustrated by using the well known example of the dining philosophers. For those unfamiliar with this; try to imagine four philosophers gathered round a table on which exists one communal bowl of food. Four forks are provided, but two are required for eating. Each philosopher will spend a random amount of time thinking, then when hungry he must pick up a left and right fork (in that order), eat and then return the forks in the same order.

This problem can be solved algorithmically by insisting that only three philosophers at any one time may be allowed to be hungry. A modula-2 program for this process capable of concurrent execution consists of over 200 lines of source code (not including IO statements) [Hewitt & Frank, 1989].

The basic flowgraph for one philosopher is as shown in figure 6.11, together with the executable procedure and sensor parameter matrices of figure 6.12.

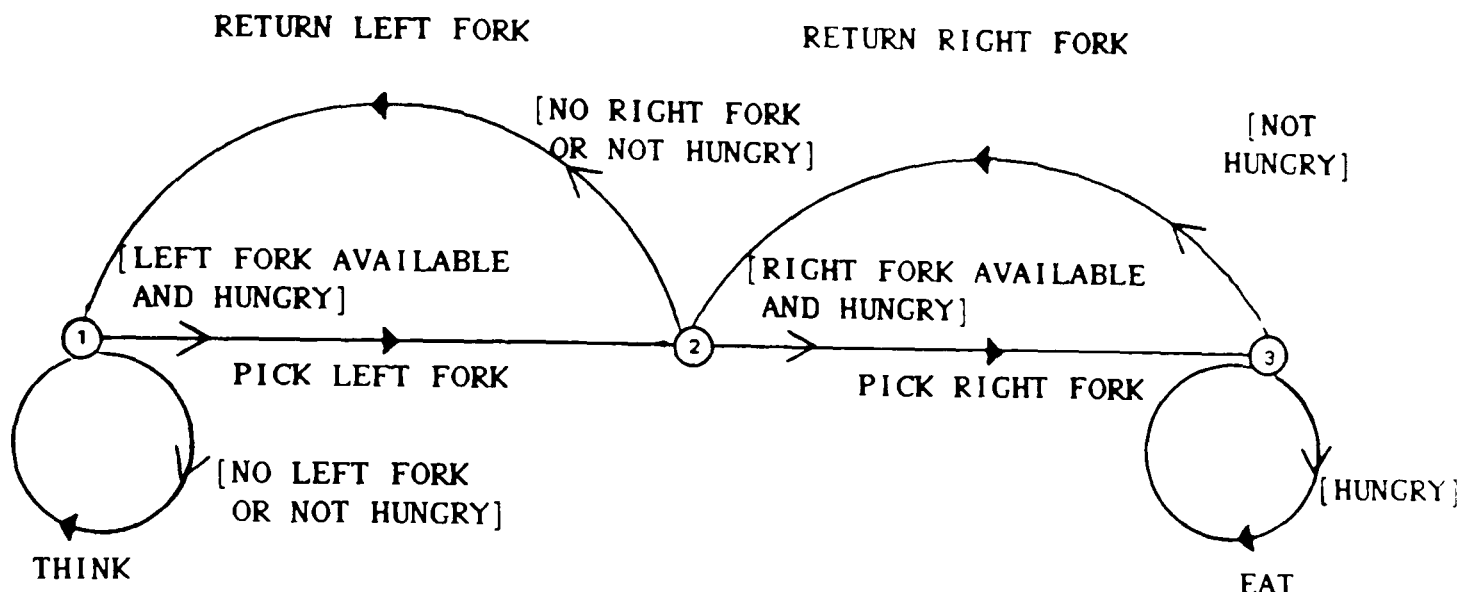


Figure 6.11 - Flowgraph for a Dining Philosopher.

$$P = \begin{bmatrix} \text{THINK} & \text{PICK LEFT FORK} & 0 \\ \text{RETURN LEFT FORK} & 0 & \text{PICK RIGHT FORK} \\ 0 & \text{RETURN RIGHT FORK} & \text{EAT} \end{bmatrix} \quad S = \begin{bmatrix} \text{NO LEFT FORK OR NOT HUNGRY} & \text{LEFT FORK AND HUNGRY} & 0 \\ \text{NO RIGHT FORK OR NOT HUNGRY} & 0 & \text{RIGHT FORK AND HUNGRY} \\ 0 & \text{NOT HUNGRY} & \text{HUNGRY} \end{bmatrix}$$

Figure 6.12 - Process and Sensor matrices for figure 6.11.

These matrices are common to all four philosophers and may be executed accordingly. If the sensing can be done in such a manner that no two philosophers can attempt to pick up a left or right fork simultaneously then the problem can be solved at object level. However, if at some stage conflicts must be resolved such that knowledge of previous events are required then the programming must be done at task level. For example, if two philosophers are hungry and reach for the same fork together how do we decide which one is to eat? Is it the hungriest?, the one who has so far eaten least? or the one who has done the most thinking?. If it is the first scenario, then provided we have a reliable method of sensing this the problem is simple. But if the one of the latter, then some form of historical data is required. In which case the action is no longer independant of previous events and the process becomes non-Markov.

This is one of the strongest differences between object level and task level programming. The advantage of task level programming is that execution can occur simultaneously without any of the restrictions needed for the previously mentioned Modula-2 program. Furthermore, when depicted in matrix form the program is considerably easier to read than 200 lines of algorithmic code.

## 6.4 Some Thoughts on Task Level Programming

Due to its inherently non-Markov construction, task level programming requires a notational structure which will allow the simultaneous operation of several events as well as the ability to control precedence. To achieve this, a full set of logic conditional nodes must be available as with the GERT notation.

For the purposes of this work it has been assumed that decisional data at task level will be available from a separate decision generating engine such as a knowledge based reasoning algorithm. However, this still leaves the question of a suitable notation for task level networks unresolved.

As discussed in chapter 2, the GERT notation is already established as a notation providing a choice in both logical input and output characteristics. Unfortunately GERT lacks the ease of transformation into matrix form, as was carried out in chapter 4 using the digraph notation.

The rest of this section is devoted to the introduction of a simple method of input weighting to enable the nodes of a digraph to behave as logic gates and yet maintain the isomorphism between flowgraph and transition matrix to enable the matrix driven programming, previously used at object level, to be implemented at task level.

Given a simple node with two inputs as in figure 6.13, simple weighting factors  $w_1$ ,  $w_2$  and  $w_3$  (shown in parenthesis) can be associated with each of the path functions.

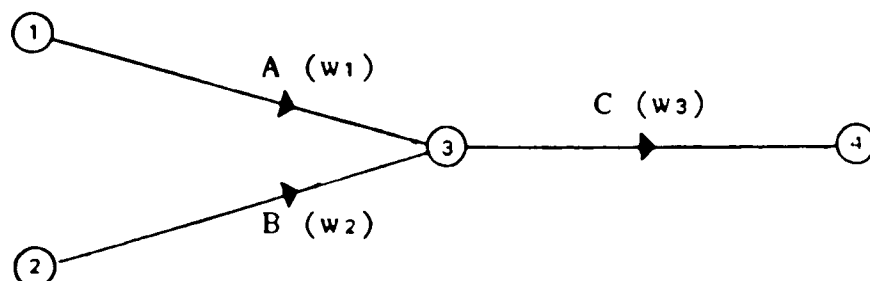


Figure 6.13 - Simple 2 input weighted network.

Using the conventional flowgraph notation figure 6.13 represents a network capable of executing the processes A then C, or B then C, but not both simultaneously (assuming the usual EX-OR property of node 3).

Now, if we allow the weighting factors  $w_1, w_2, w_3$  to have a value other than unity and set the criterion for a node to be activated by its inputs as being the absolute value of the sum of the input weighting values to be greater or equal to 1, ie.,

$$\text{For an } n \text{ input node to fire: } \left| \sum_{m=1}^n w_m \right| \geq 1 \quad (6.15)$$

then we have the basis for a full set of logic inputs. A similar technique is used in electrical transistor logic circuits and neural networks. Figure 6.14 shows the weighting factors to enable a) OR, b) AND and c) EX-OR structures.

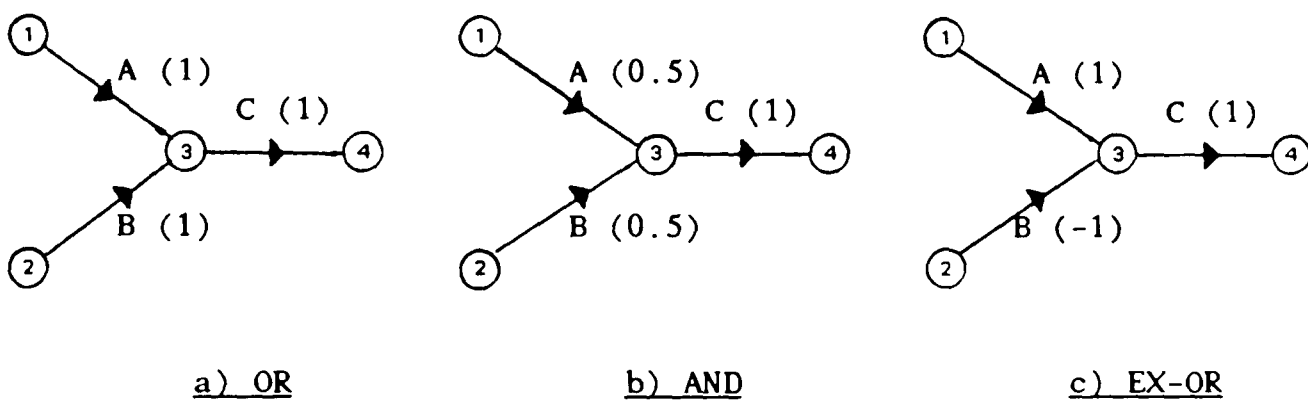


Figure 6.14 - Weighted logic structures.

Ignoring the functions A, B and C for the moment and concentrating on the action of the weighting factors alone, the corresponding transition matrices, as shown in figure 6.15, can be produced from the flowgraphs of figure 6.14.

$$\begin{array}{ccc}
 \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} & 
 \begin{bmatrix} 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} & 
 \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \\
 \text{a) OR} & 
 \text{b) AND} & 
 \text{c) EX-OR}
 \end{array}$$

Figure 6.15 - Weighting transition matrices.

Using a starting vector [ 1 1 0 0 ] to present an input to both nodes 1 and 2 simultaneously, premultiplying the transition matrix by the starting vector gives the resulting 'next' vectors [ 0 0 2 0 ], [ 0 0 1 0 ] and [ 0 0 0 0 ] respectively, according to the criterion of {6.15}. As expected, node 3 will be activated for the OR and AND cases, but not the EX-OR.

Similarly, using the starting vector [ 0 1 0 0 ] to trigger node 2 only produces the 'next' vectors [ 0 0 1 0 ], [ 0 0 0 0 ] and [ 0 0 1 0 ] respectively, i.e., only the OR and EX-OR weightings allow node 3 to fire.

Naturally such logic modules can be cascaded to produce larger networks allowing precedence control to be affected by means of the weighting factors. However, care must be taken when trying to simulate electrical logic circuits with this method as each path represents a propagation delay of one time unit. This means that extra nodes must be incorporated to maintain each node at a particular level of the execution sequence.

This provides a notation for depicting task level representations at a very basic level. It must not be forgotten that to be a truly task level implementation acting on reasoned error causes and other such conditional criteria, the weightings would be of a dynamic nature subject to change as dictated by whatever decisional control mechanism is employed.

## 6.5 Summary

This chapter has dealt with the use of simulation techniques for both simulation and task execution. The advantages of flowgraphs being their ease of transfer into matrix form, gives rise to a programming technique based on Markov chain theory. This allows both simulation and robot object level programming to be executed from the same algorithm. Unlike the Petri net simulation methods, the matrix format also lends itself to relatively easy mathematical analysis.

By extending these techniques to include time attributes, simulation to detect bottlenecks, estimation of robot collision etc., can be performed. More generally, other costs may be substituted for time to yield data on depreciation due to wear and tear etc.

It has also been shown that this two dimensional approach to programming has some distinct advantages over the conventional notation for computer programming. The restrictions in readability when using a sequence of programming statements reading down the page are not present when programming is carried out in the above matrix method.

Now the question must be asked: "why only two dimensional programming? why not three or more?". The difficulty in actually depicting a greater than two dimensional format on a two dimensional paper page or computer screen would seem to defeat the object of improved readability of code, not to mention the task of mathematical analysis of such a scheme. However, as far as the computer itself is concerned there is no reason whatsoever why this should not be investigated. This is however, the subject of further research and will not be discussed here.

Some thought has also been given to the extension of these object level programming techniques to the non-Markov task level. A system of weighting the inputs so as to obtain logic functional nodes similar to those used in the GERT notation, but at the same time using only simple digraph notation, has been introduced. This has the advantage of being easily transformed from a digraph depicting the workcell operation into a transition matrix useable in analysis and work cell execution.



## 7. OVERALL STRUCTURE AND IMPLEMENTATION

This chapter will deal with the integration of the mathematical analysis and object level programming techniques, with the intention of providing the necessary tools for a complete interactive programming system. It will be assumed that the necessary sensor decision and statistical data will be available from some form of AI or knowledge based system.

### 7.1 The New Model

Propose to any Englishman any principle or any instrument, however admirable, and you will observe that the whole effort of the English mind is directed to find a difficulty, a defect, or an impossibility in it. If you speak to him of a machine for peeling a potato, he will pronounce it impossible; if you peel a potato with it before his eyes, he will declare it useless because it will not slice a pineapple.

Charles Babbage.

Figure 7.1 shows the block diagram of the necessary structure to provide both a basic analysis package and a real time programming package together with the necessary processing and feedback from the robot and an associated knowledge base. During program operation, the knowledge base must resolve basic sensor data into object level decisions before communicating this data to the real time programming package. Similarly, statistical data compiled during program operation is continuously passed on to affect modification of the analysis model. This allows the user to identify problems during run-time and to modify the flowgraph and object level program accordingly.

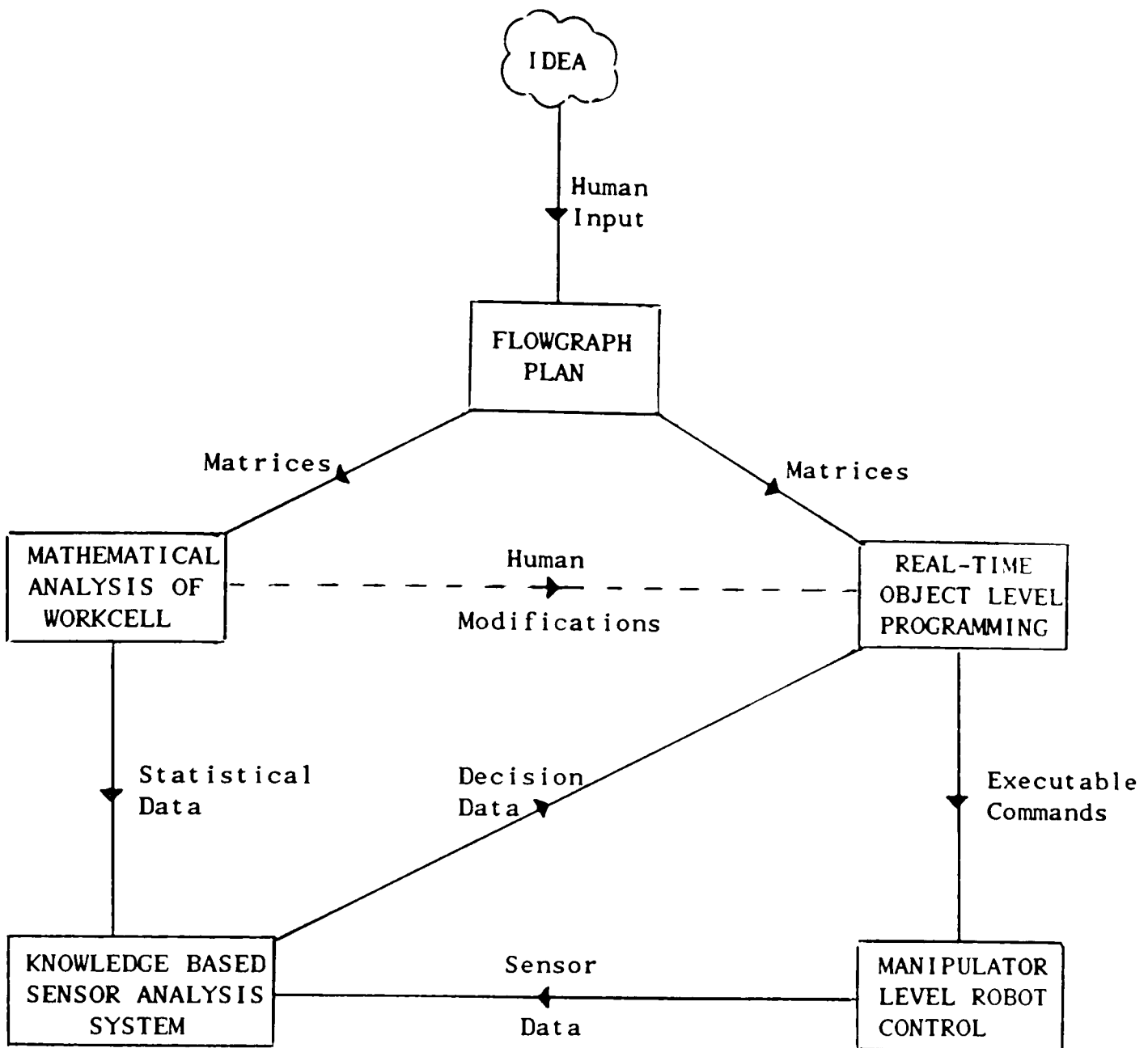


Figure 7.1 - Overall Interactive System Model

The top three boxes of figure 7.1 represent the sections resulting from the previous chapters of this thesis. The Manipulator level section is basically the robot and its associated controller, the manipulator level programming being conducted in a robot control language such as VAL II. The knowledge based system results from part of the work done by Ghis [Ghis, 1989], Song [Song, 1988] and Halloran [Halloran, 1989].

## 7.2 Selected Case Studies

The following two case studies will involve an analysis of the flowgraph using the Modula-2 programs developed for this purpose (see appendix E). They will aim to show the capabilities of the matrix simulation and programming techniques. Both models represent actual robot workcells which have been operated in real-time.

### 7.2.1 Pick and Place Model

The first example of a working model was introduced in chapter 3. Figure 3.4 gives the flowgraph for this simple pick and place model from which a toll (time) average of 16.39 seconds was calculated. This was achieved using the steady state value of the derivative of the transfer function found using Masons theorem. These same results, along with other useful data, will now be found using the matrix techniques developed in the later chapters.

The flowgraph of figure 3.4 is repeated in figure 7.2 with the addition of buffer paths as per the criteria outlined in section 6.2.4.

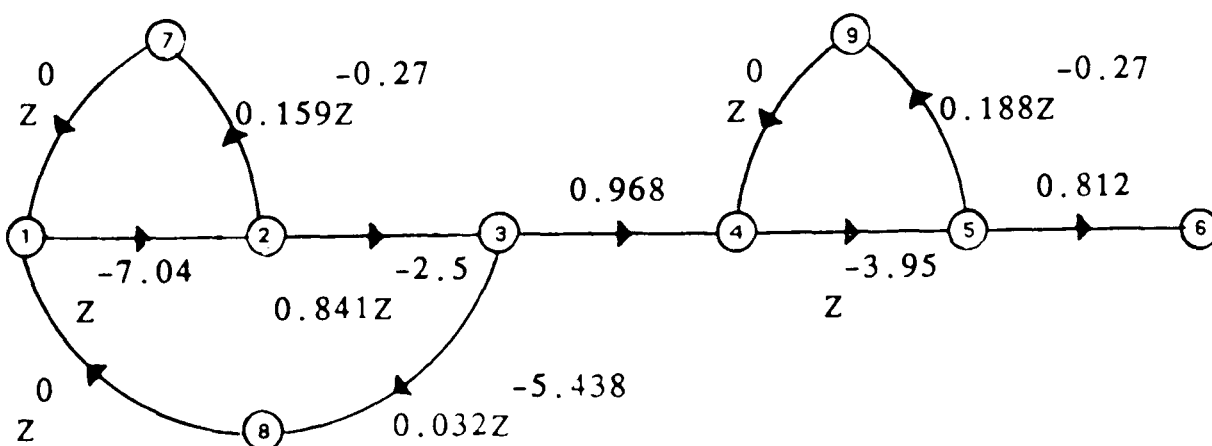


Figure 7.2 - Pick and Place Flowgraph

Using the Modula-2 programs of appendix E the data for the stochastic transition matrix {7.1} gives rise to the flow and variance matrices, {7.2} and {7.3} respectively.

$$\begin{bmatrix}
 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0.841 & 0 & 0 & 0 & 0.159 & 0 & 0 \\
 0 & 0 & 0 & 0.968 & 0 & 0 & 0 & 0.032 & 0 \\
 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0.812 & 0 & 0 & 0.188 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0
 \end{bmatrix} \quad (7.1)$$

FLOW matrix is:

$$\begin{bmatrix}
 1.23E+0 & 1.23E+0 & 1.03E+0 & 1.23E+0 & 1.23E+0 & 1.00E+0 & 1.95E-1 & 3.31E-2 & 2.32E-1 \\
 2.28E-1 & 1.23E+0 & 1.03E+0 & 1.23E+0 & 1.23E+0 & 1.00E+0 & 1.95E-1 & 3.31E-2 & 2.32E-1 \\
 3.93E-2 & 3.93E-2 & 1.03E+0 & 1.23E+0 & 1.23E+0 & 1.00E+0 & 6.25E-3 & 3.31E-2 & 2.32E-1 \\
 0.00E+0 & 0.00E+0 & 0.00E+0 & 1.23E+0 & 1.23E+0 & 1.00E+0 & 0.00E+0 & 0.00E+0 & 2.32E-1 \\
 0.00E+0 & 0.00E+0 & 0.00E+0 & 2.32E-1 & 1.23E+0 & 1.00E+0 & 0.00E+0 & 0.00E+0 & 2.32E-1 \\
 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 1.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 \\
 1.23E+0 & 1.23E+0 & 1.03E+0 & 1.23E+0 & 1.23E+0 & 1.00E+0 & 1.20E+0 & 3.31E-2 & 2.32E-1 \\
 1.23E+0 & 1.23E+0 & 1.03E+0 & 1.23E+0 & 1.23E+0 & 1.00E+0 & 1.95E-1 & 1.03E+0 & 2.32E-1 \\
 0.00E+0 & 0.00E+0 & 0.00E+0 & 1.23E+0 & 1.23E+0 & 1.00E+0 & 0.00E+0 & 0.00E+0 & 1.23E+0
 \end{bmatrix}$$

(7.2)

VARIANCE matrix is:

$$\begin{bmatrix}
 2.81E-1 & 2.81E-1 & 3.42E-2 & 2.85E-1 & 2.85E-1 & 0.00E+0 & 2.33E-1 & 3.42E-2 & 2.85E-1 \\
 2.81E-1 & 2.81E-1 & 3.42E-2 & 2.85E-1 & 2.85E-1 & 0.00E+0 & 2.33E-1 & 3.42E-2 & 2.85E-1 \\
 5.57E-2 & 5.57E-2 & 3.42E-2 & 2.85E-1 & 2.85E-1 & 0.00E+0 & 8.65E-3 & 3.42E-2 & 2.85E-1 \\
 0.00E+0 & 0.00E+0 & 0.00E+0 & 2.85E-1 & 2.85E-1 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 2.85E-1 \\
 0.00E+0 & 0.00E+0 & 0.00E+0 & 2.85E-1 & 2.85E-1 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 2.85E-1 \\
 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 \\
 2.81E-1 & 2.81E-1 & 3.42E-2 & 2.85E-1 & 2.85E-1 & 0.00E+0 & 2.33E-1 & 3.42E-2 & 2.85E-1 \\
 2.81E-1 & 2.81E-1 & 3.42E-2 & 2.85E-1 & 2.85E-1 & 0.00E+0 & 2.33E-1 & 3.42E-2 & 2.85E-1 \\
 0.00E+0 & 0.00E+0 & 0.00E+0 & 2.85E-1 & 2.85E-1 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 2.85E-1
 \end{bmatrix}$$

(7.3)

The toll data {7.4} can be entered producing the toll average matrix {7.5} and its corresponding element sum which represents the complete toll average for the flowgraph.

$$\begin{bmatrix}
 0 & 7.04 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 2.5 & 0 & 0 & 0 & 0.27 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 5.438 & 0 \\
 0 & 0 & 0 & 0 & 3.95 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0.27 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
 \end{bmatrix} \quad \{7.4\}$$

Toll average matrix is:

$$\begin{bmatrix} 0.00E+0 & 8.65E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 \\ 0.00E+0 & 0.00E+0 & 2.58E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 5.27E-2 & 0.00E+0 & 0.00E+0 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 1.80E-1 & 0.00E+0 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 4.86E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 6.25E-2 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 & 0.00E+0 \end{bmatrix}$$

(7.5)

mean toll value is: 1.6389905E+1

In section 6.2.4 some ideas on the partitioning of matrices were discussed. Some of the advantages of using partitioned matrices will now become apparent.

If {7.1} and {7.4} are each partitioned into three separate 3 by 3 matrices corresponding to the three recovery loops of figure 3.4 then these can be tackled separately as follows:

Matrices {7.6} and {7.9} represent the respective stochastic transition and toll values for the single loop of figure 7.3. This gives a toll average of 10.922.

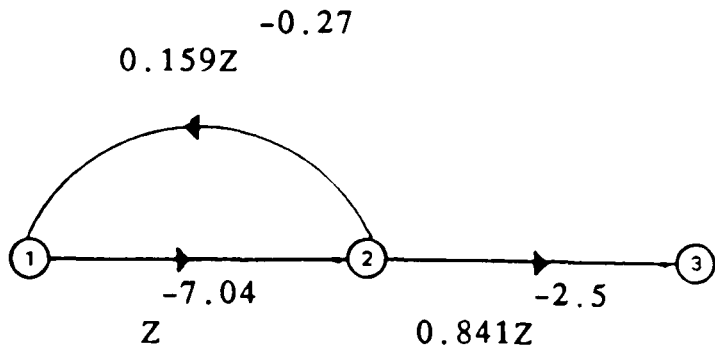


Figure 7.3 - First Loop

Entering the stochastic transition matrix:

$$\begin{bmatrix} 0 & 1 & 0 \\ 0.159 & 0 & 0.841 \\ 0 & 0 & 0 \end{bmatrix} \quad (7.6)$$

FLOW matrix is:

$$\begin{bmatrix} 1.19E+0 & 1.19E+0 & 1.00E+0 \\ 1.89E-1 & 1.19E+0 & 1.00E+0 \\ 0.00E+0 & 0.00E+0 & 1.00E+0 \end{bmatrix} \quad (7.7)$$

VARIANCE matrix is:

$$\begin{bmatrix} 2.25E-1 & 2.25E-1 & 0.00E+0 \\ 2.25E-1 & 2.25E-1 & 0.00E+0 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 \end{bmatrix} \quad (7.8)$$

Entering the toll matrix:

$$\begin{bmatrix} 0 & 7.04 & 0 \\ 0.27 & 0 & 2.5 \\ 0 & 0 & 0 \end{bmatrix} \quad (7.9)$$

Toll average matrix is:

$$\begin{bmatrix} 0.00E+0 & 8.37E+0 & 0.00E+0 \\ 5.10E-2 & 0.00E+0 & 2.50E+0 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 \end{bmatrix} \quad (7.10)$$

mean toll value is: 1.092E+1

Incorporating the results of the last operation and combining the first two loops of the flowgraph gives a mean toll of 11.46

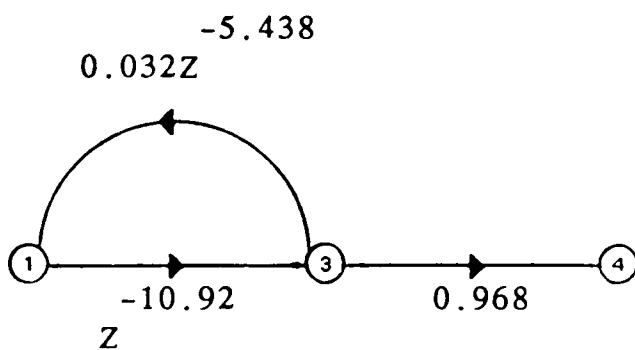


Figure 7.4 - Combined First and Second Loop

Entering the new stochastic transition matrix:

$$\begin{bmatrix} 0 & 1 & 0 \\ 0.032 & 0 & 0.968 \\ 0 & 0 & 0 \end{bmatrix} \quad (7.11)$$

FLOW matrix is:

$$\begin{bmatrix} 1.03E+0 & 1.03E+0 & 1.00E+0 \\ 3.31E-2 & 1.03E+0 & 1.00E+0 \\ 0.00E+0 & 0.00E+0 & 1.00E+0 \end{bmatrix} \quad (7.12)$$



VARIANCE matrix is:

$$\begin{bmatrix} 3.42E-2 & 3.42E-2 & 0.00E+0 \\ 3.42E-2 & 3.42E-2 & 0.00E+0 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 \end{bmatrix} \quad (7.13)$$

Entering the new toll matrix:

$$\begin{bmatrix} 0 & 10.92 & 0 \\ 5.438 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (7.14)$$

Toll average matrix is:

$$\begin{bmatrix} 0.00E+0 & 1.13E+1 & 0.00E+0 \\ 1.80E-1 & 0.00E+0 & 0.00E+0 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 \end{bmatrix} \quad (7.15)$$

mean toll value is: 1.146E+1

Similarly, from {7.16} and {7.19}, the mean toll for the final loop is found to be 4.91.

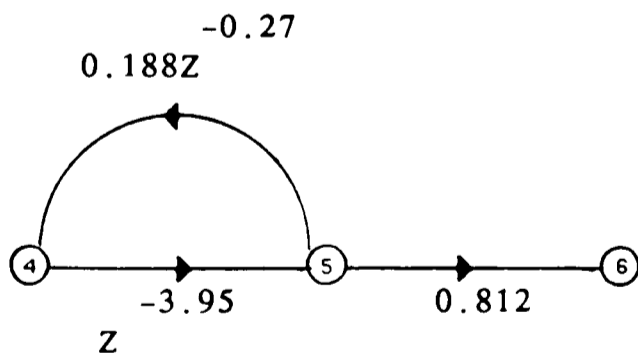


Figure 7.5 - Final Loop

$$\begin{bmatrix} 0 & 1 & 0 \\ 0.188 & 0 & 0.812 \\ 0 & 0 & 0 \end{bmatrix} \quad (7.16)$$

FLOW matrix is:

$$\begin{bmatrix} 1.23E+0 & 1.23E+0 & 1.00E+0 \\ 2.32E-1 & 1.23E+0 & 1.00E+0 \\ 0.00E+0 & 0.00E+0 & 1.00E+0 \end{bmatrix} \quad (7.17)$$

VARIANCE matrix is:

$$\begin{bmatrix} 2.85E-1 & 2.85E-1 & 1.11E-16 \\ 2.85E-1 & 2.85E-1 & 0.00E+0 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 \end{bmatrix} \quad (7.18)$$

Entering the new toll matrix:

$$\begin{bmatrix} 0 & 3.95 & 0 \\ 0.27 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (7.19)$$

Toll average matrix is:

$$\begin{bmatrix} 0.00E+0 & 4.86E+0 & 0.00E+0 \\ 6.25E-2 & 0.00E+0 & 0.00E+0 \\ 0.00E+0 & 0.00E+0 & 0.00E+0 \end{bmatrix} \quad (7.20)$$

mean toll value is: 4.927E+0

Finally, the last two results can be added together to give the total mean toll of  
 $11.46 + 4.93 = 16.39$

This is exactly the same value calculated previously with the full 9 by 9 matrices and by the algebraic method in chapter 3. There are several advantages to partitioning the matrices in this manner. As mentioned previously, storage space must be provided for large matrices, and if these are very sparse as in {7.1} and {7.4} then a great deal of room is taken up by zero's. Furthermore, many computer languages, of which Modula-2 is one, do not allow dynamic arrays. This means that all the matrices to be used must be dimensioned at the time the program is written, even though at run time they may actually be very much smaller. If it could be guaranteed that all matrices to be used would be no larger than 3 by 3 say, then considerable savings in both storage space and execution time could be made. Finally, the rounding errors caused by the large number of calculations required to invert very large matrices can lead to numerical errors. The larger the matrix the more pronounced any ill-conditioning becomes. A detailed analysis of the underlying reasons behind ill-conditioning in matrix operations can be found in Rice [Rice, 1981].

### 7.2.2 An Intelligent Robot Workcell

A workcell consisting of a Puma Robot with a sensory textile gripper [Kemp et al, 1986] has recently been used to form the basis of a project to build up error recovery strategies using an AI/knowledge base system. The basic task consists of destacking and laying up a single panel of knitted fabric after which a fusible motif is applied to the fabric panel. A large number of (relatively unpredictable) errors may occur in this type of workcell, making it an ideal subject for this research.

With regard to the techniques developed here, simulation and object level programming of the workcell are the two main factors. Such methods must be combined with the knowledge base and robot manipulator level programming system as depicted in figure 7.1. The previous case study concentrated mainly on the simulation aspects using statistical data already gathered. No such data has so far been compiled for these operations, however this case study is ideal for illustrating the object level programming aspect.

Figure 7.6 shows the flowgraph for the ply separation, pick and place operations. The motif handling part is very similar and so will not be dwelt upon here.

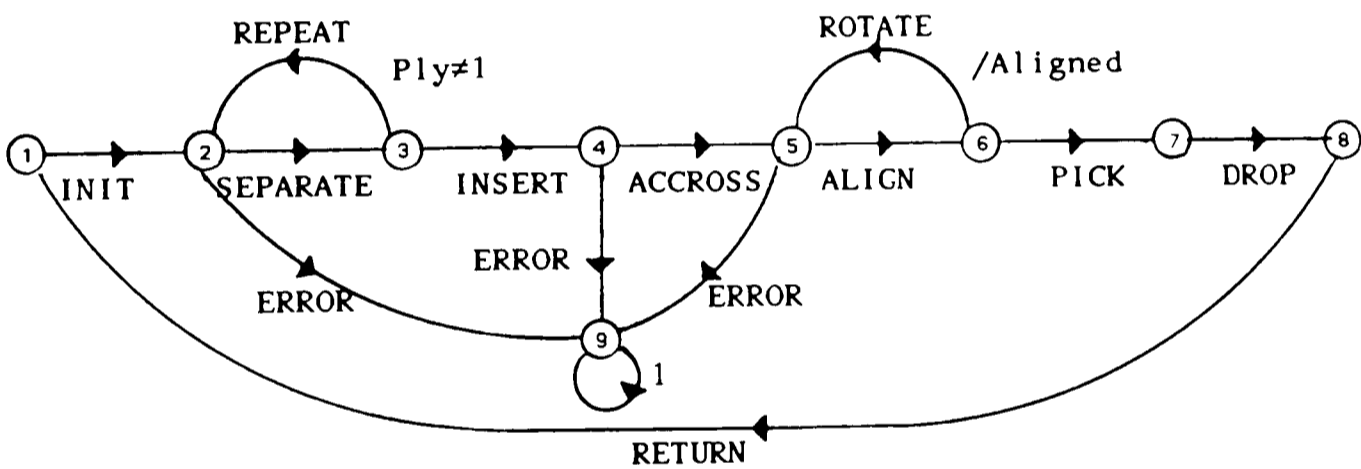


Figure 7.6 - Fabric Ply Separation Workcell Flowgraph.

At manipulator level continuous sensing is carried out to provide fine adjustments in alignment during the ALIGN routine, finding the stack side within the ACCROSS routine and the stack edge during INSERT. The ERROR routine is that which must either call the operator or seek further data before continuing. The routine matrix R, for this flowgraph is as follows:

0	INIT	0	0	0	0	0	0	0
0	0	SEPARATE	0	0	0	0	0	ERROR
0	REPEAT	0	INSERT	0	0	0	0	0
0	0	0	0	ACROSS	0	0	0	ERROR
0	0	0	0	0	ALIGN	0	0	ERROR
0	0	0	0	ROTATE	0	PICK	0	0
0	0	0	0	0	0	0	DROP	0
RETURN	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

Using the starting vector  $U_0$

$$U_0 = [ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0 ]$$

and a sensor decision matrix S

0	1	0	0	0	0	0	0	0
0	0	$\bar{E}$	0	0	0	0	0	E
0	Ply $\neq$ 1	0	Ply=1	0	0	0	0	0
0	0	0	0	$\bar{E}$	0	0	0	E
0	0	0	0	0	$\bar{E}$	0	0	E
0	0	0	0	$\overline{\text{Aligned}}$	0	Aligned	0	0
0	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1

Now  $U_1 = U_0 (R \square S)$

If no errors are encountered then the main forward path of the flowgraph in figure 7.6 will be followed, ie., the routines INIT, SEPARATE, INSERT, ACCROSS, ALIGN, PICK, DROP and RETURN will be executed in that order. Only if errors occur during execution, causing a unity element to appear in the sensor matrix  $S$  in a position other than the main off-diagonal, will there be any variation to this plan. The paths denoted in figure 7.6 as "ERROR" correspond to an E entry in the sensor matrix and represent errors of an unknown form. These errors would result in the cessation of activity until the problem is rectified by the operator. The program will in fact stop due to the trapping state of node 9.

In cases where unexpected errors could occur at any time, a path from each node to node 9 will cater for this. This would result in a set of entries in column 9 of both the routine matrix  $R$  and the sensor matrix  $S$ .

A module of Modula-2 procedures for the execution of a matrix programming system are included in the second set of listings in appendix E.

### 7.3 Summary

Due to the dual parameter nature of the toll and probability matrices, the simulation matrices are likely to be far larger than the routine and sensor matrices required for programming. However, the basic Markov chain philosophy is exactly the same.

It is not unusual for matrices representing such processes to be extremely large and sparse. Duff describes a collection of industrial 'real life' sparse matrices ranging from 76 to over one million entries. Methods of compact storage together with some algorithms (coded in FORTRAN) for reading compacted matrices stored in several different formats are also discussed. [Duff et al, 1989]. However, for systems with insufficient memory to cope with matrices of this size, partitioning of the process is possible.

Having introduced a completely new programming methodology, to complement a similar form of simulation and analysis using matrices, all which remains is to evaluate the technique. This is done throughout the next chapter by comparison to existing simulation and programming systems.

## 8. EVALUATION.

In the eyes of its mother every black beetle is a gazelle.

Old Arabic proverb.

### 8.1 A Markov Simulator.

The simulation and modelling techniques discussed in the preceding chapters are now put together in the form of computer software procedures within a module of the programming language Modula-2.

#### 8.1.1 Modula-2 Procedures.

A description of each of the Modula-2 procedures given in appendix E follows. In all cases the matrices used are square with dimension  $n$ . The dimension is set when the first matrix is read in and can only be changed during subsequent read operations. Within the simulation module all user-accessible matrix elements, are of type real. The opposite is the case for the programming module. Here only string and cardinal variables are available for matrix elements. This has the advantage of preventing the user from inadvertently entering simulation data when in the programming mode, and vice-versa.

These procedures are the basic building-block routines required for producing Markov chain simulation programs, and as such no error trapping or syntax checking is included in either of these procedures or any of the example main programs.



### matread(A)

Requests the dimension of the square matrix to be read in, then puts this data into the real variable  $n$  before reading in each of the real values for each element of the  $n$  by  $n$  matrix  $A$ . The elements of  $A$  are read in row by row.

### matwrite(A)

Prints out a square matrix  $A$ , of dimension  $n$ . Each element of  $A$  is a real variable which is printed with 3 significant figures in exponential notation. This may be altered, if desired, by changing the format of the `WrReal` statement.

### I(A)

Sets each element of the main diagonal of matrix  $A$  to the real constant 1.0 and every other element to 0.0 thereby creating an identity matrix in  $A$  of dimension  $n$ .

### M(A)

Creates a 'mirror' matrix  $A$  with unity elements along the inverse main diagonal (ie. from top right hand side to bottom left), and zero elements elsewhere. This has the effect of producing a rotation of a matrix  $B$  when the transpose of  $B$  is both post and pre-multiplied by the mirror matrix  $A$ .

### matadd(A,B,C)

Adds two real, square,  $n$  by  $n$  matrices  $A$  and  $B$  leaving the result in matrix  $C$ . Matrices  $A$  and  $B$  are left unchanged.

### matsub(A,B,C)

Subtracts the matrix  $B$  from  $A$  leaving the result in  $C$ . Matrices  $A$  and  $B$  are left unchanged.

### matsum(x,A)

Finds the sum of all the elements of the matrix  $A$  and puts the result in the real variable  $x$ .

### matconmul(A,B,C)

Performs congruent multiplication (that is element by corresponding element multiplication) between matrices  $A$  and  $B$  leaving the result in matrix  $C$ . Matrices  $A$  and  $B$  are left unchanged.

### matmul(A,B,C)

Performs standard matrix multiplication between matrices  $A$  and  $B$ , leaving the result in matrix  $C$ . Matrices  $A$  and  $B$  are left unchanged.

scalmul(x,A,B)

Multiplies the matrix A by the real scalar variable x leaving the result in matrix B. Matrix A and variable x are left unchanged.

transp(A,B)

Puts the transpose of matrix A into matrix B, leaving matrix A unchanged.

matinv(A,B)

Inverts square n by n matrix A leaving result in matrix B without changing A.

diag(A,B)

Diagonalizes (sets all elements other than those along the main diagonal to zero) square, n by n matrix A, leaving the result in matrix B without changing A.

matflow(A,B)

Finds the characteristic matrix by the formula  $B = [I-A]^{-1}$ . The elements of B represent the inter-nodal flows of the matrix A. The result is left in matrix B with A remaining unchanged.

### matvar(A,B)

Finds the statistical variance of the matrix A, leaving the result in matrix B. A remains unchanged.

### matlim(A)

Effectively raises the matrix A to the power of infinity by successive multiplication until a desired tolerance between two iterations has been achieved. This tolerance value may be changed by altering the value in the UNTIL ABS(x)<'value' statement. The resulting limiting matrix is left in matrix B.

## 8.1.2 Program Operation and User Guide.

The procedures listed in appendix E are of the JPI Modula-2 format [JENSEN and Partners, 1987] and are written for IBM PC and compatible devices. No additional co-processors or memory capability beyond that available with the standard PC (or clone) is needed, though improved performance may be achieved by the inclusion of an additional maths co-processor. The JPI Modula-2 is written for use with 64K of variable storage which limits the amount (or size) of the array dimensions. For very large modelling tasks, standard Modula-2 on VAX or SUN machines may be preferable.

The matrix evaluations illustrated throughout chapters 6 and 7 have been carried out using this programming system, and as such should suffice as example program runs. However, a brief interaction example session is provided in the next part of this chapter, intended as a basic 'user guide'.

The package consists of two basic modules: ANAL and DRIVE. ANAL performs the stochastic analysis by requesting the size of the square transition matrix followed by the probabilities which make up the stochastic transition matrix elements. This results in the *LIMITING*, *FLOW* and *VARIANCE* matrices. After this ANAL requests the *TOLL* matrix elements before giving the *average toll value* for the network.

Similarly, DRIVE requests the names of the program routines to form the *ROUTINE* matrix. These are the names which would be used by the host manipulator level language such as VAL, AML etc. After this the initial sensor values are entered (in real robot execution these would be available automatically) into the *SENSOR* matrix, followed by the starting vector (1 and 0 values only).

Execution is governed wholly by the state of the *SENSOR* matrix element values. These would normally be available directly from the robot controller, after sensor fusion or merging, and the implementation is therefore hardware dependant. Consequently, the example program whose listings appear in appendix E are written to expect these values from the keyboard for test and user familiarity purposes. Modifications needed to comply with the appropriate computer system used must be carried out by the user. In accordance with the communication protocols dictated by the host robot controller.

The following examples will take the reader through the actual execution sequence of both the ANAL and DRIVE programs. The narrative text will appear in *italics* to distinguish it from the actual input parameters and output code produced and displayed on the screen, or other interface device. The data entered via the keyboard during run time is preceded by a question mark prompt.

Using a similar example to that shown in section 4.5.3, but with two intersecting loops, we get the flowgraph of figure 8.1.

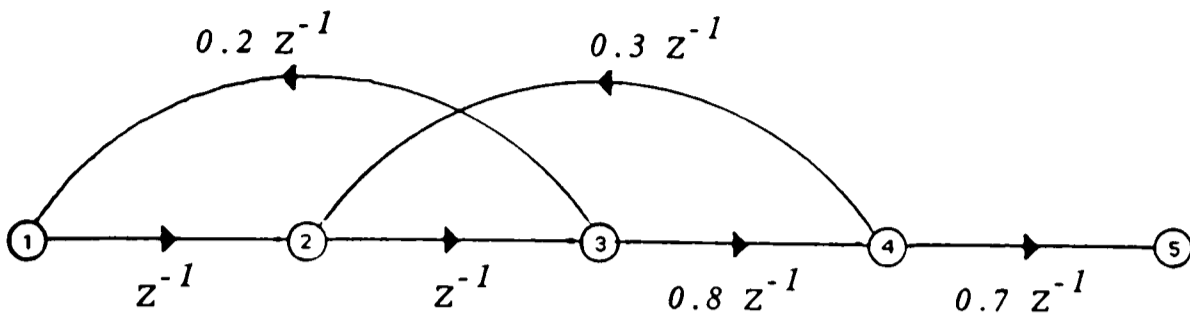


Figure 8.1 - Intersecting loop flowgraph.

This gives the resulting stochastic transition matrix  $P$  and toll matrix  $T$  accordingly.

$$P = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0.2 & 0 & 0 & 0.8 & 0 \\ 0 & 0.3 & 0 & 0 & 0.7 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad T = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The analysis package may be executed by typing ANAL on the keyboard with the disc loaded and the appropriate drive selected. On VAX systems running VMS it is necessary to type RUN ANAL for the equivalent execution. The program will start with a request for the size of the matrix (the number of nodes in the corresponding flowgraph).

Enter stochastic transition matrix

Enter matrix size = ?5

*The probabilities may now be entered for each element of the stochastic transition matrix. The data can be entered by following each element with a carriage return, alternatively a space may be used as a delimiter between values.*

Enter matrix elements

?0 1 0 0 0

?0 0 1 0 0

?0.2 0 0 0.8 0

?0 0.3 0 0 0.7

?0 0 0 0 0

LIMITING matrix is:

0.00E+0 4.94E-31 0.00E+0 0.00E+0 6.28E-31

0.00E+0 0.00E+0 4.94E-31 0.00E+0 0.00E+0

9.87E-32 0.00E+0 0.00E+0 3.95E-31 0.00E+0

0.00E+0 1.48E-31 0.00E+0 0.00E+0 1.89E-31

0.00E+0 0.00E+0 0.00E+0 0.00E+0 0.00E+0

FLOW matrix is:

1.36E+0 1.79E+0 1.79E+0 1.43E+0 1.00E+0

3.57E-1 1.79E+0 1.79E+0 1.43E+0 1.00E+0

3.57E-1 7.86E-1 1.79E+0 1.43E+0 1.00E+0

1.07E-1 5.36E-1 5.36E-1 1.43E+0 1.00E+0

0.00E+0 0.00E+0 0.00E+0 0.00E+0 1.00E+0

VARIANCE matrix is:

```
4.85E-1 1.40E+0 1.40E+0 6.12E-1 0.00E+0
4.85E-1 1.40E+0 1.40E+0 6.12E-1 0.00E+0
4.85E-1 1.40E+0 1.40E+0 6.12E-1 0.00E+0
1.72E-1 1.09E+0 1.09E+0 6.12E-1 0.00E+0
0.00E+0 0.00E+0 0.00E+0 0.00E+0 0.00E+0
```

*The three parameter matrices now show the limiting, flow and variance matrices. As would be expected for this example, the limiting matrix is approaching zero, with the flow matrix showing the greatest degree of congestion between nodes 2 and 3. Now the toll matrix may be entered in the same manner.*

Enter toll matrix

Enter matrix size = ?5

Enter matrix elements

?0 1 0 0 0

?0 0 1 0 0

?1 0 0 1 0

?0 1 0 0 0

?0 0 0 0 0

Toll AVERAGE matrix is:

```
0.00E+0 1.79E+0 0.00E+0 0.00E+0 0.00E+0
0.00E+0 0.00E+0 1.79E+0 0.00E+0 0.00E+0
3.57E-1 0.00E+0 0.00E+0 1.43E+0 0.00E+0
0.00E+0 5.36E-1 0.00E+0 0.00E+0 0.00E+0
0.00E+0 0.00E+0 0.00E+0 0.00E+0 0.00E+0
```

MEAN toll value is: 5.89286E+0



*In this case all the individual toll values were unity giving rise to the above average value of 5.893 seconds approximately. Now lets repeat the run, but with the transition toll between nodes 2 and 3 halved.*

*Initially we enter the probabilities exactly as before.*

Enter stochastic transition matrix

Enter matrix size = 5

Enter matrix elements

? 0 1 0 0 0

? 0 0 1 0 0

? 0.2 0 0 0.8 0

? 0 0.3 0 0 0.7

? 0 0 0 0 0

LIMITING matrix is:

0.00E+0 4.94E-31 0.00E+0 0.00E+0 6.28E-31

0.00E+0 0.00E+0 4.94E-31 0.00E+0 0.00E+0

9.87E-32 0.00E+0 0.00E+0 3.95E-31 0.00E+0

0.00E+0 1.48E-31 0.00E+0 0.00E+0 1.89E-31

0.00E+0 0.00E+0 0.00E+0 0.00E+0 0.00E+0

FLOW matrix is:

1.36E+0 1.79E+0 1.79E+0 1.43E+0 1.00E+0

3.57E-1 1.79E+0 1.79E+0 1.43E+0 1.00E+0

3.57E-1 7.86E-1 1.79E+0 1.43E+0 1.00E+0

1.07E-1 5.36E-1 5.36E-1 1.43E+0 1.00E+0

0.00E+0 0.00E+0 0.00E+0 0.00E+0 1.00E+0

VARIANCE matrix is:

```
4.85E-1 1.40E+0 1.40E+0 6.12E-1 0.00E+0
4.85E-1 1.40E+0 1.40E+0 6.12E-1 0.00E+0
4.85E-1 1.40E+0 1.40E+0 6.12E-1 0.00E+0
1.72E-1 1.09E+0 1.09E+0 6.12E-1 0.00E+0
0.00E+0 0.00E+0 0.00E+0 0.00E+0 0.00E+0
```

*Only this time when entering the toll matrix data, the 8<sup>th</sup> entry, corresponding to the path between nodes 2 and 3 is reduced to 0.5.*

Enter toll matrix

Enter matrix size = 5

Enter matrix elements

```
?0 1 0 0 0
?0 0 0.5 0 0
?1 0 0 1 0
?0 1 0 0 0
?0 0 0 0 0
```

Toll AVERAGE matrix is:

```
0.00E+0 1.79E+0 0.00E+0 0.00E+0 0.00E+0
0.00E+0 0.00E+0 8.93E-1 0.00E+0 0.00E+0
3.57E-1 0.00E+0 0.00E+0 1.43E+0 0.00E+0
0.00E+0 5.36E-1 0.00E+0 0.00E+0 0.00E+0
0.00E+0 0.00E+0 0.00E+0 0.00E+0 0.00E+0
```

MEAN toll value is: 5.00000E+0

Our new value for mean toll is 5, a reduction in overall process time of 0.893 seconds for a 0.5 second reduction in one path toll. Of course, as more data is gained from real run time data, the analysis can be repeated over and over to enable the process to be improved. The improved model may then be used as the basic data for the DRIVE program.

However, for this example we will use the continuous chain similar to that of section 4.5.2. In this case the three players pass a die to the left or to the right one place only, or throw again depending on their score. The flowgraph and process matrix are repeated in figure 8.2. At object level, only the decision is needed. The actual score is only important in that the decision to pass the die to the left or to the right or rethrow is derived from it.

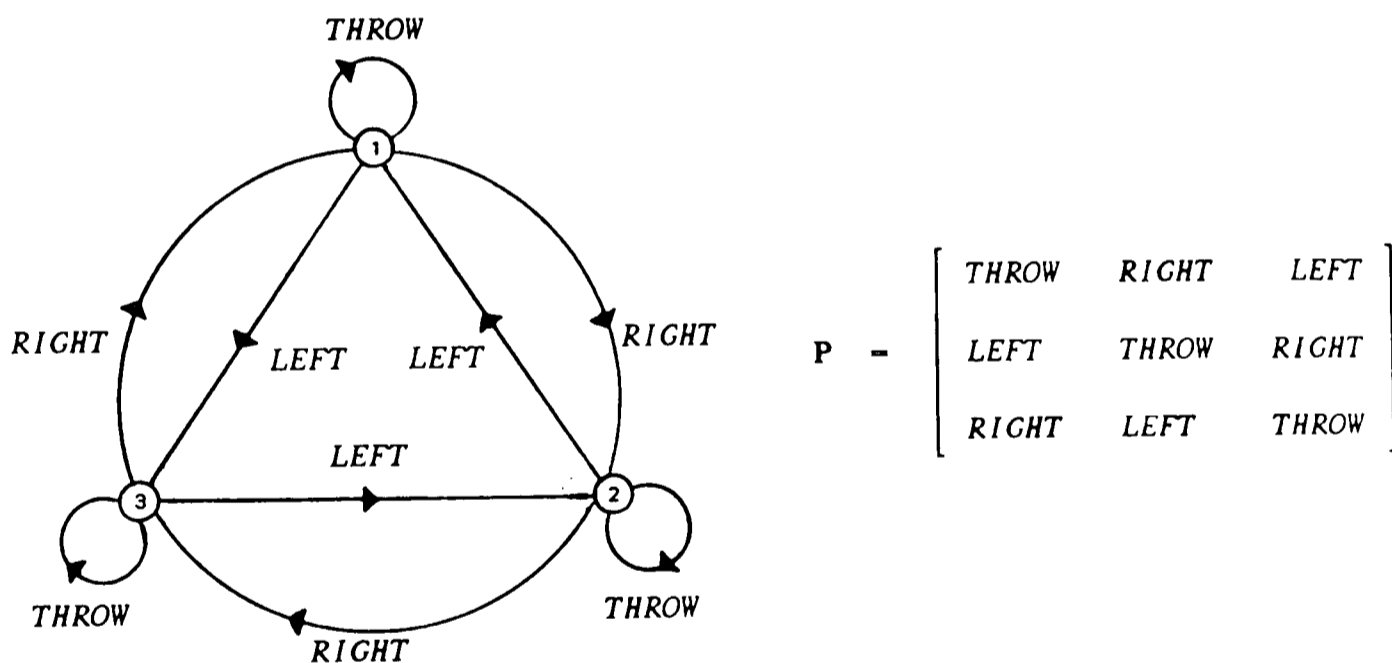


Figure 8.2 - Three dice players.

The user may start the program by typing DRIVE (again use RUN DRIVE when executing the program on VAX VMS systems). The process routines may then be entered after declaring the size of the matrix (number of nodes on the flowgraph).

Enter Routine matrix

Enter matrix size

?3

Enter matrix string elements

?THROW

?RIGHT

?LEFT

?LEFT

?THROW

?RIGHT

?RIGHT

?LEFT

?THROW

*Then the initial sensor matrix, for which we will have the first player throw the die, must be given together with the starting vector.*

Enter Initial sensor matrix

Enter 3 by 3 Sensor matrix (1, 0 values only)

?1 0 0

?0 0 0

?0 0 0

Enter starting vector

?1 0 0

THROW

*So, player one takes his turn which results in a score causing the die to be passed to the right. This information is provided by the sensor matrix.*

Enter 3 by 3 Sensor matrix (1, 0 values only)

?0 1 0

?0 0 0

?0 0 0

RIGHT

*Now the die is in the hands of the second player who we will have throw a score causing the die to be passed one place to the left and hence back to the first player.*

Enter 3 by 3 Sensor matrix (1, 0 values only)

?0 0 0

?1 0 0

?0 0 0

LEFT

*This done, we will have a score demanding player one also to pass to the left which brings the die round to player three.*

Enter 3 by 3 Sensor matrix (1, 0 values only)

?0 0 1

?0 0 0

?0 0 0

LEFT

*Player three then scores a six and so must throw again.*

Enter 3 by 3 Sensor matrix (1, 0 values only)

?0 0 0

?0 0 0

?0 0 1

THROW

*Normally this process would be controlled by the results of the die throwing. For example the sensor matrix values could be determined by a random process. To end the operation it is simply a matter of setting all the sensor values to zero. This causes the driving vector to reach a null state forcing the process to cease.*

Enter 3 by 3 Sensor matrix (1, 0 values only)

?0 0 0

?0 0 0

?0 0 0

Procedure Completed

*The S matrix values would normally be derived from sensor decisions by connecting the input of the program to the appropriate hardware IO module or manipulator level controller. Alternatively, if the parameters are to be delivered to a file then the DRIVE program inputs may be connected to that file using the MS-DOS command DRIVE < INFILE where INFILE is the file containing the input data. Similarly the resulting output parameters may be placed in a file by means of the command DRIVE > OUTFILE. Where OUTFILE is the filename of the file to contain the output results. These two can be combined using DRIVE < INFILE > OUTFILE to both input parameters from, and output data to the corresponding files [Microsoft, 1986].*

*When used with VAX systems, the Modula-2 source code for the ANAL and DRIVE programs are slightly different from those given in appendix E. This is because the present VAX implementation used is Logitech Modula-2 [Logitech, 1988] which differs considerably from the JPI Modula-2 [Jensen, 1987]. Their implementation is presently resident on the University of Hull, Robotics Research Unit micro-VAX network node designated SPOCK.*

## 8.2 A Comparative Study.

Having presented a new philosophy in object level programming and simulation, this chapter will comprise an attempt to compare and contrast the results of this work with currently available simulation and robot programming packages. Some of the techniques already implemented in such systems will be explored with a view to their possible use with the ideas put forward here.

### 8.2.1 Queueing Simulators.

The history of simulation programs for modelling queues is long but surprisingly little varied. As mentioned previously, many such simulation packages exist such as GASP, CSL etc., and are the topic of numerous surveys. Unfortunately, for simulation purposes, these are almost exclusively emulatory rather than analytical in nature.

The integration of such systems into a manufacturing environment for direct control of plant as well as background simulation is rapidly becoming an industrial reality. For example, DEDS (Discrete Event Dynamic Simulation) allows perturbation ie., "what if?" scenarios to be modelled without interfering with the normal running of the factory [Bryant, 1989]. On completion of such tests, real parameters may be changed to allow the necessary improvements to be made to the manufacturing process on the basis of the simulation results.



### 8.2.2 Network and Flowgraph Methods.

Most of the older simulation programs have an input method which the human operator can use by translating information from a graphical representation into numerical data. GPSS uses its own flowchart method and in the case of SLAM a network graph is used. Others such as SIMON, GSP and DEMOS use activity diagrams. [Torn, 1981].

More modern packages allow the user to actually enter a network graphically. STEM will handle most representations (Petri net, flowgraph etc). This runs on a SUN or VAX under a LOOPS object oriented environment. [Poplewell & Jaio, 1989].

The entering of data in matrix form can be laborious and time consuming, particularly where matrices are sparse, even with mathematical manipulation packages like MATLAB which have been designed for handling such data. Fortunately the use of spreadsheets such as LOTUS [Ingalsbe, 1987] have provided the computing community with a neat alternative. The connection of LOTUS with other software systems has been done before. The most recent example being the APL2 prototype system FUSION which translates LOTUS formulae into APL2 code [Friis, 1989].

According to Singh & Hindi very little work has been conducted in the field of temporal analysis of networks [Singht & Hindi, 1989] where perturbation (what if?) analysis could be used. The results of this work hopefully helps to fill this particular gap.

### 8.2.3 Petri Net Simulation Packages.

The recent surge of interest in Petri nets has given rise to a number of simulation packages. Considerable work has been done in the USA, particularly at the Rensselaer Polytechnic Institute (RPI), on Petri nets over the past few years. GTPNA (General Timed Petri Net Package) requires a textual rather than graphical input, and generates and analyses the Petri net reachability graph. Unfortunately the memory requirements prevent GTPNA from being used on PAWL SUNs [Robinson, 1989].

Based on the work of Molloy [Molloy, 1985] GSPN (Generalised Stochastic Petri Nets) was developed by Chiola [Chiola, 1987]. This led to the production of GreatSPN (Graphical editor & analyser for Timed Petri Nets). GreatSPN provides a graphics editor and a net validator and analyser which will run on SUN 2 workstations. Unfortunately, it is still not possible to model timed transitive Petri nets even with such powerful packages as GreatSPN [Chiola, 1987].

In addition to GreatSPN, under further development at RPI are other packages including SPNP (Stochastic Petri Net Package), a C-based system for generating reachability graph information [Robinson, 1989].

### 8.2.4 Geometrical Robot Animation Systems.

Many animated robot simulators are now available for geometrical modelling of robot workcells. Though these are not directly relevant to this work, some of the graphical techniques used are of interest.

The combination of a textual simulation language SIMAN and a CAD package allows the user to link both simulation statements and animated graphics. The simulation language SIMAN is of a BASIC type format allowing data to be entered directly from a flowchart, whilst the CINEMA part is a standard mouse driven graphics CAD package [Horrocks, 1989].

CimStation, developed by SILMA inc., is a simulation system intended for off line simulation and generation of geometrical data suitable for downloading to the robot for real-time execution. The code is translated into Cincinnati Milacron ROPS (Robot Off-line Programming System) format before execution by the robot controller [Craig, 1987]. This ability to simulate at manipulator level and then download the code for execution is becoming increasingly popular. However, nothing yet exists with the ability to do this at object level, let alone a combined simulation and programming system.

For the simulation and programming of flowgraphs some form of graphics editor would be desirable. This would allow the user to 'draw' a flowgraph on the monitor screen rather than have to enter a great deal of data into a matrix as is presently done with the Modula-2 programs. STEM has possibly the most appropriate format for this. Also, rather than two separate packages, as in the case of SIMAN/CINEMA, a better facility would be to be able to enter the control data along with the flowgraph.

### 8.3 Further Research.

On a general note, more investigation into robot programming, taking into account much of the techniques used in process control languages would be useful. Only then can robot programming be tackled in a manner which will allow the basic matrix programming ideas presented in this work to be transformed into a true object and task level programming system. Furthermore, as indicated in chapter 6, extension of these techniques to dimensions greater than 2 should be explored.

The additional control lines used in the extended flowgraph notation introduced by Taylor [Taylor, 1987] need to be investigated further if communication between robots operating in a multi robot cell are to be included. Though multi robot cells have been considered during this work, no attempt has been made to model or analyze the effects of communication delays, data errors etc between co-operating devices.

This work contains the basic tools, with a few Modula-2 procedures to facilitate a simple implementation, listed in appendix E. Some form of screen graphics facility, similar to that available on simulation packages like STEM, is needed to provide a fully interactive man-machine interface for such a robot modelling and control system.

The use of neural network systems could provide a powerful means of augmenting the distributed sensor methods presently employed in most robot workcells. These have been successfully used in the UK for monitoring railway level crossings [Sanders, 1990]. Similarly, in the USA, a neural network system called 'INFANT' (Interactive Network Functioning on Adaptive Neural Topographies) has been developed by Neurogen Inc. INFANT is a robot controller which learns by exploring its environment from whatever sensors are available [AMT, 1990]. This means that a high degree of error recovery is inherently built into the control algorithms as learning progresses.

Replacement of the manual method of updating the program by some form of automated program generation would be most valuable. This is not a straight forward task as it requires considerable care in its implementation. If it is possible to alter the program according to results simulated from data immediately obtained from the running workcell, then a very fast fine tuning arrangement may result. On the other hand, the system may become unstable, in that an apparent improvement which is in reality an error may be used to exacerbate that error.

## 9. CONCLUSIONS.

This work has endeavoured to address a number of problems associated with object level programming of robotic workcells. Ideas from many scientific disciplines have been called upon, including: electrical engineering, robotics, operations research, mathematics and computer science. Aspects of error recovery have been emphasised heavily in an attempt to include the nature of "real life" uncertain environments.

Many network notations have been explored and their relative merits discussed. To enable both mathematical analysis and object level robot programming to be implemented using the same basic methodology, the digraph has been chosen to illustrate the use of matrix techniques and Markov chain theory for these purposes. This has enabled stochastic analysis to be performed analytically rather than in an emulatory manner usual of most simulation programmes. Several new techniques have been introduced including: the use of isochronic plots, the positioning of sensors according to flow data, matrix differentiation without recourse to the usual differential calculus techniques etc.

When faced with error occurrences during run time, the detect, reject and repeat policy is usually the most cost effective. Only when an object has gathered a degree of added value whilst passing through several processes are more elaborate error recovery strategies likely to be worth consideration. Even then it is often more efficient to reject the unfinished object for 're-work' in a separate work station rather than tie up the robot in backward chaining and repair work thus retarding the progress of the entire production line as a consequence. Whichever strategy is to be implemented, the use of isochronic plots provides an aid to deciding the optimum configuration of workcell layout incorporating error recovery regimes with flow data highlighting potential bottlenecks.

A more thorough set of robot programming levels has been defined, which includes the use of sensor data as an inseparable part of the overall structure of robot control programming. Many of the attributes found in other (non-robot) programming languages, such as interrupts, concurrency etc., have been considered.

With regard to object level programming, a new technique using a matrix representation for all parameters has been introduced, based on Markov chain theory. This improves the readability of large programs and allows data to be stored and manipulated in a manner whose structure is also readable and clearly defined. This philosophy is not restricted to object level robot programming, and its extension to other forms of high level language processing is also explored.

A set of algorithms are provided which form the framework of a true object level programming system in which sensor integration and error recovery capability is an inherent part, rather than an addition to be appended afterwards. Many of the methods used are also applicable to task level programming, for which some extensions to the present object level notations have been introduced, and the door is now open to this field of research.

The main difference between object and task level programming is that of concurrent workcell control. At object level each workcell, containing one or more robots, is considered to be a separate entity whose activity is governed by the arrival and departure of objects into and out of the workcell. Only at task level is any consideration given to the overall control and sequencing of several cells simultaneously whose operation is not independent of one another. This difference in independence and inter-dependence is what differs between Markov and non-Markov systems, and is hence used as the delimiter between object and task level representations.

An attempt has been made to compare and contrast the ideas put forward in this work with those already realised within presently available implementations of robotic simulation and programming packages. However, most of these concentrate on only a few aspects of the field such as physical modelling, robot programming or simulation, rather than a combination of simulation and programming as a complete and integrated system.

A new model has been outlined in which the complete simulation and robot workcell control can be integrated to provide an overall management system having the potential for very fine tuning and optimisation without interruption of the cells operation during run time.



## REFERENCES.

- ABB - *Programming manual, Robot Control System S3* - Chapter 4.12: Interrupts - ASEA, May 1987.
- AHUJA. J.S. & K.P. Valavanis. - *Extended Petri Nets for Comprehensive Modelling of Flexible Manufacturing Systems*. - Technical Report No. 11., Robotics Laboratory, Northeastern University, Boston MA, November 1987.
- ALBUS. J.S., A.J. Barbera & M.L. Fitzgerald - Programming a Hierarchical Robot Control System - *NBS Prog: 6<sup>th</sup> International Conference on Industrial Robot Technology*. June 1982.
- AMT - Advanced Manufacturing Technology - *Technical Insights inc.*, Vol 11, No. 1, 15 January 1990.
- ARAKI. T., T. Kagimasa & N. Tokura - Relations of Flow Languages to Petri net Languages - *Theoretical Computer Science* - PP51-75, 1981.
- BACCELLI. F., C.A. Courcoubetis & M.I. Reiman - Construction of the Stationary Regime of Queues with Locking - *Stochastic Processes and their applications*, Vol 26, No. 2, November 1987.
- BAJPAI. A.C., L.R. Mustoe & D. Walker. - *Advanced Engineering Mathematics*. - Wiley, 1979.
- BAJPAI. A.C., L.R. Mustoe & D. Walker. - *Engineering Mathematics*. - Wiley, 1980.
- BEDWORTH. D.D. & J.E. Bailey - *Integrated Production Control Systems* - John Wiley, 1987.
- BELL. W. A. - *An Investigation of the Extension of Analytic GERT to Generalized Logical Structures* - MSc Thesis, Lehigh University, 1971.
- BERMAN. A. & R.J. Plemmons - *Nonnegative Matrices in the Mathematical Sciences* - Academic Press, 1979.
- BLUME. C. & W. Jakob - *Programming Languages for Industrial Robots* - Springer-Verlag, 1986.
- BOFFEY. T.B. - *Graph Theory in Operations Research*. - MacMillan, 1982.
- BONNER. S. & K.G. Shin. - *A Comparative Study of Robot Languages* - Computer, pp 82-96, Decmber 1982.
- BOOTHROYD. D.G., C. Poli & L.E. Murch - *Handbook of Feeding and Orienting Techniques*. - University of Masseurhusetts, 1978.
- BOUCHER. T.O. - Using Simulation to test the Feasibility of Robotic Assembly - *Computer and Industrial Engineering*, Vol 10, No. 1, pp 29-44, 1986.
- BRICKELL. F. - *Matrices & Vector Spaces* - George, Allen & Unwin, 1972.
- BRYANT. G.F. - The use of Discrete Events Simulations in Factory wide Control - *Modelling, Simulation and Control of Discrete Event Systems* - IEE Computing and Control Colloquium - Plymouth Polytechnic, December 1989.
- BUSACKER. R. G. & T. L. Saaty. - *Finite Graphs and Networks, An Introduction with Applications*. - Mc.Graw-Hill, 1965.

- CARRE. B. - *Graphs and Networks* - Clarendon 1979.
- CASH. C.R. & W.E. Williams - A simulation modelling approach for analysing robotic assembly cells. - *Proc. Winter Simulation Conf.* 1986.
- CHAPMAN. D & P.E. Agre - Abstract Reasoning as Emergent from Concrete Activity - *Workshop on Planning & Reasoning about Action*, Portland, Oregon 1986.
- CHIOLA. G - *GreatSPN Users Manual* - Version 1.3, Politecnico di Torino, September 1987
- COLL. J. - *BBC Microcomputer System Users Guide* - BBC, October 1984.
- COOKE. D.E. - PETRI Nets; A tool for Representing Concurrent Activities in Space Station Applications - *Advances in Intelligent Robotic Systems* (SPIE). Vol 851, 1987.
- COOPER. R.B. - *Introduction to Queueing Theory.* - Edward Arnold 1981.
- CRAIG. J.J. - Arc Welding Simulation Simplifies Programming - *Robotics World* - March 1987.
- DARZEN. E. - Data Modelling using Quantile Density Functions. - *Some Recent Advances in Statistics.* - J. Tiago De Oliveira (Ed.) - Academic Press, 1982.
- DENHAM. M.J. - A Petri net approach to Discrete Event Control - *Modelling, Simulation and Control of Discrete Event Systems* - IEE Computing and Control Colloquium - Plymouth Polytechnic, December 1989.
- DONALD. B.R. - Robot motion planning with uncertainty in the geometric models of the robot and environment: A formal framework for error detection and recovery - *Proc. International Conf. on Robotics and Automation* - IEEE, 1986.
- DOWDY. S. & S. WEARDEN. - *Statistics for Research.* - John Wiley, 1983.
- DUFF. I.S., R.G. Grimes & J.G. Lewis - Sparse Matrix Test Problems - *ACM Transactions on Mathematical Software* - Vol 15, No. 1, March 1989.
- DYER. K.D.F. - *The use of High Level Languages in Robot Programming* - University of Hull, MSc. Thesis, 1985
- EISENMAN. R.L. - *Matrix Vector Analysis.* - Mc. Graw - Hill, 1963.
- ELMAGHRABY. S.E. - *Activity Networks; Project Planning and Control by Network Models.* - Wiley 1977.
- ELSAYED. E.A. & T.O. Boucher - *Analysis & Control of Production Systems.* - Prentice-Hall, 1985.
- ESPOSITO. A. & M. Vento - A Structured Language for Multi-Layered Simulation Models in Factory Automation Systems - *IEEE Workshop on Languages for Automation*, pp 113-116, 1987.
- FIELDING. P.J., F. DiCesare, G. Goldbogen & A. Desroches - Intelligent Automated Error Recovery in Manufacturing Workstations - *Proc. International Symp. on Intelligent Control* - IEEE, 1987.
- FIRBY. R.J. - An Investigation into Reactive Planning in Complex Domains - *Proc. 6<sup>th</sup> National conf. on AI* - pp 202-206, 1987.
- FORSYTHE. A.I., T.A. Keenan, E.I. Organik & W. Stenberg. - *Computer Science, p542* - John Wiley, 1975

- FOX. B.R. & K.G. Kempf - Opportunistic Scheduling for Robotic Assembly. - *Proc. International Conf. on Robotics & Automation*. IEEE, 1985.
- FREEDMAN. P. & A. Malowany - The Analysis and Optimization of Repetition within Robot Workcell Sequencing Problems - *IEEE Transactions on Robotics and Automation* - pp 1276-1281, 1988.
- FRIIS. E.S. - A fusion of LOTUS 123 and APL2 - *APL89 Conf. Proc. p68-74*. - APL QUOTE QUAD, Vol 19, No. 4, ACM/SIGAPL, August 1989.
- FROMMHERZ. B & J. Hornberger - Automatic Generation of Precedence Graphs. - *Proc. International Symposium on Industrial Robots*, pp453-466. - IFS Ltd, April 1988.
- GHRIS. D - *Errors and Sensing in Autonomous Assembly Workcells* - MSc. Thesis, University of Hull, December 1989.
- GINI. G. & M. Gini - Robot Languages in the Eighties - *Robotic Assembly* - pp 189-200, Ed. K. Rathmill - IFS, 1984.
- GINI. M. - The Future of Robot Programming. - *Robotica*, vol 5, pp 235-246., 1987.
- GONNET. G.H. - *Handbook of Algorithms and Data Structures* - Addison-Wesley, 1984.
- GORDON. G. - *System Simulation*. - Prentice-Hall, 1969.
- GRIMALDI. R.P. - Chapter 13: Optimization & Matching. - *Discrete and Combinatorial Mathematics; An Applied Introduction* - Addison-Wesley, 1989.
- GRUBBSTROM. R.W. & J. Lundquist - Completion Times in Networks - *Kybernetes*. Vol 16, No. 3, pp 155-159. 1987.
- GRUVER. W.A. - Evaluation of Commercially Available Robot Programming Languages - *Proceedings XIII ISIR*, pp 1258-1268, 1983.
- HALLORAN. I. - *Dynamic Error Recovery: Transfer Report* - Dept. of Electronic Engineering, University of Hull, October 1989.
- HARARY. F. & E. M. Palmer. - *Graphical Enumeration*. - Academic Press, 1973.
- HARHALAKIS. G., C.P. Lin & L. Mark - A knowledge based prototype of a factory level CIM system - *Computer Integrated Manufacturing Systems* - Vol 2, No. 1, Butterworths, 1989.
- HARTLEY. J. - Cost versus Faults: The optimum balance in debugging - *Assembly Automation*, Vol 6, No. 1, pp40-42, February 1986.
- HEGINBOTHAM. W.B., M. Dooner & K. Case - Robot Application Simulation - *The Industrial Robot* - June 1979.
- HEWITT. J.A. & R.J. Frank - *Software Engineering in Modula-2: An object-oriented approach*. - MacMillan, 1989.
- HOCUS Manual - PE. Information Systems, Egham, UK.
- HORROCKS. R. - 1990's Manufacturing Systems - *Modelling, Simulation and Control of Discrete Event Systems* - IEE Computing and Control Colloquium - Plymouth Polytechnic, December 1989.
- HOWARD. R.A. - *Dynamic Probabilistic Systems - Volume 1; Markov Models* - John Wiley, 1971.

- HUGGINS. W.H. - Signal Flow Graphs and Random Signals. - *Proc. IRE*, vol 9, pp 74-86, 1957.
- HSU. H. T. - An Algorithm for Finding a Minimal Equivalent Graph of a Digraph. - *Journal of the Association for Computing Machinery*, Vol 22, no. 1, pp 11-16, January 1975.
- IBM - *IBM Robot System/1, AML Reference Manual* - IBM Corporation, 1981
- INGALSBE. L. - *LOTUS 123 with Version 2.0 for the IBM PC*. - Merrill, 1987.
- IOSIFESCU. M. - *Finite Markov Processes and their applications* - Wiley, 1980
- JETER. M.W., W.C. PYE. & C.E. Robinson - An Alternative Way of Computing Matrix Inverses. - *Mathematics & Computing Education (USA)*, vol 21(3), pp182-6, 1987.
- JOHNSON. D.G. - *Integrated Sensors & Actuators for Robotic Assembly*. - University of Hull, PhD. Thesis, 1986.
- JENSEN & Partners - *JPI Modula-2 Owners Handbook* - J & P International, 1987.
- KAMEL. M.S. - Planning and Sensing Tradeoffs in Robotics - *NATO Advanced Research Workshop* - Springer Verlag, October 1988.
- KARKKAINEN. P., T. Heikkila & U. Niemela. - Supplementing a Standard Assembly Robot by Multisensor Capabilities. - *Components, Instruments & Techniques for low cost Automation*. - IFAC Symposium. pp 95-99, 1988.
- KATZAN. H. - *APL Programming and Computer Techniques* - Van Nostrand Reinhold, 1970.
- KELLY. F.P - *Reversibility and Stochastic Networks* - Wiley 1979.
- KEMENY. J.G. & J.L Snell - *Finite Markov Chains*. - Van Nostrand, 1965.
- KEMP. D.R., P.M. Taylor & G.E. Taylor - A sensory gripper for handling textiles - *Robot Grippers* - pp 155-164, Ed. D.T. Pham & W.B. Heginbotham - IFS 1986.
- KODRES. U.R. - Discrete Systems and Flowcharts - *Trans. on Software Engineering* - IEEE vol SE-4, No. 6, pp521-525, November 1978.
- KOENIGSBERG. E. - Cyclic Queues - *Operations Research Quarterly*. Vol 9, No. 1, 1958.
- KUMPEL. D.M. & R.G. Rosa - Automatic Task Generator with Incomplete Information for a Robot Endowed with Sensors. - *Proc. IECOM 87: Int. Conf. Industrial Electronics*, IEEE 1987.
- LALONDE. W.R., D.A. Thomas & K. Johnson - Smalltalk as a Programming Language for Robotics? - *Proc. IEEE Conf. Robotics & Automation*, pp1456-1462, March 1987.
- LE BEUX. P. - *Lexique Micro-informatique* - Sybex, Paris 1984.
- LEE. A.M. - *Applied Queueing Theory* - Studies in Management - MacMillan 1966.
- LEE. C.N., M.Y. Chiu, P. Liu & S.J. Clark - Model-based Hierarchical Diagnosis of Robotic Assembly Cell. - *Intelligent Robots and Computer Vision*, SPIE Vol 848, pp182-189, 1987.
- LEE. M.H., N.W. Hardy & D.P. Barnes. - Research into Automatic Error Recovery. - *Proc. Conf. on UK Robotics Research*. - Inst. Mech. Engineers, London. Dec. 1984.

- LEVI. P & T. Loeffler - The use of Assembly Graphs for Robot Programming. - *Languages for Sensor-based Control in Robotics - Proceedings of the NATO International Advanced Workshop on Languages for Sensor-based Control*, Vol 7, pp233-259, September 1986.
- LIPSCHUTZ. S. - *Finite Mathematics* - Schaum's Outline Series, Mc. Graw-Hill, 1966.
- LOGITECH - *M 2 VMS LOGITECH MODULA-2 Users Manual* - Logitech S.A., Switzerland, 1988.
- LYONS. D.M. - A Novel Approach to High Level Robot Programming. - *IEEE Workshop on Languages for Automation*, pp48-51, 1987.
- MALCOLM. C.A. & A.P. Fothergill - Some Architectural Implications of the use of Sensors. - *NATO ASI series*. Vol F29 - Springer Verlag 1987.
- MARKOWITZ. H.M. - *SIMSCRIPT: Past, present and some thoughts about the future in current issues in computer simulation*. - N. Adam & A. Dogramaci (Eds.). - Academic Press, NewYork 1979.
- MARTIN. J.J. - Distribution of the Time through a Directed, Acyclic Network. - *Operations Research*, Vol 13, No. 1, pp46-66, January 1965.
- MASON. S.J. - Feedback Theory - Further Properties of Signal Flowgraphs - *Proc. Transactions on Circuit Theory*. - IRE, July 1956.
- MASON. T.M. & J.K. Salisbury, Jr. - *Robot Hands and the Mechanics of Manipulation* - MIT Press, 1985.
- MAYER. S.L. - *Data Analysis for Scientists and Engineers*. - Wiley, 1975.
- MELLOR. P.V. - *An Adaptation of Modula-2 for Distributed Computing Systems*. - Ph.D Thesis, University of Hull, 1987.
- MERLIN. P.M. - *A Study of the Recoverability of Computer Systems* - Ph.D Thesis, University of California, Irvine 1974.
- MICROSOFT - *MS-DOS v3.2 Disc Operating System Manual* - Microsoft Corporation, 1986.
- MILOVANOVIC. R. - Towards Sensor-based General Purpose Robot Programming Language. - *Robotica*, vol 5 1987, pp 309-316.
- MOLER.C., J. Little, S. Bangert & S. Kleiman - *PC-MATLAB for MS-DOS Personal Computers* - MathWorks Inc., Masseurhusses, 1986.
- MOLLOY. M.K. - Discrete time Stochastic Petri nets - *Trans. on Software Engineering* - IEEE vol se-11, No. 4, April 1985.
- MONKMAN. G.J. - *Electrostatic Techniques for Fabric Handling* - MSc. Thesis, University of Hull. 1987.
- MONKMAN. G.J. - *Sensor Transition and Object Driven Programming*. - Internal Report No. 84/89, University of Hull, December 1989.
- MONKMAN. G.J., G.E. Taylor & P.M. Taylor - Flowgraph Techniques in Workcell Assesment and Design - *International Symposium on Intelligent Control* - IEEE, Albany, NY, September 1989.
- MONRO. D.M. - *Fortran 77* - Edward Arnold, 1983.

- NELSON. R.A., L.M. Haibt & P.B. Sheridan - Casting Petri nets into programs - *Trans. on Software Engineering*. - IEEE vol SE-9, No. 5, pp590-602 September 1983.
- NILSSON. N.J. - *A Hierarchical Robot Planning and Execution System* - SRJ Artificial Intelligence Centre, Technical note 76, April 1973.
- OPEN UNIVERSITY - Simulation II - *M351 Mathematics* - M351 units 14 & 15, Open University 1982.
- PETERSON. J.L. - *Petri net Theory and the Modelling of Systems* - Prentice Hall, 1981.
- POPPLEWELL. K. & H. Jaio - Simulation, Object-oriented Programming, and System Description Methodology: A Progress Report - *Modelling, Simulation and Control of Discrete Event Systems* - IEE Computing and Control Colloquium - Plymouth Polytechnic, December 1989.
- PRITSKER. A.A.B. - GERT: *Graphical Evaluation and Review Technique* - NASA memo: RM-4973 Rand Corp. 1966.
- PRITSKER. A.A.B. - *User's Manual for GERT Simulation Program* - NASA/ERC NGR 03-001-034, Arizona State University, July 1968.
- REISIG. W. - *Petri Nets: An Introduction* - Springer Verlag, 1985.
- RICE. J.R. - *Matrix Computations and Mathematical Software*. - Mc.Graw-Hill, 1981.
- ROBINSON. J. - *Petri Net Software* - Rensselaer Polytechnic Inst. 22 February 1989.
- RODIGHIERO. F. & A. Canciani - An Experience in Task Level Robot Programming - *Workshop on Languages for Automation*, pp 86-89, IEEE, 1987.
- ROMANOVSKY. V.I. - *Discrete Markov Chains*. - Wolters-Nordhoff, 1970.
- SANDERS. J. - Training no barrier - *Parallelogram* - Issue 22, p6, January 1990.
- SCHRUBEN. L.W. - Simulation Modelling with Event Graphs. - *Communications of the ACM*, vol 26, No. 11, November 1983.
- SERC Final Report. - *Automation of Shirt Collar Inspection and Assembly*. - University of Hull. 1988.
- SHAN. Y.P. - An Event-driven Model-view-controller Framework for Smalltalk - *OOPSLA'89 Proc.* pp 347-352 - ACM, 1989.
- SIEGRIST. K. - Reliability of Systems with Markov Transfer of Control - *Trans. on Software Engineering* - IEEE vol 14, No. 7, pp1049-1053, July 1988.
- SINGH. M.G. & K.S. Hindi - An Overview of Research on discrete event dynamical systems. - *Modelling, Simulation and Control of Discrete Event Systems* - IEE Computing and Control Colloquium - Plymouth Polytechnic, December 1989.
- SONG. X.K. - *A Prototype Expert System Application for Run-time Error-recovery in a Robotic Assembly Workcell* - Internal Report No. 52/88 - University of Hull, August 1988.
- SPIEGEL. M.R. - *Mathematical Handbook of Formulas and Tables* - p 107 (19.4) - Mc.Graw Hill 1968.
- SRINIVAS. S. - *Error Recovery in Robot Systems* - Ph.D Thesis - California Ints. of Technology, Information Science, December, 1976.

- SRINIVAS. S. - Error Recovery in Robots through Failure Reason Analysis - *Proc. of International Computer Conf.*, Anaheim CA - AFIP, June 1978.
- STEFIK. M. & D.G. Bobrow - Object-Oriented Programming: Themes and Variations - *AI Magazine*, Vol 6, No. 4, pp40-62, Winter 1986.
- TAHA. H.A. - *Operations Research, An Introduction*. - 4<sup>th</sup> Ed., MacMillan, 1987.
- TAYLOR. P.M. - Multisensory Assembly and Error Recovery. - *NATO Workshop*, Castlevecchio, Italy. Oct. 1987.
- TAYLOR. P.M., A.J. Wilkinson, G.E. Taylor, M.B. Gunner & G.S. Palmer - Automated Fabric Handling Problems and Techniques - *IEEE Systems Engineering Conf.* - Pittsburgh Pa., August 1990.
- TORN. A.A. - Simulation Graphs: A General Tool for Modelling Simulation Designs - *Simulation* - pp187-194, December 1981.
- TUTTE. W. T. - *Connectivity in Graphs*. - University of Toronto Press, 1966.
- UNIMATION - *PUMA 500 mk II User's Guide to VAL II. 398T1, Version 1.4B*. - Unimation, May 1985.
- VAL II - *Version 1.0, User instruction Manual* - Adept Technology inc. September 1984.
- VALAVANIS. K.P. - *On the Hierarchical Modelling Analysis and Simulation of Flexible Manufacturing Systems with Extended Petri nets* - Internal Report, Northeastern University, 1989.
- VALETTE R., J. Cardoso & D. Dubois - Monitoring Manufacturing Systems by means of Petri nets with imprecise markings - *Proc. International Symposium on Intelligent Control* - IEEE, September 1989.
- WEISSENBORN. H.O. - Technology Tomorrow - *Proc. 5<sup>th</sup> Intl. conf. in Simulation in Manufacturing* - Ed. Dr. Ing. H. Baumgarten - pp 185-196, 13/14 June 1989.
- WERUM. W. & H. Windaur - *Introduction to PEARL, Process and Experiment Automation Realtime Language* - Friedr. Vieweg & Sohn, 1982.
- WILLIAMS. D.J., P. Rogers & D.M. Upton - Programming and Recovery in Cells for Factory Automation - *The International Journal of Advanced Manufacturing Technology* - IFS Pubs., 1986
- WILSON. R.J. - *Introduction to Graph Theory* - Longman (2<sup>nd</sup> Ed) 1979.
- WHITEHOUSE. G. - *Model Systems on Paper with Flowgraph Analysis*. - *Industrial Engineering*, pp30-35, June 1969.
- WHITEHOUSE. G.E. - *System Analysis and Design using Network Techniques*. - Prentice-Hall, 1973.
- WOODWARD. P.M., P.R. Wetherall & B. Gorman - *Official Definition of CORAL 66* - Ministry of Defence, HM Stationary Office, 1970.
- ZWARICO. A. - Robot Programming Languages: Issues of Concurrency and Real-Time. - *Proc. International Conf. on Systems, Man & Cybernetics* - IEEE, November 1985.

APPENDIX



## APPENDIX A

### Queueing Theory Nomenclature.

The following is a brief list of the nomenclature commonly used in queueing theory. The information was obtained from a number of texts, particularly Lee [Lee, 1966].

Queues are usually denoted by a type of code which contains information on various aspects of the queue involved. For example M/M/2:(12/LIFO) etc.

So, given the syntax A/B/C:(d/e) where:

- A - Arrival pattern.
- B - Service time distribution.
- C - Number of servers.
- d - Maximum number of customers in queue (including the one being served).
- e - Queue discipline.

#### Examples:

For example :- M/M/1:(20/FIFO)

means :- Random Arrival/Random Service/One Server:  
Max queue length of 20/FIFO Queue)

and :- M/E<sub>k</sub>/5:(∞/SIRO)

means :- Random Arrival/Erlangian Distribution/5 Channels:  
(Infinite Queue Length/Service In Random Order)

#### Distributions:

- M - Random distribution.
- D - Constant distribution.
- E<sub>k</sub> - Erlang distribution (distribution of the sum of k independantly and identically distributed negative exponential variables).
- G - General distribution.
- GI - Independant general distribution.

## Queue Disciplines:

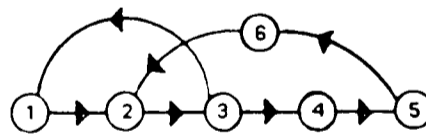
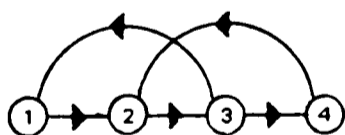
- FIFO - First In First Out.
- SIRO - Service In Random Order.
- LIFO - Last In First Out.
- PSPO - Pre-emptive Service Priority Order.
- NPPS - Non Pre-emptive Priority Service.

## APPENDIX B

### Digraph Nomenclature.

The following notation, taken from Wilson [Wilson, 1979], appears in most texts discussing digraph theory, particularly when taken from a mathematical view point.

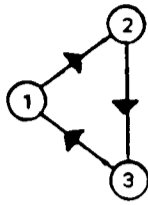
<b>ABSORBING</b>	A node from which it is impossible to get to any other state, ie. a sink.
<b>ADJACENT</b>	Two vertices, U & V, are said to be ADJACENT if there exists paths UV or VU.
<b>ARCS</b>	Pairs of elements (paths).
<b>BIPARTITE</b>	If each node were coloured (say, red and blue) and each path has both a red and a blue end.
<b>DIGRAPH</b>	Directed graph.
<b>ERGODIC</b>	Both PERSISTENT and APERIODIC.
<b>HOMEOMORPHIC</b>	Identical to within vertices of degree two. ie., (a) and (b) are HOMEOMORPHIC. (HOMEOMORPHIC $\equiv$ EQUIVALENT)



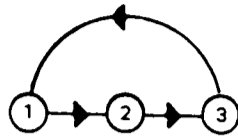
<b>INCIDENT</b>	A vertex U is INCIDENT to an ARC UV, VU, WU etc.
<b>IRREDUCIBLE</b>	A transition matrix is irreducible if its digraph is strongly connected.
<b>IRREFLEXIVE</b>	A matrix A having only zeros in the main diagonal.

**ISOMORPHIC**

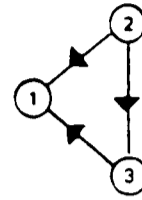
Two graphs having the same flow into each element. ie., (a) and (b) below are ISOMORPHIC, but (a) and (c) are not.



(a)



(b)



(c)

**PERSISTENT**

A node whose transition is sooner or later inevitable (ie. Probability of transition = 1).

**PERIODIC**

A node to which it is only possible to return to after some multiple of time, t.

**REFLEXIVE**

A digraph having a loop incident with each vertex.

**STRONGLY CONNECTED**

Contains a direct path from U to V and back. Each node is mutually reachable from every other node.

**TRANSITION MATRIX**

Probability matrix (each row of a TRANSITION MATRIX is a probability vector).

**VERTICES**

Elements (nodes).

## APPENDIX C

### Markov Chain Nomenclature.

ABSORBING	A chain, all of whose non-transient states are absorbing (trapping), is called an absorbing chain.
DIFFERENTIAL MATRIX	Represents multinomial processes - all rows and columns sum to zero.
DOUBLY STOCHASTIC	Both columns and rows of the transition matrix sum to zero. The limiting matrix = $nI$ , ie., the solutions to the transition matrix are all equal to $1/n$ .
DUODESMIC	A process containing two chains.
ERGODIC	A Markov chain which is both persistent and aperiodic.
MARKOV CHAIN	A discrete Markov process.
MARKOV PROCESS	A process in which each state is independent of the last state.
MONODESMIC	A process which has only one solution to the limiting matrix - usually represents a strongly connected digraph.
MULTINOMIAL PROCESS	The probability of transition to each state is independent of the state occupied.
POLYDESMIC	A process containing two or more chains.
REGULAR MARKOV CHAIN	Has no transient sets and contains a single ergodic set with only one cyclic class.
STOCHASTIC MATRIX	A matrix whose elements all lie within the range $[0,1]$ and whose rows sum to 1.
TRANSIENT STATE	A state which has zero probability of being occupied after a large number of transitions.

## APPENDIX D

### Statistical Distributions.

The following statistical distributions are a sample of the commonest ones used in queueing theory, simulation etc. This is not intended as a full study of probability distributions, but rather an overview for the sake of easy reference when reading the main text. A number of publications have been consulted in the compilation of this appendix and these will be referred to where appropriate.

#### BINOMIAL.

PROB( $r$  successes in  $n$  trials) =  ${}^n C_r p^r q^{n-r}$ ,  $r = 0, 1, 2, \dots, n$

where:  ${}^n C_r = \frac{n!}{r!(n-r)!}$

#### POISSON.

Given an average occurrence rate  $\lambda$  units/second.

PROB( $r$  events in a given interval) =  $\frac{\lambda^r e^{-\lambda}}{r!}$

#### NORMAL (GAUSSIAN).

$P(x)$  has a normal distribution over  $x \in [a, b]$

$$\text{iff } \int_a^b e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx = \frac{1}{\sqrt{2\pi}} \int \frac{(x-\mu)/\sigma}{e^{-z^2/2}} dz$$

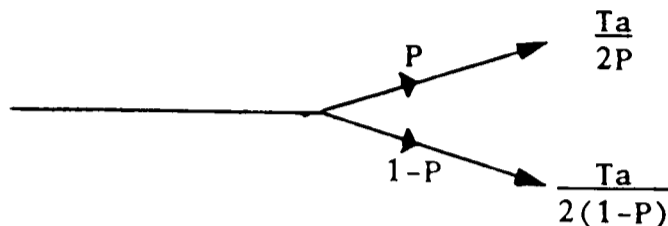
where  $x$  is normally distributed with mean  $\mu$  and standard deviation  $\sigma$ .

$P(z)$  found from tables [Dowdy & Wearden, 1983].

## HYPER-EXPONENTIAL DISTRIBUTION.

Describes a distribution where the standard deviation is larger than the mean, for example low and high values occur more frequently. The data may be bimodal [Gordon, 1969].

This can be modelled as a branch process.



if the distributions  $Ta$  are exponential, then:

$$A_0(t) = P e^{-2P\lambda t} + (1-P) e^{-2(1-P)\lambda t} ; \quad P \in ]0, 0.05]$$

## MULTIMODAL DISTRIBUTIONS.

Given a continuous function  $f(X)$ , then the mode is defined as the abscissa of a local maximum such that:

$$\left. \frac{df(X)}{dX} \right|_{X=X_{\text{mode}}} = 0 \quad \text{and} \quad \left. \frac{d^2f(X)}{dX^2} \right|_{X=X_{\text{mode}}} < 0$$

[Mayer, 1975]

Many tests of sample variance exist for normal population distributions, such as  $t$  and  $\chi^2$  distributions [Bajpai et al., 1979].

However, when dealing with more complex distributions, such as bimodal, more complex techniques must be employed. For example, the use of quantile and density quantile functions which treat the normal distribution simply as one of many [Darzen, 1982].

## APPENDIX E

### Modula 2 Program Routines.

#### ANAL Procedures.

Each of the procedures of the Modula 2 programs mentioned in the main text are described below. These individual descriptions are followed by annotated Modula 2 program listings for the complete package.

All matrices are square and of dimension  $n$  unless specified otherwise.

`matread(a)`                       $a \leftarrow \text{input}$

Reads an  $n$  by  $n$  matrix of reals from the keyboard into variable  $a$ .

`matwrite(a)`                       $a \rightarrow \text{output}$

Writes an  $n$  by  $n$  matrix of reals from variable  $a$  onto the screen.

`I(a,n)`                             $a \leftarrow I(n,n)$

Puts an  $n$  by  $n$  identity matrix into variable  $a$ .

`M(a,n)`                             $a \leftarrow M(n,n)$

Puts an  $m$  by  $m$  mirror matrix into variable  $a$ .

`transp(a)`                         $b \leftarrow a^T$

Finds the transpose of matrix  $a$  returning the result to matrix  $b$  leaving matrix  $a$  unchanged.

`matadd(a,b,c)`                     $c \leftarrow a + b$

Adds matrix  $a$  to matrix  $b$  leaving the result in  $c$ .

`matsub(a,b,c)`                     $c \leftarrow a - b$

Subtracts matrix  $b$  from matrix  $a$  leaving result in  $c$ .



matconmul(a,b,c)  $c \leftarrow a \square b$

Performs congruent multiplications (element by element multiplication) between matrix a and matrix b leaving the result in c.

matmul(a,b,c)  $c \leftarrow a * b$

Performs standard matrix multiplication between matrix a and matrix b leaving the result in c.

matinv(a,b)  $b \leftarrow a^{-1}$

Inverts matrix a leaving result in b.

diag(a,b)  $b \leftarrow \text{DIAG}(a)$

Takes the main diagonal of matrix a and places along the main diagonal of matrix b leaving all other elements of matrix b zero.

matflow(a,b)  $b \leftarrow [I - a]^{-1}$

Finds the characteristic matrix using a and places the result in b.

matvar(a,b)  $b \leftarrow a * (2 * \text{DIAG}(a) - I) - a \square a$

Finds the variance of matrix a and places the result in b.

scalmul(x,a,b)  $b \leftarrow x * a$

Performs scalar multiplication between scalar x and matrix a leaving the result in b

matlim(a,b)  $b \leftarrow a^{\infty}$

Finds the limiting matrix of matrix a leaving the result in b.

## Modula-2 ANAL Simulator Listing.

```
MODULE anal;
(***) This is a matrix manipulation package for the stochastic simulation
of weighted flowgraph networks. All matrices are assumed square and
of dimension n, as determined during the initial matrix entry (***)

    FROM IO IMPORT RdCard, RdReal, WrLn, WrReal, WrStr;

    VAR x: REAL;
    VAR n: CARDINAL;
    TYPE matrix = ARRAY[1..32] OF ARRAY[1..32] OF REAL;
    VAR A,B,C,D: matrix;

PROCEDURE matread(VAR a: matrix);
(***) Reads in a square matrix a of dimension n (***)
VAR i,j: CARDINAL;
BEGIN
    WrStr('Enter matrix size = ');
    n:=RdCard();
    WrStr('Enter matrix elements');
    WrLn;
    FOR i:=1 TO n DO
        FOR j:=1 TO n DO
            a[i,j]:=RdReal();
        END;
    END;
END matread;

PROCEDURE matwrite(VAR a: matrix);
(***) Writes out a square matrix a of dimension n (***)
VAR i,j: CARDINAL;
BEGIN
    FOR i:=1 TO n DO
        FOR j:=1 TO n DO
            WrReal(a[i,j],3,3);
        END;
        WrLn;
    END;
END matwrite;

PROCEDURE I(VAR a: matrix;
            VAR n: CARDINAL);
(***) Puts the identity matrix I into matrix a (***)
VAR i,j: CARDINAL;
BEGIN
    FOR i:=1 TO n DO
        FOR j:=1 TO n DO
            a[i,j]:=-0.0;
            a[i,i]:=-1.0;
        END;
    END;
END I;
```

```

PROCEDURE M(VAR a: matrix;
            VAR n: CARDINAL);
(***) Puts the mirror (rotation) matrix M into matrix a (***)
VAR i,j: CARDINAL;
BEGIN
    FOR i:=1 TO n DO
        FOR j:=1 TO n DO
            a[i,j]:=0.0;
            a[i,n+1-i]:=-1.0;
        END;
    END;
END M;

```

```

PROCEDURE transp(VAR a,b: matrix;
                VAR n: CARDINAL);
(***) Transposes the matrix a returning the result to b (***)
VAR i,j: CARDINAL;
BEGIN
    FOR i:=1 TO n DO
        FOR j:=1 TO n DO
            b[j,i]:=a[i,j];
        END;
    END;
END transp;

```

```

PROCEDURE matadd(VAR a,b,c: matrix);
(***) Adds the matrices a and b leaving the result in c (***)
VAR i,j: CARDINAL;
BEGIN
    FOR i:=1 TO n DO
        FOR j:=1 TO n DO
            c[i,j]:=a[i,j]+b[i,j];
        END;
    END;
END matadd;

```

```

PROCEDURE matsub(VAR a,b,c: matrix);
(***) Subtracts the matrix b from a leaving the result in c (***)
VAR i,j: CARDINAL;
BEGIN
    FOR i:=1 TO n DO
        FOR j:=1 TO n DO
            c[i,j]:=-a[i,j]-b[i,j];
        END;
    END;
END matsub;

```

```

PROCEDURE matsum(VAR x: REAL;
                 a: matrix);
(***) Puts the sum of all the elements of matrix a into variable x (***)
VAR i,j: CARDINAL;
BEGIN
    x:=0.0;
    FOR i:=1 TO n DO
        FOR j:=1 TO n DO
            x:=x+a[i,j];
        END;
    END;
END matsum;

```

```

PROCEDURE matconmul(VAR a,b,c: matrix);
(***) Performs congruent multiplication between matrix a and matrix b
    leaving the results in matrix c (***)
VAR i,j: CARDINAL;
BEGIN
    FOR i:=1 TO n DO
        FOR j:=1 TO n DO
            c[i,j]:=-a[i,j]*b[i,j];
        END;
    END;
END matconmul;

```

```

PROCEDURE matmul(VAR a,b,c: matrix);
(***) Performs matrix multiplication between a and b with result in c (***)
VAR i,j,k: CARDINAL;
BEGIN
    FOR i:=1 TO n DO
        FOR j:=1 TO n DO
            c[i,j]:=0.0;
            FOR k:=1 TO n DO
                c[i,j]:=-c[i,j]+a[i,k]*b[k,j];
            END;
        END;
    END;
END matmul;

```

```

PROCEDURE scalmul(VAR x: REAL;
                 VAR a,b: matrix);
(***) multiplies matrix a by scaler x putting result into b (***)
VAR i,j: CARDINAL;
BEGIN
    FOR i:=1 TO n DO
        FOR j:=1 TO n DO
            b[i,j]:=-x*a[i,j];
        END;
    END;
END scalmul;

```

```

PROCEDURE matinv(VAR a,b: matrix);
(***) Inverts matrix a putting inverted matrix result into matrix b (***)
VAR i,j,k: CARDINAL;
    z: REAL;
BEGIN
    FOR i:=1 TO n DO
        FOR j:=1 TO n DO
            b[i,j]:=0.0;
        END;
        b[i,i]:=1.0;
    END;
    FOR k:=1 TO n DO
        FOR i:=1 TO n DO
            IF i#k THEN
                z:=a[i,k]/a[k,k];
                FOR j:=1 TO n DO
                    a[i,j]:=a[i,j]-a[k,j]*z;
                    b[i,j]:=b[i,j]-b[k,j]*z;
                END;
            END; (***)ENDIF(***)
        END;
        z:=a[k,k];
        FOR j:=1 TO n DO
            a[k,j]:=a[k,j]/z;
            b[k,j]:=b[k,j]/z;
        END;
    END;
END matinv;

```

```

PROCEDURE diag(VAR a,b: matrix);
(***) Puts main diagonal elements of matrix a into matrix b diagonal (***)
VAR c: matrix;
BEGIN
    I(c,n);
    matconmul(a,c,b);
END diag;

```

```

PROCEDURE matflow(VAR a,b: matrix);
(***) Finds the characteristic (flow) matrix b=inv[1-a] (***)
VAR c: matrix;
BEGIN
    I(b,n);
    matsub(b,a,c);
    matinv(c,b);
END matflow;

```

```

PROCEDURE matvar(VAR a,b: matrix);
(***) Finds variance of matrix a returning result to matrix b (***)
VAR c,d: matrix;
    x,z: REAL;
BEGIN
    diag(a,b);
    x:=-2.0;
    scalmul(x,b,c);
    I(b,n);
    matsub(c,b,d);
    matmul(a,d,c);
    matconmul(a,a,d);
    matsub(c,d,b);
END matvar;

```

```

PROCEDURE matlim(VAR a,b: matrix);
(***) Finds the limiting matrix of a returning result to matrix b (***)
VAR c,d: matrix;
    x: REAL;
BEGIN
    x:=-1.0;
    b:=-a;
    REPEAT
        matmul(b,b,c);
        matmul(c,c,b);
        matsub(b,c,d);
        matsum(x,d);
    UNTIL ABS(x)<0.00001;
END matlim;

```

```

(***) Example Main program for execution of matrix procedures (***)
BEGIN
  WrStr('Enter stochastic transition matrix');
  WrLn;
  matread(A);
  matlim(A,B);
  WrStr('LIMITING matrix is:');
  WrLn;
  matwrite(B);
  WrLn;
  matflow(A,B);
  WrStr('FLOW matrix is:');
  WrLn;
  matwrite(B);
  WrLn;
  matvar(B,C);
  WrStr('VARIANCE matrix is:');
  WrLn;
  matwrite(C);
  WrLn;
  WrStr('Enter toll matrix');
  WrLn;
  matread(A);
  matconmul(A,B,C);
  WrStr('Toll AVERAGE matrix is:');
  WrLn;
  matwrite(C);
  WrLn;
  matsum(x,C);
  WrStr('MEAN toll value is:');
  WrReal(x,6,6);
END anal.

```

## DRIVE Procedures.

`WrString('a')`  $a \rightarrow$  output

Writes the string between quotes ' ' to the screen (or other selected output device).

`RdStr(a)`  $a \leftarrow$  input

Reads a string from the keyboard (or other selected input device).

`matstread(a)`  $a \leftarrow$  input

Reads a matrix of string values from the keyboard (or other selected input device).

`matread(a)`  $a \leftarrow$  input

Reads a matrix of cardinal elements from the keyboard (or other selected input device).

`vecread(a)`  $a \leftarrow$  input

Reads a vector of cardinal values from the keyboard (or other selected input device).

`vecmatmul(a,b,c)`  $a \leftarrow b * c$

Multiplies vector b by matrix c leaving the result in vector a. Vector b and matrix c are left unchanged.

`execute(a)`  $a \rightarrow$  output

Executes a Markov process. The string a is output to the screen or other selected output device (usually the robot controller). The string a will normally be a program routine recognisable by the host controller.



## Modula-2 DRIVE Programming Listings.

```
MODULE drive;
(** Robot control module. A transition matrix of VAL II procedures is
    entered together with an initial sensor state matrix and starting
    vector. A Markov process is executed producing VAL II commands to
    drive the robot. A new driving vector is produced and the sensor
    state matrix updated after each action. ***)

FROM IO IMPORT RdCard, RdChar, WrStr, WrChar, WrCard, WrLn;

VAR n          : CARDINAL;

TYPE vector = ARRAY[1..32] OF CARDINAL;
    matrix = ARRAY[1..32] OF ARRAY[1..32] OF CARDINAL;
    string = ARRAY[1..10] OF CHAR;
    stringmatrix = ARRAY[1..32] OF ARRAY[1..32] OF string;

VAR x,y        : string;
    R          : stringmatrix;
    S          : matrix;
    vector0,vector1 : vector;

PROCEDURE WrString(VAR x:string);
(** Writes a string variable (NOTE: WrStr does not work with string
    variables! - only characters between quotes, ie. WrStr('abc') ***)
VAR k: CARDINAL;
BEGIN
    k:=0;
    REPEAT
        INC(k);
        WrChar(x[k]);
    UNTIL x[k]<=" ";
END WrString;

PROCEDURE RdStr(VAR x: string);
(** Reads a string of 10 characters ***)
VAR k,l: CARDINAL;
BEGIN
    REPEAT
        x[1]:=-RdChar();
    UNTIL x[1]>" ";
    k:=1;
    REPEAT
        INC(k);
        x[k]:=-RdChar();
    UNTIL x[k]<=" ";
END RdStr;
```

```

PROCEDURE matstread(VAR R:stringmatrix);
(***) Reads in a matrix of string variables (***)
VAR i,j: CARDINAL;
BEGIN
    WrStr('Enter matrix size ');
    n:=RdCard();
    WrStr('Enter matrix string elements ');
    WrLn;
    FOR i:-1 TO n DO
        FOR j:-1 TO n DO
            RdStr(R[i,j]);
        END;
    END;
END matstread;

```

```

PROCEDURE matread(VAR S: matrix);
(***) Reads a matrix of cardinal elements (***)
VAR i,j,p: CARDINAL;
BEGIN
    WrStr('Enter');
    WrCard(n,3);
    WrStr(' by');
    WrCard(n,3);
    WrStr(' Sensor matrix (1, 0 values only)');
    WrLn;
    FOR i:-1 TO n DO
        FOR j:-1 TO n DO
            S[i,j]:=-RdCard();
        END;
    END;
END matread;

```

```

PROCEDURE vecread(VAR vector0: vector);
(***) Reads in a process driving vector (***)
VAR i,p: CARDINAL;
BEGIN
    FOR i:-1 TO n DO
        vector0[i]:=-RdCard();
        vector1[i]:=0;
    END;
END vecread;

```

```

PROCEDURE vecmatmul(VAR vector1,vector0: vector;
                    S: matrix);
(***) multiplies a vector by a matrix (***)
VAR i,j: CARDINAL;
BEGIN
    FOR j:-1 TO n DO
        FOR i:-1 TO n DO
            vector1[j]:=-vector1[j] + vector0[i] * S[i,j];
        END;
    END;
END vecmatmul;

```

```

PROCEDURE execute(VAR y: string);
(***) executes a markov process (***)
VAR k,p,q: CARDINAL;
BEGIN
    p:=0;
    q:=0;
    FOR k:=1 TO n DO
        IF vector0[k] >= 1 THEN
            p:=k;
        END; (***) ENDIF (***)
    END;
    vecmatmul(vector1, vector0, S);
    FOR k:=1 TO n DO
        IF vector1[k] >= 1 THEN
            q:=k;
        END; (***) ENDIF (***)
    END;
    IF (p=0) OR (q=0) THEN
        WrStr('Procedure Completed');
        WrLn;
        HALT;
    END; (***) ENDIF (***)
    y:=R[p,q];
    (***) Output VAL II procedure (***)
    WrString(y);
    WrLn;
    matread(S); (***) next sensor state (***)
    vector0:=vector1;
    (***) Re-initialise vector1 (***)
    FOR k:=1 TO n DO
        vector1[k]:=0;
    END;
END execute;

```

```

(***) main program (***)
BEGIN
    WrStr('Enter Routine matrix');
    WrLn;
    matstread(R);
    WrStr('Enter Initial sensor matrix');
    WrLn;
    matread(S);
    WrStr('Enter starting vector');
    WrLn;
    vecread(vector0);
    REPEAT
        execute(y);
    UNTIL vector1=vector0;
END drive.

```

CONTAINS DISKETTE

UNABLE TO COPY

CONTACT UNIVERSITY

IF YOU WISH TO SEE

THIS MATERIAL