Computer Control Of Machines Utilising Independent Drive Mechanisms

CHRISTOPHER MARK DRAPER

Doctor Of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

September 1992

The University Of Aston in Birmingham

Computer Control Of Machines Utilising Independent Drive Mechanisms.

CHRISTOPHER MARK DRAPER

Doctor Of Philosophy

1992

Summary

The thesis describes an investigation into methods for the specification, design and implementation of computer control systems for flexible manufacturing machines comprising multiple, independent, electromechanically-driven mechanisms. An analysis is made of the elements of conventional mechanically-coupled machines in order that the operational functions of these elements may be identified. This analysis is used to define the scope of requirements necessary to specify the format, function and operation of a flexible, independently driven mechanism machine. A discussion of how this type of machine can accommodate modern manufacturing needs of high-speed and flexibility is presented.

A sequential method of capturing requirements for such machines is detailed based on a hierarchical partitioning of machine requirements from product to independent drive mechanism. A classification of mechanisms using notations, including Data flow diagrams and Petri-nets, is described which supports capture and allows validation of requirements. A generic design for a modular, IDM machine controller is derived based upon a hierarchy of control identified in these machines.

A two mechanism experimental machine is detailed which is used to demonstrate the application of the specification, design and implementation techniques. A computer controller prototype and a fully flexible implementation for the IDM machine, based on Petri-net models described using the concurrent programming language Occam, is detailed. The ability of this modular computer controller to support flexible, safe and fault-tolerant operation of the two intermittent motion, discrete-synchronisation independent drive mechanisms is presented. The application of the machine development methodology to industrial projects is established.

Key library index words:
DISTRIBUTED COMPUTER CONTROL, SPECIFICATION, VALIDATION, DATA FLOW DIAGRAM, PETRI-NET.

To Helen, Nan and Tony

Acknowledgements

I would like to express my thanks to the many members of staff at Aston University and Molins PLC who gave me guidance and encouragement in this research project.

In particular I would like to thank Doctor D.J.Holding for arranging and supervising my research project and for continuing with my supervision after I moved into industry.

Christopher Draper
September 1992

Table of contents

List of figures

List of Tables

List of abbreviations

AC	Alternating current
ACP	Activity, channel, pool
ADC	Analog to digital convertor
CAD	Computer aided design
CAM	Computer aided manufacture
CASE	Computer aided software engineering
CFD	Control flow diagram
CNC	Computer numerical control
CORE	Controlled requirements specification
CRT	Cathode ray tube
DAC	Digital to analog convertor
DC	Direct current
DFD	Data flow diagram
DNC	Direct numerical control
ec	Encoder counts
FMS	Flexible manufacturing system
FSM	Finite state machine
HUI	Human user interface
IDA	Intercommunication data areas
IDM	Independent drive mechanism
IO	Input / output
IPSE	Integrated programming support environment
JSD	Jackson system design
MAP	Manufacturing automation protocol
MASCOT	Modular approach to software code operation and test
MIS	Management information system
MIMD	Multiple instruction, multiple data
PID	Proportional, integral, differential
PLC	Programmable logic controller
RTOS	Real time operating system
SAFRA	Semi-automated functional requirements analysis
SD	Structure diagram
SIMD	Single instruction, multiple data
SSD	System specification diagram
TDS	Transputer development system

Chapter 1

Introduction

1.0 General

This thesis details research undertaken to provide generic methods for the specification, design and implementation of computer control systems for manufacturing machines constructed using multiple independently driven mechanisms. This type of machine is an innovative design which facilitates high-speed operation whilst allowing flexibility in the component manufacturing operations to produce a group of related products.

The characteristics of this approach to machine design are the use of multiple independent servo-driven mechanisms which replace the prime mover and drive train of a conventional machine; the grouping of independently driven mechanisms into mechanical modules which perform specific manufacturing tasks; and computer control of the set of mechanism actuators incorporating software which provides flexibility in motion control and synchronisation of these actuators.

The techniques used in the construction of the mechanics, power electronics, computer hardware and software of these independent drive mechanism (IDM) machines are still under development. For the computer control system in particular methods of specification, design and implementation which can provide safe and reliable operation whilst supporting the IDM machine requirements of modularity, speed and flexibility need to be identified.

The research work was sponsored by the Science and Engineering Research Council (SERC) under it's Specially Promoted Program (SPP) on high speed machinery. An industrial sponsor, Molins PLC, has posed pertinent test cases and technical problems which have guided the work to address relevant development issues. The work forms part of a programme of research at Aston University into high-speed machine design.

1.1 High speed machines

In this section the nature of high speed machines and conventional methods of construction are examined. The evolution of conventional designs to accommodate changing customer requirements is discussed together with the advantages and disadvantages of conventional machine designs. New driving-points in machine development are introduced which have forced a change in the approach to machine design.

1.1.1 Nature and Characteristics of high-speed machines

Modern manufacturing machines have been subdivided into three categories by Jarvis [1]. The division has been made by indexing number of parts produced a year against number of different parts that can be produced. For low volume production many different types of product can be made by a reconfigurable or programmable machine. For medium volume production a limited number of products may be made by one machine by changing components in the machine. For high volume production a dedicated machine manufactures a single product. A graphical representation of this division can be seen in figure 1.1.

A typical high speed machine is required to manufacture product at a rate of 1000 units a minute. These machines are dedicated to manufacturing a single product, mechanical modifications to the machines may allow a limited number of parameters of this product to be changed. This places conventional high speed machines in the fixed automation partition of the graph in figure 1.1.

Capital costs of the machines are high due to the high cost in developing the machines and the advanced technology and materials used in their construction. Reliability of the machines is important due to cost penalties incurred as a result of lost production due to faults. Efficiency in the use of material is also significant due to the large volumes of product manufactured and associated raw material consumed.

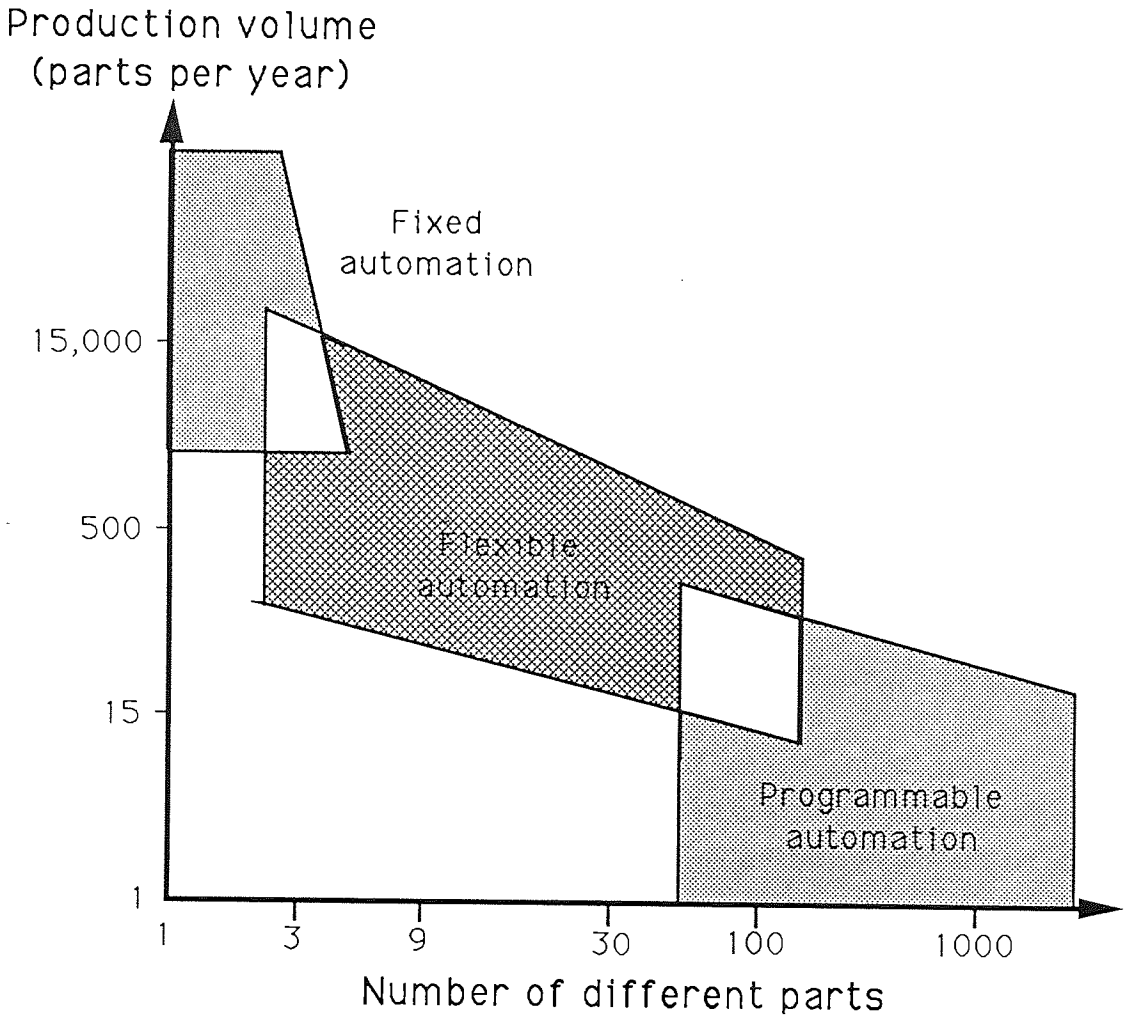


Figure 1.1 Machine and product characteristics

1.1.2 Conventional design of machines

High-speed machines have conventionally been designed according to a philosophy of using cam-actuated, mechanically-linked mechanisms to process and combine materials as they move through the machine to produce a finished product.

The mechanisms inside the machine can have complex motion requirements. The set of mechanisms are mechanically coupled via a drive-train comprising a series of shafts, gears and pulleys to a central prime mover. An example of a complex drive train is shown in figure 1.2. The central prime mover provides power to the drive train and also

facilitates speed control of the machine. The drive-train provides fixed synchronisation of the machine mechanisms.

The cams provide tight control of the actuator functions by providing motion control synchronised to the drive-train. Chen [2] describes how the motion control provided by a cam may be continuous, with no dwells in the motion profile or it may be intermittent, allowing dwells.

The mechanical couplings serve to coordinate and synchronise cam motions. The synchronisation function of a mechanical linkage may involve continuous synchronisation, in which two or more mechanisms on the machine are held in mutual synchronisation at all times, or intermittent synchronisation in which two or more mechanisms are brought into synchronisation at specific spatial positions or points of time. Both the cams and the mechanical couplings distribute energy to the product manipulating mechanisms about the machine and redistribute energy during braking.

An emergency stop safety function for the machine can be accommodated by fitting a fail-safe brake to the main drive shaft together with electrical interlocks to disable the main drive motor. The fixed synchronisation of the machines mechanisms ensures mechanical clashes do not occur during emergency stops.

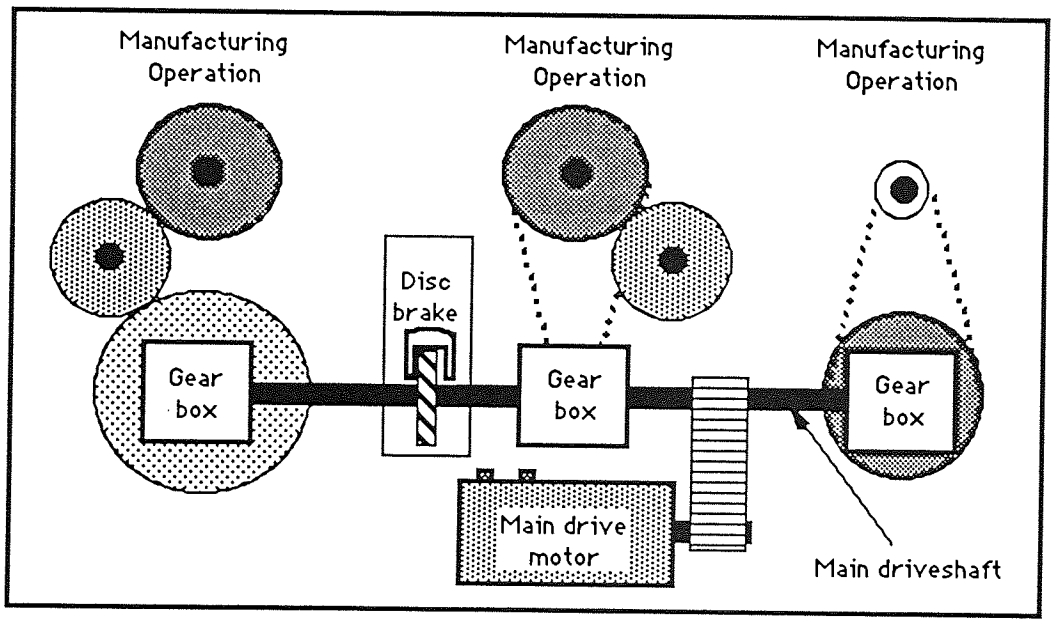


Figure 1.2 Machine showing complex drive train and central prime mover

1.1.3 Evolution of machines

Hall [3] describes how current Molins manufacturing machines are an evolutionary development of earlier models which can be traced to machines designed in the early twentieth century.

The speed of operation, quality of output and the functions provided by the modern machines are superior to the first models due to improvements in materials, design and manufacturing methods. For example a Molins Mark 1 cigarette packing machine of 1923 could produce up to 1000 cigarettes a minute of variable quality, whereas a Molins Mark10-N introduced in 1988 can produce 10,000 cigarettes a minute with tolerances of 1% on specification.

The evolutionary approach to improving machine performance is discussed by Fenny et al [4] who noted that this approach is used by many machine manufacturers. Although there has been continuous development in machine design over the years, with improvements in materials and mechanisms, the underlying design philosophy of a central prime mover and complex drive-train powering cam mechanisms has not changed.

1.1.4 Advantages of conventional systems

Conventional machine designs have proved capable of maintaining correct and safe machine operation over a range of normal and abnormal operating conditions. Strandh [5] describes how the mechanical mechanisms used in the machines are both reliable and durable. However with the trend toward modular machines which operate at higher speeds and support more flexible operation problems have been encountered with conventional machine designs. The underlying design philosophy limits the extent to which these machines can be adapted to meet the new requirements.

1.1.5 Limitations of conventional systems

Problems occur when attempting to introduce flexibility into conventional designs. For example in short production run applications, where the machine functions are reconfigured between runs to accommodate changes in product specification, the

inflexible nature of the fixed function cam mechanisms hinders the implementation of such alterations. As a result product changes can take considerable periods of time, leading to reconfiguration times which can be high in comparison with production run times.

A modular design is not achieved easily with the fixed coupling resulting from the use of a single drive train, since this coupling constrains the designer by forcing the manufacturing operations of the machine to be centred around the drive train and not the product being manufactured. In consequence the maximum manipulation speed of raw material in the machine may be reached at lower overall production speeds. This is a result of the material being moved around the machine rather than the machine being designed around the material flow.

Fogarasky [6] has identified that the speed of conventional machines can not be increased easily because high production speeds introduce fatigue and fretage in the mechanical parts. In order to compensate these parts have to be redesigned or reinforced and this often leads to the use of increasingly massive machine elements. Such machines are expensive to produce and generate high intensity acoustic noise.

The complex drive train of a conventional machine is expensive to design and produce. Costs can rise disproportionately when increased machining accuracy and finishing are necessary to construct complex high-speed mechanisms.

If the machine uses an open-loop control system, with no knowledge of product and machine state available from actuator feedback, then faults or abnormal machine operation may only be detected by the machine operator. The operator must then clear the fault manually or initiate an emergency stop. If sensor feedback is available a conventional machine may still be limited to performing an emergency stop rather than continue operating until the fault is cleared. This is due to the inflexible control of the machine actuators.

Work aimed at improving the performance of machines has begun to move away from the evolutionary development of conventional machine configurations, in favour of considering alternative design philosophies.

1.1.6 Driving points in machine development

Rosegger [7] states that users of manufacturing machines require reliability, efficiency and maximum machine utilisation in order to obtain a good return on their investment. To increase both utilisation and efficiency there is a trend in manufacturing machine design, identified by Russell [8], towards higher production speeds and greater flexibility.

If this flexibility is available in a rapidly reconfigurable manner then one machine can cost effectively produce a variety of products. This is achieved by operating the machine for a short duration at high-speed, reconfiguring to a different product and then running again, increasing the utilisation. Leighton [9] discusses a case of a company requiring a single machine to produce many similar varieties of product.

If the machine can operate at high speeds with reduced wastage without compromising reliability then efficiency is increased. Thus the users require machines which have the flexibility to allow rapid reconfiguration of manufactured product, whilst also achieving the close tolerances necessary for high efficiency.

Modern trends in automated production have added further requirements, Hidde et al. [10] details the requirements for integrating a single manufacturing machine into a production line of material preparation, making, packing and cartoning machines. The integration of machinery into a plant wide control system enabling flexible manufacture whilst maximising utilisation is examined by Tistar [11].

Machine developers are also constrained by the economics of production. To perform successfully they aim to reduce the time taken to design and commission a new machine. Boehm [12] proposes that this allows a faster response to market demands and a reduction of risk associated with development.

Johnson [13] shows how these aims can be achieved by adopting a modular design approach which allows a greater freedom of design. Standardisation of components and assemblies allows reuse of existing machine parts in new machine developments and simplifies commissioning and maintenance.

1.2 A design philosophy - Independent drives

To achieve an advance in machine design it has been necessary to re-examine the role of the mechanical linkages and cams, which have been identified, by associated research at Aston [14][15], as the principal limiting factors in conventional centrally driven, cam-actuated, high-speed machines.

The aim of the analysis was to identify precisely the function of the cams and linkages and to derive alternative ways of achieving these functions. The objective was to identify approaches which either introduced flexibility or allowed the use of light robust mechanisms for high-speed operation.

1.2.1 Function of conventional components

Analysis by Fenny et al [4] showed that the mechanical linkages served two distinct functions: they distribute power around the machine to the various product manipulation mechanisms and they coordinate and synchronise these mechanisms. The cams provide tight control of the mechanism functions by providing motion control synchronised to the drive-train.

Stamp [16] describes how in many cases the mechanisms of a high-speed machine must be capable of satisfying tight specifications on performance and accuracy, often under demanding load conditions.

1.2.2 New methods of achieving required functions

Flexibility can be introduced into the problem of power distribution by replacing the central prime mover and mechanical linkages by a set of independent drives which deliver power direct to the point of use. Figure 1.3 shows an IDM implementation of the machine functions of figure 1.2.

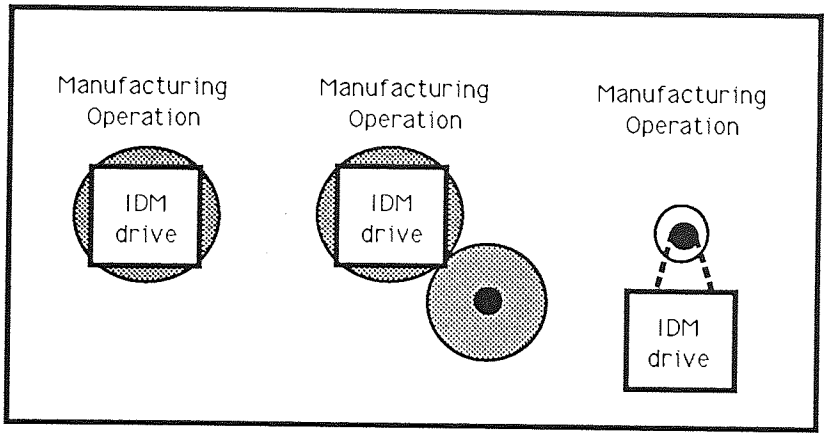


Figure 1.3 Machine design showing independent drives

When each product manipulating mechanism is driven independently, the total energy demands of the mechanism must be supplied by it's drive. This is unlike conventional designs where inertia in the drive-train can assist the central prime-mover by absorbing and redistributing energy. In the case of independently driven mechanisms which perform intermittent motions or motions that are subject to large disturbances, energy requirements can place severe demands on mechanism drives and careful consideration has to be given to the selection of appropriate drives.

The use of independent drives also allows flexibility to be extended into the mechanism function. Quinn [17] describes how cam driven mechanisms can be replaced by appropriate independently driven mechanisms, such as software controlled electro-mechanical actuator mechanisms, in which the actuator and hence the product manipulating mechanism function can be changed through programming.

The proper control of independently-driven mechanisms is essential if they are to carry out their function reliably, it is also a prerequisite for their use in multiple drive systems.

When the centrally-driven mechanically linked systems are replaced by a modular system based on independent drives, the drives must be forced into the correct motion trajectories and synchronised by an appropriate control system. In the case of software-based solutions, the synchronisation and motion control will be implemented by programming.

1.2.3 The components of an independent drive machine design

A manufacturing operation is achieved by performing a number of manipulative and process tasks on raw material to produce finished product. A modular approach to machine design constructs discrete modules to perform each task.

Each task may require a number of IDMs. Associated work at Aston [14][15] has identified that an IDM may comprise an energy source or drive which actuates a mechanical mechanism and effectors to manipulate product; a servo-controller to give tight control of the mechanism's dynamics; a trajectory generator to provide motion control of the drive; and a means of monitoring the IDM's performance to ensure it is functioning correctly. Figure 1.4 part (1) shows the components of an independent drive mechanism.

Each IDM is then integrated into a machine module by a suitable control system, which may provide synchronisation and motion control for the IDMs. Figure 1.4 part (2) shows a grouping of IDMs to form a machine module. Finally the set of machine modules are linked by the computer control system, which may provide synchronisation and motion control for the modules, to form the final machine, shown in figure 1.4 part (3).

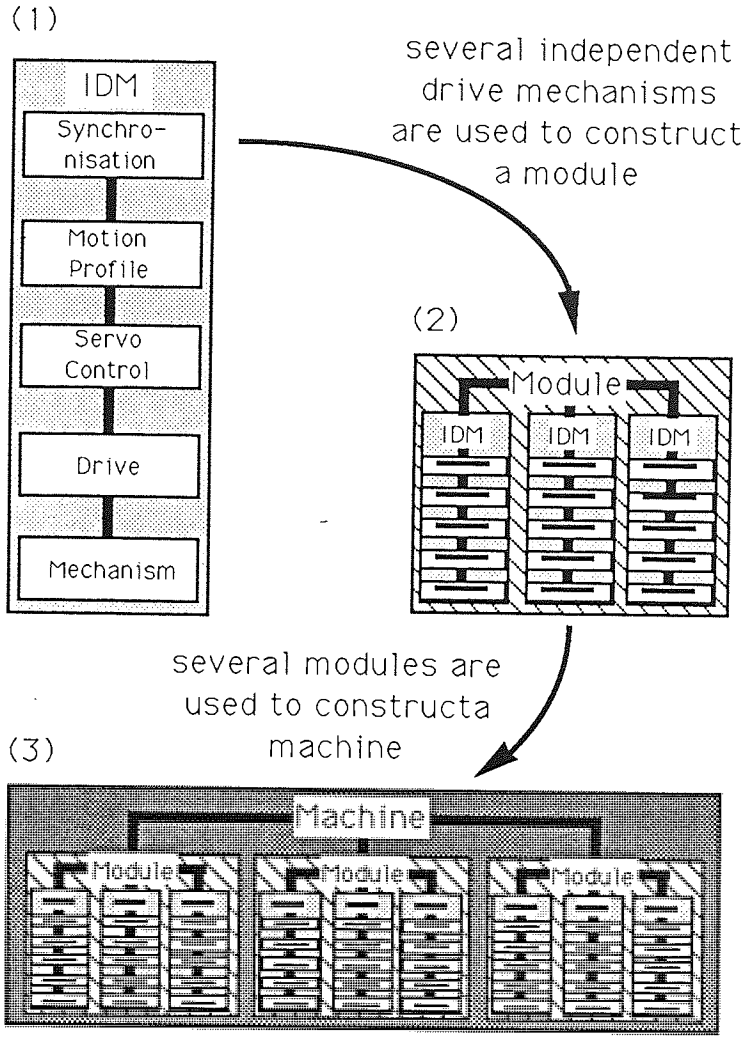


Figure 1.4 Components of an independent drive machine

1.2.4 Liberalisation of design - advantages

An independent drive machine has the capability to change it's function since actuator functions are programmable. Therefore modifications to the manufactured product are possible without changing mechanical components. For example a wrapping machine may accommodate different size packs by altering the pack foil feed length in relation to pack size, the feed length being determined by a programmable independent drive.

The design of computer controlled independent drive machines readily accommodates the concept of modular structure, which is a design goal to achieve faster machine development and allow reuse of machine components. For example Fenny [15] describes

how mechanical modules may be formed from standard mechanisms, such as four-bar linkages and transfer sliders, or sets of mechanically coupled mechanisms, and an appropriate controlled drive.

A modular approach can also be applied to the controlling software and hardware. A multiprocessor or distributed computer system provides hardware modularity. Whilst modular decomposition techniques and multiprocessing provides software modularity. These modular computer structures may be used to control, coordinate and synchronise the various mechanical modules.

Closed-loop computer control allows feedback of machine state to higher levels of machine control. Williams [18] describes how feedback of information in a hierarchical machine controller may be used to safeguard the machine under abnormal operating conditions by taking action to avoid machine stoppages and if necessary to provide fail-safe shutdown.

A modular design also supports standardisation of components and an easier upgrade path, since only the modules which restrict performance require modification. Commissioning and maintenance is simplified since modules may be tested individually, whilst faulty modules can be replaced and then repaired off line.

1.2.5 Disadvantages and problems

The liberation of design allowed by programmable independent product manipulation mechanisms greatly increases the flexibility of operation of a machine.

However when using multiple independent drives the proper functioning of the machine resides with the computer control system and not the linkages and cams.

Holding [19] examines implementation of flexibility in independent drive machines and concludes that flexibility can only be safely realised if it can be ensured that the set of drives are properly coordinated and synchronised. In order to ensure that the synchronisation is correct and is implemented in a reliable and safe manner careful consideration has to be given to the specification and design of the control system.

The control system must also monitor for abnormal operation within the machine, to allow corrective action to be taken to deal with product jams, drive/mechanism failures and other exceptional conditions. The action taken to these abnormal circumstances must be detailed in the specifications of the machine, accordingly the specification of machine function in all operating conditions is a demanding task.

The operator of an independent drive machine may have the facility to change the function of the machine. The machine controller must accommodate this flexibility of function, however Leveson [20] describes how a computer controller must ensure safe operation in the presence of inappropriate operator requests.

Fenny [15] states that methods for the specification and design of computer control systems are required in order to successfully develop modular IDM machine systems. In particular methods are required for: the capture of machine requirements for motion and synchronisation; validation to transform these requirements into feasible and correct machine specifications; the design of a modular computer controller which embodies the specifications and provides safe operation; and the implementation of the design with verification of the controller operation and fail-safe features.

1.3 Computer control of independent drives

The components of an independent drive module for use in a machine were introduced in section 1.2.3. Many of these components can incorporate computing elements. This section discusses the reason control is required, the types of control used and the particular case of programmable elements in control systems.

1.3.1 Reason for control

Control is necessary in a machine to direct and constrain the manipulations of mechanisms which transform raw material into product. Hannah [21] describes the principle control elements of conventional machines as the speed control of the prime-mover, the motion control of cams and gearboxes and the synchronisation control of mechanical linkages. Further control elements are required to deal with abnormal operation, such as safety interlocks and process fault detection.

These areas of speed, motion, synchronisation and safety control are required in independent drive machines. In addition control to permit flexibility in mechanism function is necessary to support the design aim of reconfigurable machine operation.

1.3.2 Types of control

Control can be achieved in many forms. In the earliest machines the human operator provided the control. Advances in mechanical design led to the development of automatic control elements, such as the centrifugal governor used by Watt. Mechanical mechanisms and mechanical control elements are the basis for conventional machine designs, where mechanical control is used extensively in the motion and synchronisation control tasks. Strandh [5] describes how analogue electronic continuous control has been used in association with electro-mechanical prime-movers, developing the concept of servomechanisms and servo control.

Simple logic circuits were introduced to support sequencing and safety control. Relay ladder logic, detailed by Noble [22], provided a foundation for introducing Programmable Logic Controllers (PLCs) into machine designs. Ziemba [23] describes how digital electronic control and PLCs are now widely used in conventional machine control systems and for integrating multiple machines into a production operation. They can provide both continuous and discrete logic control. Other control technologies such as hydraulic and pneumatic logic control may also be applied, however Leskiewicz [24] states that these are more often associated with actuator technology.

An IDM machine may incorporate mechanical, electronic, digital, programmable digital, pneumatic and hydraulic control technologies. The systems analysis process used to specify machine requirements and develop a preliminary design for the machine control system must accommodate input from these technologies. This process is difficult since pertinent information relating to specific technologies must be extracted from appropriate development engineers, the captured information must then be integrated into a coherent set of machine requirements.

1.3.3 Software controlled devices (programmable systems)

Digital control elements have been implemented using hard wired logic. This is suitable for simple Boolean logic and sequencing operations. However complex logic such as digital feedback control requires arithmetic calculations, Warwick [25] shows how these may be performed using processing elements such as microcontrollers or microprocessors. These microcontrollers and microprocessors are programmed to perform the required control functions and are inherently flexible.

A single microprocessor may perform many control tasks whilst also providing other required system functions. This flexibility has resulted in the widespread use of microprocessors in control systems, often replacing discrete digital control elements. The multiplicity of function supported by a microprocessor can reduce the physical complexity of a controller, however the software which drives the microprocessor to provide the required functions can be complex.

The logical complexity of a microprocessor control program may introduce problems, Leveson [26] demonstrates how programs which contain errors may cause unpredictable and unsafe behaviour of the controlled system. The generation of programs which incorporate the required control functions is also demanding due to the difficulty of capturing the required system performance from system designers. In particular the correct specification of flexible operation of the IDMs in an IDM machine is critical so that the computer controller can not only provide the flexibility in operation but also control IDM interaction to allow fault-tolerant and fail-safe operation. A method is needed for the capture and validation of IDM motion and synchronisation. The correct programming of any digital control system element is of great importance in building a safe and reliable IDM machine.

1.4 Aims and objectives of the research

The phase one SERC/MOLINS projects at Aston University [14][15] have addressed the problems of developing the independent drive mechanism technology to allow high performance operation. The integration of multiple independent-drive mechanisms by computer control, to construct modular machines, has been identified as a fruitful area for further study by Fenny [15].

The work described in this thesis aims to build on the knowledge base and technology established in the work of Fenny and Seaward by examining the specification of IDM machine operation and deriving methods of generating computer control systems capable of meeting these specifications.

An objective of the research is to identify suitable tools for the specification of machine requirements which can be used by and produce deliverables for a multi-disciplinary engineering design team. These tools will be used in a sequence of analysis phases which will result in a machine requirement specification via a well defined route. The proposal of a defined requirements capture and specification method for an IDM machine project is intended to impose a rigour on the initial stages of machine development currently lacking in industry.

In order to specify the operation of an IDM machine the thesis aims to classify the purpose of an IDM according to product operation; to identify heuristics for partitioning IDMs to create machine modules; and to classify the logical relationships between IDMs according to types of motion and synchronisation employed.

To ensure a correct, complete and feasible machine specification is produced methods of requirements validation are proposed. The validated machine requirements specification document can then be used to drive the initial design of the computer controller.

The thesis aims to provide a generic design for an IDM machine computer controller based on a hierarchy of control in the machine supported by a modular software structure. Tools will be identified for the design of the computer controller which can accommodate the IDM philosophy of distributed control, flexibility in operation of components, reusability and safety. A rigorous notation will be used in the design of safety critical components. The thesis will concentrate on a class of mechanisms which perform intermittent motions and can be discretely synchronised using mechanism position.

An implementation of the generic controller design will be presented for an IDM demonstrator machine proposed by Molins. The computer controller implementation will use a programming language and hardware platform which has a close match to the design notation. The goal of the IDM machine is to demonstrate the flexibility of IDMs in performing different product operations; to show practical behaviour of IDMs under normal and abnormal operating conditions; and to demonstrate the suitability of a specific

hardware/software configuration, Occam and the Transputer, for use in real-time control systems.

1.5 Outline of thesis format

The work has been split into seven Chapters with appropriate appendices. The ordering of the material follows the chronological life cycle of a computer control system development to emphasise the structured approach the thesis advocates.

1.5.1 Chapter 1 introduction to HSM

This Chapter has introduced the concept of independent drive mechanisms as a method for introducing flexibility and higher-speeds into modern manufacturing machines. The use of modular computer control systems in the construction of these machines has been proposed. The research goals of providing generic methods for the specification, design and implementation of IDM machine computer control systems have been detailed.

1.5.2 Chapter 2 technology review

Chapter 2 provides a critical literature review, examines the use of independent drives in machine systems and details the IDM demonstration machine used as a development example throughout the thesis. It examines technological developments in associated fields of robotics and process control to identify common technology and design methods. Real-time computer control is discussed with particular reference to methods of requirement capture, specification and design suitable for the IDM machine problem. Tools and techniques appropriate for IDM machine systems analysis and design are examined, those tools best suited to the IDM machine problem are discussed in detail.

1.5.3 Chapter 3 capturing the requirements

Chapter 3 develops a method of capturing the functions required of the various IDMs of the machine. It details a sequential method for capture and specification of the various

design data required and examines the relationship between different sources of data. Deliverables of capture phases are detailed. Classifications of IDM purpose, logical relationships between IDMs and heuristics for partitioning IDMs into machine modules are introduced.

1.5.4 Chapter 4 validating the requirements & creating the machine requirements specification

Chapter 4 takes the machine requirements generated in Chapter 3 and details a rigorous method by which the requirements may be validated. This process is necessary to remove inconsistencies, omissions and errors in the requirements. The goal is to generate a validated machine requirement specification which can be used to drive an implementation.

1.5.5 Chapter 5 the design of a computer controller which embodies the machine requirements

Chapter 5 introduces a generic design for an IDM machine computer controller based on a hierarchy of control in the machine supported by a modular software structure. Partitioning of the controller to build a modular computer control system based on the Transputer microprocessor and Occam programming language is introduced as a precursor to controller implementation.

1.5.6 Chapter 6 controller implementation and results

A prototype controller for the IDM demonstrator machine is described as is the implementation of the modular computer controller introduced in Chapter 5. The prototype and final implementation controllers are used to demonstrate the flexibility that programmable IDMs can achieve. Operation under normal and abnormal operation is discussed. The verification of the final controller against design specifications is detailed to demonstrate how quality assurance of software can be provided by the development method.

1.5.7 Chapter 7 conclusions

Chapter 7 examines the requirements specification, design and implementation methods used in the thesis. A critical review of the methods as applied to full scale machine projects is provided. Conclusions on the suitability of the tools and techniques are drawn and recommendations for further work are made.

1.5.8 Appendices

Pertinent information which was not incorporated into the main body of the thesis can be found in the appendices. It includes computer program listings and published papers.

1.6 Conclusions

This Chapter has introduced the author's research on computer control of independent drive mechanism machines. It has outlined the research goals of deriving generic methods for the specification, design and implementation of modular computer controllers for IDM machines and constructing an IDM demonstrator utilising intermittent motion IDMs together with position based discrete synchronisation.

The nature and characteristics of high speed machines have been discussed together with the trend in machine design towards higher-speeds and more flexible operation. The conventional design of machines together with the advantages and disadvantages of the method have been discussed. A new design philosophy based on IDMs operating under software control has been introduced. The components of an IDM machine have been detailed and technologies used to construct IDM machines discussed. Problems specific to computer control have been highlighted. The research goals and objectives have been introduced. Finally a summary of the Chapters of the thesis has been given.

----------//----------

Chapter 2

Literature and technology review

2.0 Introduction

Chapter 1 introduced the concept of computer control coupled with programmable independently driven mechanisms (IDMs) to provide flexible operation of high speed manufacturing machines. The need for a method of specifying IDM operation and developing a computer controller capable of implementing these specifications was identified in section 1.4.

This Chapter provides a critical review of pertinent literature and technology required to construct an IDM machine. The literature and technology review comprises three sections. The first section examines multiple-drive system applications to extract relevant design features and techniques. The second section discusses the mechanical, electrical and computer technology required to implement an independent drive machine, highlighting areas where research work is needed. The third section provides a critical comparison of various methods of computer controller requirements capture, validation, specification and implementation in order to identify an appropriate development technique for an independent drive machine controller.

In section 2.4 a method for the development of a computer controller for such a machine is proposed. The two main tools of the proposed design method, Data flow diagrams and Petri-nets are described in detail. Finally a demonstration machine used to illustrate the application of the development method is presented.

2.1 Machines using multiple independent drives

The use of independent drives in high-speed packaging machinery is a recent development. The projects at Aston described by Seaward [14] and Fenny [15] which

started in 1986 being some of the first British studies in the field. However other control applications such as robots, computer numerically controlled (CNC) machines and Programmable Logic Controller (PLC) based automation systems, have been utilising multiple drives or multiple control loops for a considerable time. A study of these applications gives an insight into the technology and control strategies necessary to utilise multiple independent drive mechanisms in high speed manufacturing machines.

2.1.1 Robots

A significant number of robots have been classified by Nazemetz et al. [27] as jointed armed, flexible automate. Early robots utilised single processor computer hardware. However Andeen [28] describes how the trend in robot control is towards a multiprocessor configuration. Naghdy [29] describes an example of this topology comprising a distributed robotic system based on multiple Transputer microprocessors.

Distributed processing is examined as a means of speeding-up calculation of the kinematics of the various joints by Zalzala and Morris [30] who have also used a modular software approach based on a network of Transputers. Physical distribution of the computer hardware is not used since the robot is relatively compact. The use of multiprocessors forming a distributed processing topology also supports the IDM machine design goals of modularity in hardware and software whilst providing high-performance processing.

The motion trajectory of a robot can be programmed on line by the use of a teach pendant. On line programming of IDM machines is not possible due to the absence of physical coupling between the independently driven mechanisms.

Off line motion programming of robots is usually achieved using a proprietary language such as VAL II detailed by Braganca et al. [31] or APT described by Grotzinger [32]. These languages are tailored for the description of mechanically linked sets of drives with a single end-effector unlike IDM machines which have no mechanical links between drives and many product effectors.

The mathematics required to generate the motion profiles necessary to obtain movement of a robot are different from those required in multi-axis machinery since each robot axis is mounted and moves relative to another axis. Inverse kinematics, discussed by Paul et al. [33], are used to obtain the demands for each joint's motion given a required end-

effector position. The mathematics used to generate motion profiles for IDMs use pivotal values from motion specifications together with curve fitting techniques, such as linear, quadratic and cubic spline interpolation detailed by Kreysig [34].

The type of flexibility required of robots is different to that of high-speed manufacturing machines. Vail [35] describes how robots usually operate at a single speed, are functionally very flexible in performing a wide variety of manipulative tasks and operate in low to medium volume production environments, they occupy the flexible automation area of figure 1.1. IDM machines require more limited functional flexibility for changing the operation of the machine but require dynamic flexibility when running, for example to provide a range of production speeds, and they must also be able to operate at high speed for high volume applications.

2.1.2 CNC Machines

Computer numerically controlled machines are used to machine discrete components. The component is located on a positionable bed and a tool is used in the machining process. The movement of bed and tool uses multiple IDMs and is programmable giving the machine flexibility to manufacture many different components.

The axes of the machine are controlled by a centralised processor often running commands generated from an interpreted 'part program'. Gibbs [37] describes how programming of the machine is achieved off line. The part programs use a programming language specific to the CNC machine.

Control of a CNC machine using a supervisor computer to download part programs and initiate operation, described by Gupton [37], is termed direct numerical control (DNC). DNC provides a hierarchical control structure which integrates individual CNC machines into a larger manufacturing operation.

Operation of the machine is based around the product being manufactured. Koren [38] demonstrates how CNC machines are often used in flexible manufacturing system (FMS) cells. The FMS cells being controlled by a supervisor computer. This hierarchical control structure is a possible template for an IDM machine controller since it supports modular, distributed controller design and multiprocessing.

Like robots CNC machines offer very flexible operation to produce many different components. IDM machines are much less flexible in operation than CNC machines but perform continuous operations on various raw material to produce finished product.

CNC part programs are required for each component manufactured. IDM machine control programs will be developed for a particular product type with flexibility incorporated to change parameters of the product. Thus CNC machines may produce many different products using interpreted part programs whereas IDM machines produce families of related products using compiled product programs with in-built product parameter flexibility.

2.1.3 Programmable Logic Controller (PLC) systems

Programmable Logic Controllers are computers designed to fulfil industrial control requirements, Puente [39] describes how they are characterised by real-time operation of multiple Input/Output points and communication between PLC units. They are incorporated into a diverse range of manufacturing operations from complete production lines such as steel rolling mills to stand alone machines like the injection moulding machine detailed by Ziemba [23].

The capabilities of PLCs to control manufacturing operations, detailed by Webb [40], include sequence control using discrete IO, closed loop analog control of dynamic processes and networking facilities. Warnock [41] shows how the ability of a PLC to perform continuous control functions has allowed their use in larger applications for the process industries such as petrochemicals, paper production and steel making. However these processes require control loops with only a limited number of set-points unlike independent drive machines where complex trajectory generation is required. IDM motion is described further in Chapter 3.

Programming of PLCs is achieved in several ways, ladder logic diagrams are well established but lack the descriptive power required for the continuous control and communication facilities of advanced PLC systems. Graphical programming languages, based on state machines, such as G++ or Grafcet, introduced by Fanard [42], can accommodate advanced functionality while simplifying the programming of these devices by separating the continuous and sequence control being described. Program development for multiple PLC systems using these languages are supported by computer aided software engineering CASE tools, for example Telemecanique [43] supports

Grafcet. These CASE tools simplify and speed program development, however each tool only relates to a single manufacturer's PLC products and programming of different manufacturer's PLC types is impossible.

PLC programming based on state machines is a technique for sequence control which can be readily transferred to IDM machine controllers, unfortunately the CASE tools to support this technique are not portable.

The use of PLCs in a FMS work cell to link automation components, such as robots for material / component manipulation and CNC machines for component machining, is described by Vail [35]. This FMS approach suits a low speed, multiple component manufacturing and assembly operation based on sequence control of discrete operations with limited continuous control. This is different to the operation required of IDM machines which use multiple high-performance continuous control loops together with sequence control.

PLCs can support physically distributed processing solutions due to their modular nature and communication facilities. The application of PLC to the control of flexible independent drive machine applications is limited due to the problem of complex motion trajectory generation and speed of control loop sampling. Seaward [14] describes how a 1ms sample rate is typical for IDM control whereas sample rates of 50ms are common for PLCs. However the integrated programming environments using easily assimilated graphical notations and the technical sophistication of PLCs under development means that there is potential for the application of PLCs to future independent drive machines.

2.1.4 Manufacturing machines

A number of high speed independent drive mechanism manufacturing machines have been constructed whose technical details are available. Most of these have been demonstrators or linked with academic research. Leighton [9] describes a machine for feeding metal sheets into a stamping machine. Very high precision positioning is required, to within 1mm on a 1000mm sheet. The sheet feed/extraction system incorporates four independent drives controlled by a centralised computer. In this machine intermittent motion and intermittent synchronisation has been used to facilitate simplified interlocking with the stamping machine controller.

A wrapping machine demonstrator utilising five independent drives is described by Seaward [44], the machine can wrap varying size packets at rates of up to 750 per minute. The use of independent drives permits a more compact design than has conventionally been used, with fewer mechanical components. Three of the machines' IDMs use intermittent motion profiles, the remaining two continuous motion profiles. The IDMs are continuously synchronised with no dynamic flexibility in motion or synchronisation, the machine operating at a single speed. To facilitate a speed or pack-size change the computer controller is loaded with a different IDM motion profile data file, this reconfiguration process taking 2 minutes. The machine utilises three microprocessors.

A Molins printing machine incorporates twelve independent axes. This machine requires six processors cards mounted in a VME rack system. In addition a bitbus field-bus network connected to seven distributed intelligent processor nodes is used. The machine has a dynamically variable operating speed from 100 to 1000 sheets per minute with accuracy on print to within 0.5mm. It can be reconfigured in three minutes to perform one of 4 different printing operations. Motion profiles are precalculated for each print operation, however the motion data is scaled dynamically to accommodate varying speeds of operation.

The problems experienced in developing the last two systems described included: Design information for specifying motion and synchronisation was difficult to transfer between mechanical and software engineers due to a lack of a common notation. The correct implementation of the necessary motion profiles for the IDMs was labour intensive since iterative prototyping was used to evaluate different profiles for a mechanical mechanism, subsequent changes to the mechanism design required development of new motion profiles.

Multiprocessing was required in order to provide the necessary processing capability. Assembler coding of motion data dispatchers was necessary to support the high speed servo control loops of 1ms.

The problems specific to the computer control system were: The constraints on computer hardware topology due to the large system processing requirement; a lack of standardisation of control system design and implementation resulting in poor modularity of software; and a lack of validation and verification methods for ensuring the controller achieved specifications for performance and safety.

2.2 Component technologies

The components and control technologies used in IDM high-speed machines were introduced in Chapter 1, and can be seen in figure 2.1 which shows components of a Molins maker machine. The components have been partitioned into product, mechanism, drive, sensor, electronics and computer technologies. These six categories are discussed individually in sections 2.2.1 to 2.2.6. The aim is to identify features and constraints of various components that will affect an IDM machine design. This will facilitate the creation of a more considered machine requirement specification due to a greater understanding of the operation and function of machine components and their interdependencies. The design of the computer controller will be affected accordingly.

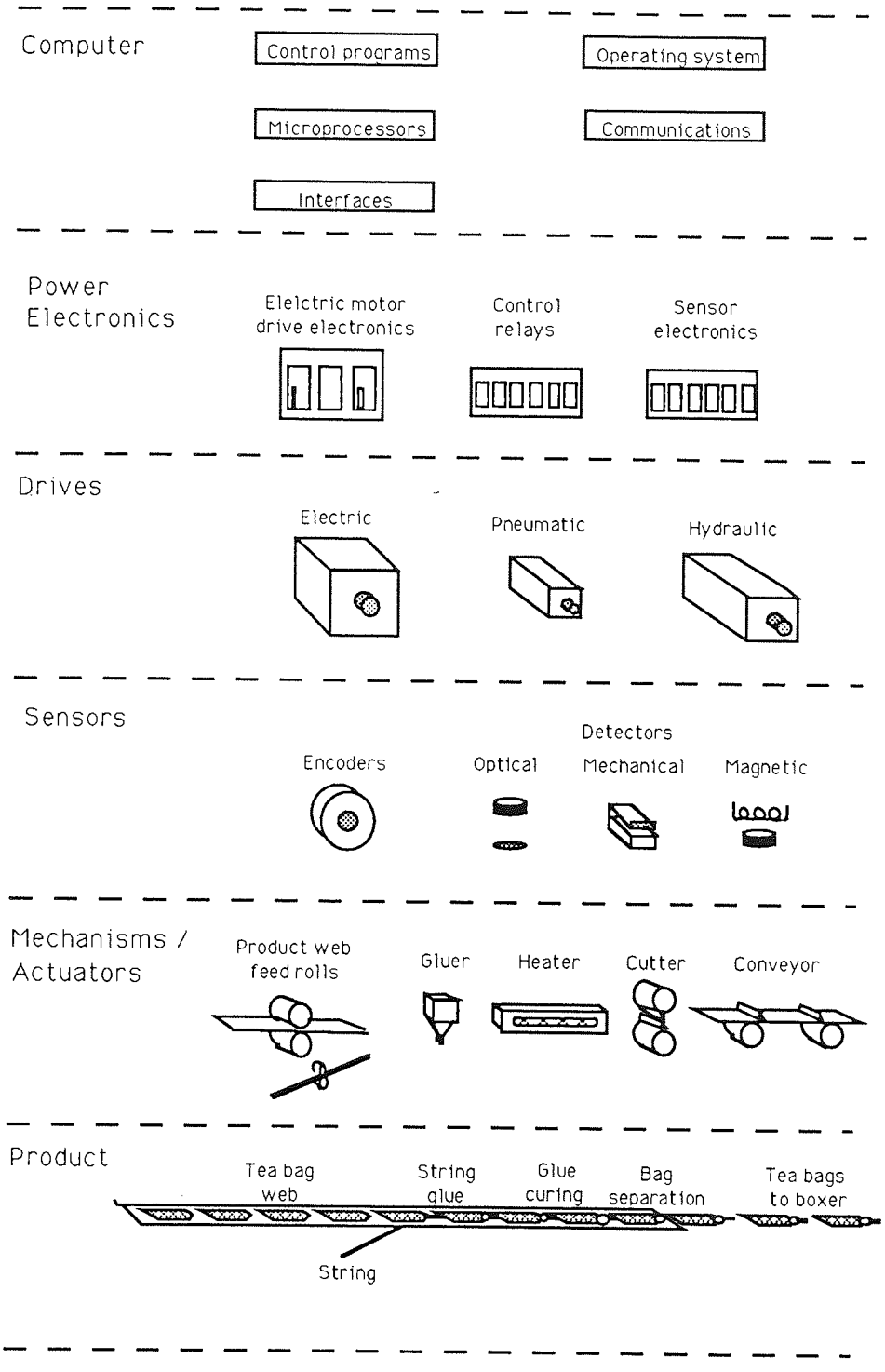


Figure 2.1. Component parts of a typical IDM machine (Molins maker)

2.2.1 Product technology

The liberation in mechanism design achieved by removing the central drive train allows IDM machines to be built around the product to be manufactured, reducing product manipulation and resulting in a more compact, less mechanically complex machine.

An IDM product operation may have lower product manipulation speeds in comparison with a conventional design, this allow higher product throughput before product damage takes place and hence increased maximum operating speed.

IDM mechanisms can be used to increase material economy by using more sophisticated motion profiles than are available with conventional mechanisms.

The flexibility in the IDMs and the computer controller facilitates changes to the manufactured product, a series of related products being produced efficiently by one machine, due to rapid reconfiguration of IDMs to support parameter changes which accommodate changes in product specification. This allows group technology approaches to manufacturing described by Hyer [45].

2.2.2 Mechanism technology

A mechanism is defined by Naylor [46] as a system of mutually adapted parts working together. They may not however constitute a complete machine. In the context of an IDM machine this thesis proposes a mechanism to be the mechanical connection between motor or actuator and the product.

Fenny [4] describes how an IDM does not have the large inertia of a drive train to use for energy storage when driving a dynamic load. Accordingly the independent drives must both inject and remove power from their associated mechanisms when performing motions to move loads. To reduce the amount of energy required for acceleration and braking of IDMs light robust mechanisms, with reduced inertia, such as those described by Stamp [16], are being developed.

Small load disturbances on IDMs are important since this reduces the energy required by the driver to move the load, this is particularly important if inertia cannot be reduced. Dynamic balancing of loads on mechanisms attached to a single drive may also be used to reduce load disturbances.

Groups of drives and associated mechanisms can be linked to form a manufacturing task or module. Integrating sets of manufacturing modules into a composite machine accomplishes the overall machine function of converting raw material into finished product. Each drive-mechanism pair with it's associated computer controller is referred to as an independently driven axis. The physical format of an independent drive machine can be seen in figure 1.4 of Chapter 1.

This modular approach to incorporating the required product manipulations facilitates separate development and commissioning of each manufacturing module. Module reuse in other projects is possible and maintenance and upgrading is eased by replacing modules affecting performance.

2.2.3 Sensor technology

Belove [47] provides a comprehensive description of sensor types and their interfacing. A fundamental difference between conventional and IDM machines is the reliance of IDM machines on feedback of machine state to allow closed loop control of mechanisms. Typically this involves encoder or resolver feedback of position allowing closed servo-control loops of the drives of the IDMs. Discrete sensors are also used by linear actuators to perform simple closed-loop logic control of these actuators.

It is possible to classify the purpose of machine sensors into one of two categories. The first category deals with dynamic detection of an event in the manufacturing process, such as the presence or absence of a pack in a product stream or extension/retraction of a linear actuator, the second category deals with enabling or inhibiting events, such as the detection of open machine guards, or activation of an emergency stop switch. Thus individual sensors are used both in normal machine operation and to detect abnormal conditions which require exceptional process control.

2.2.4 Drive motor and actuator technology

Each of the electromechanical drives used in an IDM machine supplies power, in the form of torque or force and motion by rotary or linear displacement, to an independent product manipulating mechanism of the machine. Seaward [14] identified that a drive for an IDM should possess high-torque over a wide speed range, fast acceleration and low-

inertia. Components suitable for this purpose include electric motors for rotary motion and actuators for linear motion.

Considine [48] compares three commonly used types of electric motors applied in servo-systems: AC, DC and Brushless DC motors. AC motors suffer from low starting torque, slow acceleration and torque breakdown at overload. DC motors are straightforward to apply, they have predictable torque speed characteristics and do not suffer the speed problems of AC motors.

Brushless DC motors are more efficient than other motor types and are more compact. These motors have good torque speed characteristics, continuous and peak, with high torque-to-inertia ratio, a measure of performance identified by Seaward [49] of particular relevance to IDM drives. Tal [50] shows how they exhibit fast acceleration due to low inertia rotors which are also suitable for inertia matching to actuators.

However Brushless DC motors require electronic commutation derived from rotor position feedback resulting in additional wiring to the motor and complex drive electronics. These requirements are mitigated when used for independent drive mechanisms since Quin [51] states that closed loop servo-control allowing accurate motion control and positioning is a necessity for many IDMs. Also brushless DC motors have a longer life since the absence of mechanical commutation removes wear and maintenance associated with brushes. The sealed unit design also allows brushless DC motors to be used in hazardous, such as potentially explosive, environments.

Seaward [14] performed studies on drive performance for IDM machines and concluded that brushless DC motors were most suitable for independently driving mechanisms, in particular the rare-earth stator models such as the Electrocraft BRU [52]. Seaward also concluded that direct drive of product manipulating mechanisms reduces backlash, increasing the bandwidth of the servo-control loop allowing higher speeds whilst maintaining accuracy.

Electric, hydraulic and pneumatic actuators, described by Fairchild [53], are often used for short stroke linear motion especially where simple motion profiles are sufficient for actuating mechanisms. Hydraulic actuators are more suited to mechanisms requiring high forces whereas pneumatic mechanisms are more suitable for lower force applications. The use of pneumatics in environmentally sensitive manufacturing, such as food preparation, is possible due to the inherently clean nature of the compressed air used for motivation.

2.2.5 Electronics technology

The electronics in an IDM machine perform a number of different tasks including interfacing computer systems to plant by converting command signals into power supplied to drives and converting sensor readings into computer system inputs; performing analog control of actuators and drives; and implementing communication networks between computing elements.

Drive control is provided by analog power electronics and supporting analog and digital control electronics. Tal [50] describes how brushless DC motors, commonly used in IDM machines, require electronic commutation of the field coils to operate the motor.

Computer controllers may transfer set-points as analog signals for conventional analog demand control or in a digital form implementing Direct Digital Control (DDC). Input / output from the computer for reading analog sensors and providing control signals to analog power electronics can be provided by analog to digital convertor (ADC) and digital to analog convertor (DAC) electronics, described by Belove [47]. Digital sensors and actuators can be connected using I/O ports mapped into the computer memory together with voltage conversion and isolating circuitry. Considine [48] describes industrial interface standards such as 24v DC, current loop and TTL [54] used to support these connections. Digital inputs are often used as input points for PLC type sequence logic in the computer controller. Digital outputs of the controller may be used for timing pulses of actuators such as glue-guns and bang-bang control of linear actuators.

Discrete switches and lamps are often used for operator interfaces. However the use of hard-wired digital I/O for the operator interface is inflexible and can be expensive. The operator input of commands to the machine can also be achieved by keypads whose keys change function. Display technology for identifying key functions and also for presenting machine status information is well established, for example employing cathode ray tube (CRT) [55], liquid crystal display (LCD) [56] and electro-luminescent (EL) technology with associated driver electronics. Combined display/input technology such as touch-screens, described by Quick [57], are being used to simplify operator interfaces, they also have the benefit of significantly reducing wiring complexity compared with discrete buttons.

Machine electronics should be protected against and should not produce electrical interference. The electronics should be designed to be fail-safe if possible since the intrinsic mechanical synchronisation of a drive train is absent in an IDM machine. Fenny [15] details how redundant wiring and system health checks reported to or performed by the higher level computer controller can be incorporated.

Legislation from bodies such as the British Standards Institute [58][59] requires that a computer controller operating a potentially hazardous system must be supported by a backup control system. For manufacturing machines hard wired electronics for emergency use can provide this secondary level of control. In the case of Molins machines an emergency stop circuit consisting of several thump-switches mounted around the machine are directly connected to the drive enables and main power relays. Other methods of secondary control using replicated computer controllers are prohibitively expensive.

2.2.6 Computer controller technology

The potential for flexible operation of IDM machines may only be fully exploited given a computer controller designed to utilise the flexibility of the independent mechanisms by suitable programming.

The computer system of an IDM machine replaces the control functions of the central drive train, cams and linkages of a conventional machine. The computer controller is responsible for motion control and synchronisation of the multiple drives, it is also responsible for machine safety by providing fault-tolerant and fail-safe operation of the set of IDMs under both normal and abnormal machine conditions. The computer controller is required to support sensor interfacing, the machine operator or human user interface (HUI) and connections to flexible manufacturing and factory management information systems, (FMS & MIS). All these functions must be accommodated in real-time and in a safe and reliable manner, the design and implementation of a computer control system to fulfil these requirements is a major task.

The above functions span many different levels of machine control, from factory management to feedback control of a single motor. Hierarchy can be used in constructing such a machine controller in the same manner as that used by FMS installations using CNC machines and robotics, described in section 2.1.

The requirement for real-time response from a number of processes operating at different levels in a control hierarchy will be supported by an appropriate hardware processor / software topology. In the following sections computer hardware and software technology is examined to identify the most suitable topology for an independent drive machine controller.

2.2.6.1 Computer hardware technology

The simplest hardware for a computer utilises a central processor unit (CPU) with associated random access memory (RAM) for dynamic data storage and read only memory (ROM) for static data/program storage. Basu [60] describes this single CPU system when executing a single sequential program, as a single instruction, single data stream, (SISD), Von-Neumann architecture. This hardware topology is suitable for executing a single real-time control loop such as the embedded controller of the BRU-500 brushless DC motor from Electrocraft [61].

To execute more than one control-loop in a single processor system requires that multitasking be used, sharing the available CPU time between software processes. Bennett [62] describes how the hardware supporting multitasking often utilises hardware interrupts and priority levels attached to individual processes to facilitate scheduling on a SISD architecture.

Multitasking is used widely as a solution to provide concurrent real-time processing to multiple software processes. However, the performance of a single processor system may be inadequate to support multiple, high sample rate, digital servo-control loops such as those required for the control of independent drive machines.

The alternative to a single processor system is to use multiple processors with inter processor communication, Basu [60] describes this as a multiple instruction, multiple data stream, (MIMD) architecture. The individual processors in the network execute discrete software processes. The set of processes on the network executing simultaneously providing concurrency. Multitasking may also be used within the processors in the network.

The processing capacity of the distributed processor network may be much greater than a single processor. However Carling [63] shows that linear increases in capacity are not always possible due to the additional overhead of inter processor communication.

Communication between concurrent software processes on different processors is achieved by message passing via a communication medium such as a dedicated high-speed inter processor link described by Basu [64]; shared memory associated with a common bus such as VME and MULTIBUS discussed in Conte [65]; or a broadcast network based on a standard such as Bitbus, MAP or Ethernet described by Dwyer [86]. Communication between concurrent software processes on the same processor uses shared data areas held in RAM.

Problems with inter-processor communications such as deadlocking and data corruption can be prevented using techniques such as monitors, semaphores, locking protocols and watchdog timers detailed by Dijkstra [67], Hoare [68] and Holding [69].

The MIMD architecture supports implementation of hierarchical control structures by appropriate partitioning of controller tasks to individual processors. The processors themselves often being physically partitioned according to level in the control hierarchy. For example the Molins maker computer controller described in section 2.1.4 is partitioned, as shown in figure 2.2, according to scan rate, with the lowest levels having the highest control-loop scan rates. This existing controller also demonstrates the feasibility of IDM control using distributed computer control systems constructed from existing processing technology.

The physical hardware structure not only supports a hierarchical control topology but also follows a natural division of the machine process in terms of time response. The lower level has high-speed response, processing large quantities of data for individual servo-loops at set sample rates. The higher levels have reduced data rates, with less arduous processing demands but the data is from a number of sources and is of greater significance - dealing with modes of machine operation and safety. Data rates increase down the hierarchy but the significance of the data reduces. This is the principle of increasing precision with decreasing intelligence introduced by Saridis [70] and is a useful heuristic when partitioning controller functions into software processes and hardware processors.

Computer hardware is often available as standard components suitable for integration into a proprietary computer architecture. A wide variety of computing elements are available for architectures such as the IBM personal computer [71], VME [72], MULTIBUS and Inmos link systems [73]. Puusaari [74] describes how these standard architectures

facilitate a modular approach to developing specialised computer controllers for machine systems.

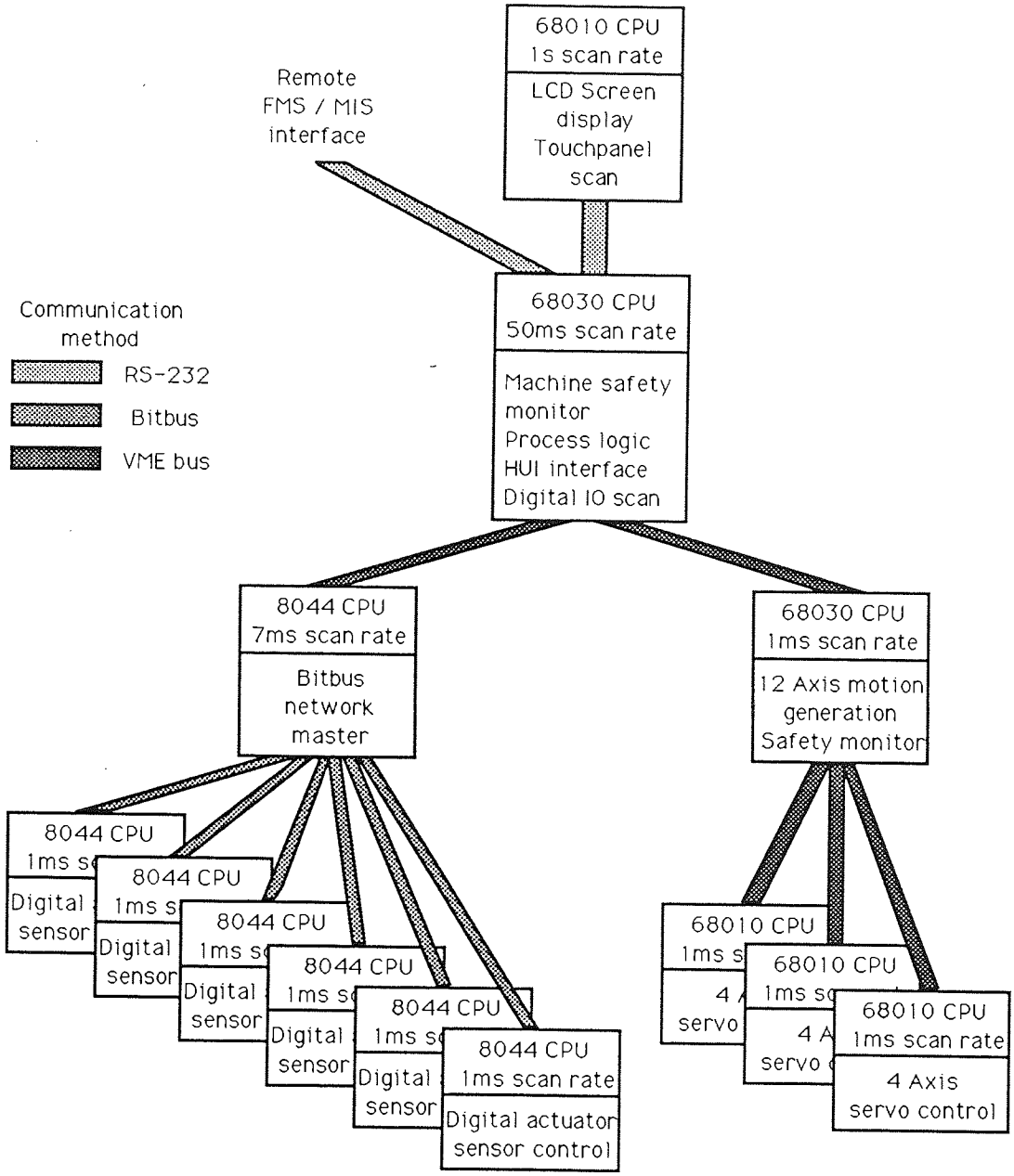


Figure 2.2. Molins maker hardware/software partitioning

2.2.6.2 Computer software technology

The computer software for IDM machines is the principle element in maximising the flexibility of the independent drive mechanisms. The computer programs embody the

required functions of the computer controller as software processes. The specification, validation and design of these software processes will be discussed in detail in section 2.3. To control the machine the software processes must execute and communicate in real-time on a suitable processor architecture. Various methods for achieving this multiprocessing are available.

A single processor at a specific moment in time will execute an instruction for a single software process. This one process may perform several controller functions to achieve a simple form of multiprocessing. A more refined solution is to utilise a special generic software process, termed a kernel or executive, described by Bennett [62], to provide scheduling and context-switching services for several user defined software processes. This form of multiprocessing, emulates simultaneous execution of several sequential processes by the use of time-multiplexing of the available processing power over the set of concurrent sequential processes. General system services to user processes such as input/output to CRT, keyboard and memory are also required. Miller [75] terms the system services and kernel program an operating system.

For IDM machine control the software processes must execute within time constraints to achieve real time control. Real-time operating systems, such as OS-9 [76][77], allow the scheduling of processes with special regard to real-time constraints. Ready [78] describes how a kernel supporting queues of active processes together with software based process priority allows pre-emptive scheduling to be performed under software control. Additional system services to construct watchdog timers, real-time clocks and process event signalling mechanisms eases the real-time programming burden. For safety-critical processes deterministic operation of the software is required in order that system performance be guaranteed. Deterministic scheduling of software processes is lacking in most real-time operating systems, including OS-9.

In a multitasking design each sequential software process may accomplish a separate controller function, facilitating software modularity. In a multiprocessor system physical concurrent operation of software processes is achieved. Again the necessary controller functions can be decomposed into individual software processes, however the processes are distributed across the hardware network of processors. Thus modularity is applied to both the hardware and software implementation.

Linkens [79] demonstrates how a multiprocessor, multitasking topology supports modular hardware and software reuse since each processing node may execute standard software

concurrently and each processor may use standard interfaces to both plant and other processors in the network.

In some programming environments inter process communication between CPUs and within a single CPU utilise a consistent programming notation. For example Occam, described by Hoare [80] and Pountain [81], supports message passing between concurrent processes using message passing via channels. After definition of channels as internal or external to the processor the language treats the communication mechanism in the same way. This allows consistent programming of both intra and extra processor communication, facilitating prototyping and modular development of control processes.

The physical partitioning of software processes allows supervisory tasks such as the human user interface, data logging and networking to flexible manufacturing systems, to be separated from axis control and clarifies the function of each node. Figure 2.2. shows the software process partitioning for a Molins maker machine. The higher levels of the hierarchy accommodates the needs for external command and report interfacing without loading the drive processing nodes unnecessarily.

The logical control hierarchy developed in the design phase of the computer controller will be mapped onto a physical implementation of concurrent software processes executing on a distributed network of processors. The combination of distributed processing for the axes of the machine together with centralised processing of command / status and monitoring functions results in a hierarchical, layered control architecture. This computer control structure maps closely to the functional requirements of a general machine. It also uses modular techniques to facilitate design and to reduce development and implementation costs.

2.2.6.3 Programming environments

The computer controller software will be created using a number of tools which form the software development environment. The controller software will then be executed together with additional system software in the execution environment on appropriate hardware.

For system design computer aided software engineering (CASE) tools may be employed. For example Easycase [82] supports system analysis and design by automating development and verification of methods such as Yourdon, SSADM and JSD.

Hierarchical decomposition of control systems defined using these methods result in software process designs ready for implementation.

The program development environment will include tools for creating source code from code-generators and editors, compilers for creating assembler code and linkers for creating executable code from assembler code. Libraries of software routines provide interfaces to system code such as real-time operating systems and mathematical packages.

The execution environment will support the developed programs when running. This environment may provide many system services, such as those provided by the Unix operating system described by Coffin [83] or it may be a minimal package providing compact high-speed operation such as the PLM51 executive from Intel [84].

An integrated development environment provides both development and execution environments in one package. Examples of these are the Ada Integrated Programming Support Environment (IPSE) described by Lyons [85] and the Transputer Development System (TDS) which supports the Transputer microprocessor and Occam programming language [73].

Software tools such as Grafcet for PLCs [43], allow more complete control system development for specific PLC systems. A graphical description of control processes being used to directly create PLC control programs. Currently these tools concentrate on sequence control with simple feedback control loops and are restricted to a single manufacturers PLC technology.

The languages and tools used in developing an IDM machine controllers must support modular embedded real-time control and will therefore need functionality to describe communicating, concurrent processes operating in real-time. They must also be useable with a computer controller systems analysis and development method. The development of an IDM controller is discussed in further detail in the next section.

2.3 Development of computer control systems

In order to provide a computer control system capable of utilising the flexibility of independent drive mechanisms a sequence of control system analysis, design and implementation phases, tailored to the needs of IDM machines, must be performed. A

framework for IDM controller development based on the waterfall model of the software life cycle, a well established model for computer system development described by Bollinger [85], is presented in section 2.3.1. This framework provides a structured development strategy for the successful specification, design and implementation of the computer control system. Aktas [86] describes how a structured development strategy is required to cope with the complexity inherent in an IDM control system.

To support the controller development, tools, notations and methods for requirements capture, validation, specification, system design and implementation are required. An appraisal of mature techniques is given in section 2.3.2. These techniques have been analysed to identify generic tools and notations appropriate for an IDM machine computer controller development. An IDM machine controller requires the specification, design and implementation of real-time, modular computer control embodying flexibility in function with safe and reliable operation.

2.3.1 Waterfall model of the computer development life cycle

The waterfall model of the development of a computer controller is partitioned into several phases. Starts [87] describes how each phase deals with a separate issue of the development process. Aktas [86] groups the phases into planning and construction stages.

The planning phases include the capture and specification of requirements for machine and controller functions, the design of the computer controller and the decomposition of the controller into hardware and software modules. The construction phases include software coding and the hardware build, system integration, commissioning and maintenance.

In the waterfall model the phases of development are sequentially ordered, but recursion to previous stages supports alterations to the computer controller development. Thus amendments and modifications can be accommodated throughout the development life cycle.

In practice a waterfall development approach assumes that the user's requirements for the controller will remain relatively fixed, what Birrell [88] terms 'prespecified computing'. This is necessary to minimise iterations through the phases since changes to higher level

phases require that all subsequent phases be reworked which Boehm [12] shows may involve considerable effort and cost .

Each phase of the development must have a clear start and a clear end. To provide this clarity deliverable items, usually documents, are produced as the output of a phase. These output deliverables are then used as inputs to the next development phase. The deliverable of each phase becomes a definitive statement on the controller development, as such it should be complete and correct to avoid the need for recursion through previous phases.

A waterfall development life cycle for an independent drive machine computer control system can be seen in figure 2.3. A brief discussion of the components of the figure elaborates each development phase and describes the deliverable documents which it produces.

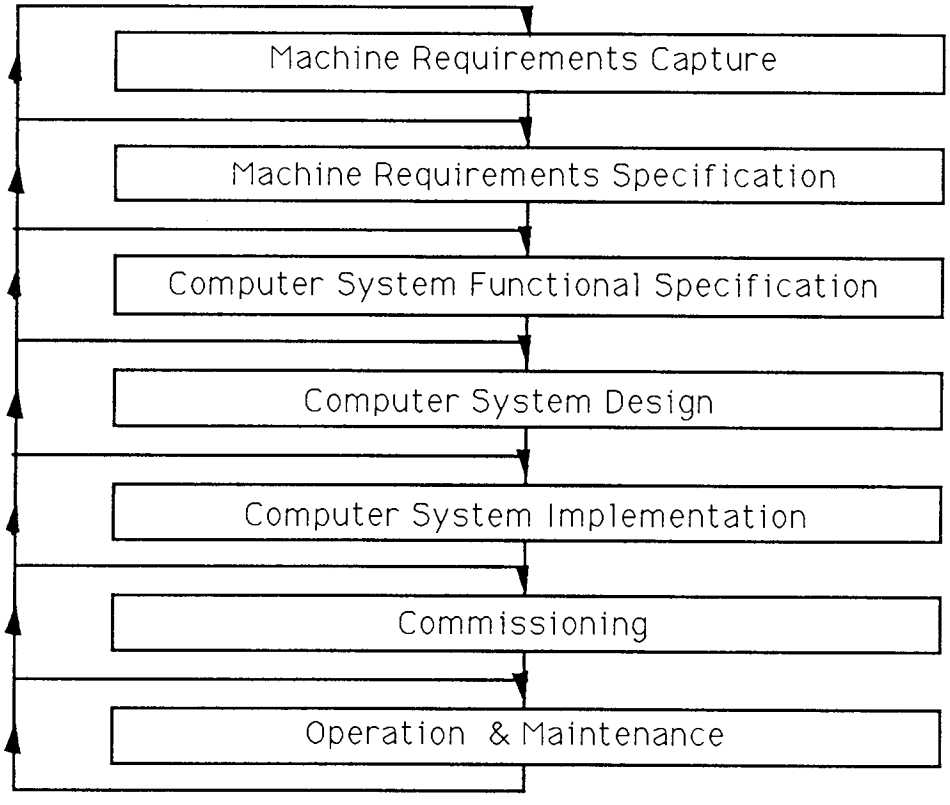


Figure 2.3. A waterfall development life cycle for an IDM machine computer controller

2.3.1.1 Machine requirements capture phase

This phase gathers the developer's principal requirements for the machine. These requirements are sufficient to describe the manufacture of the user's desired product. They will include both functional and non-functional requirements drawn from mechanical, electrical and control fields. McMenamin and Palmer [89] describes how functional requirements relate to operation of the machine and controller, non-functional requirements to methods of development.

The initial requirements will be defined by the user's desired product. These product specifications can be used as a basis for acceptance tests for the manufacturing operation. Anderson [90] describes how the presence of verification tests are necessary to allow quality assurance during development.

The operations required to convert raw material into the product and the interrelationships between operations are then used to define the format of the manufacturing process. Physical components of the process including drives, mechanisms and sensors can be specified to accomplish the various manufacturing operations. Fenny [14] describes how these components may then be grouped into individual manufacturing modules. Functional requirements of the components such as motion and synchronisation requirements of drives can be captured.

The deliverables of this phase will include documents to describe the physical layout of the machine and the modules which will achieve the overall manufacturing operation. Details of each individual module will include motion and synchronisation requirements including flexibility. To facilitate quality assurance process constraints will be described. To ensure machine safety motion, synchronisation and physical drive constraints will be detailed.

Goldsack [91] describes how textual descriptions and graphical models are often used as notations for requirements capture. These capture methods can be unstructured and prone to omission and contradiction. As a result validation and verification of machine requirements is required before a specification document can be produced. A rigorous notation which support analysis for validation of motion and synchronisation of IDMs is proposed later in this Chapter.

2.3.1.2 Machine requirements specification phase

The machine requirements data obtained during the capture phase is structured into models and tables which specify the machine requirements. This data is then correlated, validated and verified to ensure it is correct, complete, feasible and unambiguous. Errors detected necessitate a return to the capture phase to resolve the problem.

Validation, the confirmation of the requirements as suitable for the purpose intended, ensures the captured requirements are appropriate, this involves consultation with users to confirm that the manufacturing process results in the correct product. To ensure the captured requirements are correct and complete, verification, described by McLeod [92] as the process of proving the correctness of qualities in the requirements, is necessary. For requirement verification Goldsack [91] describes how modelling can be used since particular qualities of the requirements can be exercised and assessed for correctness if an appropriate model is available. A model of IDMs as cooperating state-machines is proposed to support validation and verification. Validation and verification are required throughout the controller life cycle as the requirements are embodied in the developing controller.

The machine requirements specification is the deliverable of the phase. This document should contain all pertinent requirements in a format usable by the system developer, textual descriptions, tables and graphical models being used. The machine requirements specification should also be understandable by the user to permit 'signing-off' of the specifications. The approved document becomes the principle definition of machine function.

2.3.1.3 Computer control system functional specification phase

This phase is the first development phase aimed specifically at the machine computer controller. The aim of the phase is to define what is to be produced, a functional specification for the controller, and to define how it is to be produced by writing a project plan and quality management plan.

The project plan details the steps needed to deliver the product on time and within budget. The quality management plan is a statement of technical and managerial steps to ensure quality in the controller.

The functional specification forms the basis of controller development. The phase uses machine requirements relevant to the controller to define functional specifications for the controller. These specifications can then be used to construct control system models, based on state-machine representations of IDM operation. The models are used to assess qualities of the functional specification, such as the safety of inter-IDM synchronisation. This ensures that feasible, correct and safe specifications are produced from machine requirements. To ensure that the final controller implementation can be verified against the functional specification acceptance test specifications for the controller implementation are required.

The deliverables of the phase are a controller functional specification verified against machine requirements, project and quality management plans and acceptance test specifications.

2.3.1.4 Computer control system design phase

This phase takes the validated controller functional specification and produces a computer control system design which satisfies it.

A computer system design method is used to produce a controller design, the output of the phase. Methods of design are detailed in sections 2.3.3.1 - 2.3.3.5. Different design methods produce different output documents but the format and contents of the these documents will have been specified in the quality management plan.

The system design provides a detailed breakdown of work in subsequent phases. Models to validate design features against functional requirements are required, these may include load and throughput models, timings for critical computations and refinements to models in the system definition phase. This new detail may require a review of project and quality management plans.

2.3.1.5 Control system implementation phase

This phase takes the validated and verified system design as an input and produces a system ready for commissioning as it's major output deliverable. It should also provide user documentation and acceptance test descriptions.

The controller hardware implementation involves procuring processing components, integrating components into processing nodes, configuring and testing processing nodes and integrating all nodes to produce the final computer hardware system.

A number of intermediate stages within the phase will be required for software development. Birrell [88] describes how these stages include transformation of the controller design into detailed design of software modules, coding of modules, testing of modules, integrating modules into subsystems and integrating subsystems to produce the final software controller.

Software and hardware development can proceed concurrently. Testing of functional parts of the controller may be possible allowing controller verification. The phase deliverable of a computer control system ready for commissioning is produced when all software and hardware development is complete.

2.3.1.6 Control system commissioning

The computer system ready for commissioning is connected to the IDM machine. System acceptance tests based on the documents produced in the functional specification and system implementation phase are performed. The models of machine requirements and computer functional specifications can be used to correlate observed machine behaviour with that predicted by the models. Controller models will identify particular states of the machine in which hazards exists, the machine can be forced into these states to verify that the controller can deal with the hazard. Detected errors in the controller are removed by recursion through stages of the waterfall process. The deliverable of the phase is a machine system acceptable to the user with appropriate user documentation and training schedules.

2.3.1.7 Operation and maintenance

This phase deals with the machine and controller in service. The controller may be modified to remove errors and according to changing user requirements. Boehm [12] describes how modification and support of existing software systems can equate to 50% of the cost of software projects. To minimise the cost of maintenance Birrel [88] identified two major design heuristics: The first is to build longevity into the system through good design and implementation; The second is by managing post-

commissioning development using configuration and change control to preserve integrity of the system and hence delay the decay introduced by ad-hoc changes to the system.

The maintenance phase can therefore require much modification of the developed controller. A separate life cycle to deal with these modifications is proposed by Birrell [88] but is outside the scope of this thesis.

2.3.2 Application of the waterfall method

An industrial project may be forced to use the waterfall development process in a pragmatic fashion, particularly when creating prototype machine systems, when using unfamiliar technology and when user uncertainties as to machine requirement are present, since a 'prespecified' or fully understood machine specification cannot be created. In addition to incomplete specifications many machine developments encounter changing requirements during machine construction when development team members experience problems in construction of machine elements which result in changes to machine specification and design. However maintaining deliverables of development phases imposes an important structure on the controller development and provides quantified evidence of progress which is vital for correct management of the project.

The compromise between producing many versions of documents and relying on outdated documentation is to group changes into revisions which are then used in updating the deliverable documents. Automated tools for managing documentation and version control such as the Unix software management system [93] and CASE tools such as Easycase [82] can be used in reducing the effort required to keep documentation up to date.

2.3.3 Development methods

In section 2.3.1 the framework for system development based on phases subdivided into planning and construction stages was introduced. The planning stage requires methods for capturing requirements, writing specifications, describing controller designs and generating models. The construction stages require methods for modular design decomposition, generation of hardware and software modules, and facilities for coding, testing and integrating these modules.

Sections 2.3.3.1 to 2.3.3.5 describe mature approaches used in industry to provide tools and methods for controller development. The methods described have been chosen because of their application across the system development life cycle and their use of generic approaches to development issues.

Not all methods cover the entire development life cycle but some notations can be used in several development phases. Standardisation on tools and notations through the life cycle is proposed as an aid to productivity by allowing tool reuse and reducing the training time needed for engineers to become competent with the tools.

The five methods discussed are SAFRA, JSD, structured development using DFDs, FSMs, Petri-nets and prototyping. Notations and approaches of particular relevance to the IDM controller will be highlighted. A comparison of the techniques with respect to suitability for IDM machine development is presented in section 2.4.

2.3.3.1 SAFRA - semi-automated functional requirements analysis

SAFRA defines a set of tools for use in large real-time computer system developments such as the British Aerospace experimental aircraft program described by Starts [87]. The tools cover the development life cycle from requirements capture to maintenance.

There are three components used in the SAFRA toolset: Controlled Requirements Specification (CORE) a method for requirements specification; Modular Approach to Software Code Operation and Test (MASCOT) a set of facilities for real-time program design; and Perspective, a real-time Pascal programming language supplied with a MASCOT kernel for implementation code and testing.

Looney [94] describes CORE as a method for determining the functional requirements of a system and drawing up a functional specification. The technique provides requirements capture from a number of viewpoints.

CORE analysis iteratively decomposes these viewpoints into levels, this results in a top-down hierarchic expansion of the viewpoints. At a particular level functional processes and data flows for a viewpoint are described. Tightly coupled flows of data through a viewpoint's processes are identified as 'threads', providing a temporal sequencing of process operations on data. An experienced analyst identifies the relevant viewpoints at

each level of decomposition. CORE defines how the analyst should propose, define and 'confirm' the viewpoints.

The viewpoint information at each level is presented in a precise diagrammatic form which can then be checked for consistency and completeness across the whole requirement. Thus CORE produces requirements which are unambiguous, consistent and complete. The CORE diagrammatic notation is accessible to users for validating requirements. However CORE diagrams may need to be supported by a computer-based tool, such as the CORE workstation tool developed by British Aerospace [87]. In a SAFRA development the CORE procedure and diagrams are used as the input for the design stage.

CORE provides a method and notation for capturing and describing high quality requirements. It has a visible and maintainable structure which promotes improved quality through a disciplined design technique. However requirements capture requires an experienced analyst to obtain information from many interested parties.

MASCOT [95][96] is a set of facilities for real-time programming, it facilitates the design of real-time computer systems and provides a route to implementing the design using an executable MASCOT machine. It consists of three elements. A generic formalism for expressing the software structure of a multi-tasking or real-time system. A disciplined approach for design, implementation and testing which uses a concept of modularity in real-time systems and supports system reliability by controlled access to data. An interface to support the implementation and testing methodologies is provided by a small kernel which provides sequencing, concurrency control and data access protocols.

The MASCOT formalism is based on hierarchical design, a graphical notation and an equivalent textual representation. The graphical design format or Activity Channel Pool (ACP) diagram places emphasis on data flow and concurrency. Starts [87] details the ACP notation which comprises data flow through concurrent activities and intercommunication data areas, IDAs.

IDAs are passive but encapsulate the interactions between activities. They maintain integrity of data and hence inter-activity interfaces by providing data access mechanisms which follow Parnas's principle of information hiding [97]. The MASCOT kernel [96] hides the implementation details of concurrency control and data access from the concurrent user processes.

Development of a MASCOT design requires that the ACP diagram be implemented in a programming language with a suitable MASCOT kernel. CORAL 66 [98], Pascal [99] and Ada [85] are used in MASCOT implementations. The MASCOT kernel uses a model of synchronisation between concurrent sequential processes which supports mutual exclusion of competing processes (races) and the cross-stimulation of cooperating processes. Interrupts, co-operative and pre-emptive scheduling of processes are supported. A MASCOT design may execute on multiple processors with single or multiple MASCOT kernels.

MASCOT is a mature method for computer program design and with a suitable MASCOT machine, implementation. MASCOT development encourages modular design and is aimed specifically at real-time embedded computer applications. However MASCOT requires a kernel for implementation and support languages are limited. The design method concentrates on data-flow, control flow and synchronisation being incorporated at the detailed design stage. There is no concept of time or synchronisation within an ACP activity. ACP diagrams tend to be large but hierarchical decomposition can be used in the latest version of the MASCOT standard.

In the SAFRA toolset a MASCOT design is implemented using PERSPECTIVE. PERSPECTIVE is a software development package, based on Pascal with real-time extensions which embodies calls to a MASCOT kernel. The real-time Pascal language used is slow for some real-time applications. The ERA project, described by Starts [87], resorting to assembly coding to achieve required performance.

SAFRA is a comprehensive and powerful development system dealing with multi-tasking and real-time constraints. It provides tools and methods to support all of the software life cycle with extensions particularly relevant to real-time control.

2.3.3.2 JSD (Jackson System Design)

JSD is a structured design method for computer software. Smith [100] describes how it covers the software system life-cycle from initial definition of software requirements, through design, to implementation and maintenance. It is a general purpose method which can be applied to embedded real-time applications.

JSD has two features which separate it from other methods. The first is the use of a 'real-world' model as the primary development step. This model is a definition of the aspects

of the real world which are going to be implemented in the computer system. The second feature is that program code is generated by applying standard techniques and transforms to the system specification [101]. This ensures the design is faithfully implemented in code minimising omission, and irrelevant addition.

A JSD software specification comprises a network of simultaneously active processes. It uses the Communicating Sequential Processes (CSP) concept of Hoare [68] which is also used by MASCOT. JSD processes communicate using a co-routine model.

The JSD technique represents the waterfall life cycle phases of requirement specification, design and implementation with three development phases. These being the 'model phase' in which the real world processes upon which the system functions are to be based are selected and defined, the 'function phase' in which the functions of the system are determined and inserted in the model producing a logical design, and the 'implementation phase' in which the processes and their data are fitted on the available processors.

The modelling and function steps are supported by two types of diagrammatic notation. A system specification diagram (SSD) defines the whole system as a set of logical processes, communicating with each other via data transfer paths. Whilst structure diagrams (SD), one for each logical process, define the internal structure of the processes. JSD modelling also makes use of entities and actions. An entity is an object in the real-world which participates in a time ordered set of actions. An action is a point event in time performed on or by an entity. The JSD model, function and implementation phases are supported by a progression through several development stages.

The method does not include any type of decomposition or hierarchy of design. The only description of the processes and communication paths of the complete system being a single diagram which can be very complex. However system complexity can be mitigated by computer based tools, tools for diagramming and code generation such as Speedbuilder and PDF [102] are available to support the technique. JSD does not consider verification testing of the developing system and no method is described to abstract design information such as that required for safety analysis.

The real-world modelling proposed in JSD supports capture of requirements outside the computer system. Capturing requirements in the initial JSD model is seen as a difficult but key task by Smith [100]. However Birrell [88] states that industrial projects often use a separate requirements capture method before using JSD for software design and implementation.

For IDM machine controller design the JSD method has disadvantages. System requirements are not easily captured and the initial design is unwieldy and difficult to validate. Hierarchy is not used in the method, however hierarchy can be used as a means of introducing and managing modularity in a controller. In a JSD implementation automated generation of code is a benefit to incorporating design functions but constraints on hardware and software topology may not be supported by this automation.

2.3.3.3 Structured analysis and data flow

Structured systems analysis is a methodology defined by DeMarco [103] and promoted by Yourdon [104]. The method has been refined to support real-time issues and control flow by McMenamin & Palmer [89] and Ward & Mellor [105]. The method can be used to produce a structured functional specification from user requirements and develop a software design suitable for coding.

Structured systems analysis uses data flow and process specification methods to create a model of the required computer system. Data flow diagrams (DFDs) are used to show the movement of data through the system by annotating system activities, interactions between activities, interactions between the system and it's environment, and data storage (essential memory). Process specification methods define the operation of activities using minispecifications or pseudo-code, and detail the composition of interactions and essential memory using data-dictionaries.

Essential systems analysis, detailed by McMenamin and Palmer [89], uses the tools of structured analysis as a basis for a technique which attempts to create an implementation independent model of the requirements of the system which can be transformed into a physical implementation. There are three stages to the process.

The first stage defines the essence of the system and involves capturing the computer system requirements and producing a functional specification model. This involves the determination of the purpose of the system, the events to which it must respond, the fundamental activities, the information both from the current and past events, or essential memory, that the system must store in order to carry out it's responses, and any custodial activities required to establish and maintain essential memory.

The second stage involves selecting an incarnation of the essence by producing a system design. This involves transforming the functional specification embodied in the essential model into a physical design represented by an incarnation model. This incarnation model implements the system's essential activities and memory by defining physical processors, 'infrastructure' processes for inter-activity communication and 'administration' processes which provide data-integrity.

The third and final stage is the construction of the physical system. This involves the hierarchical decomposition of the incarnation model into software using a method based on the data-flow notation such as Yourdon's structured design method [104][105] or Myer's composite design method [106]

The essential systems analysis technique is biased towards requirements capture and initial specification of 'true requirements'. Automation of the method is limited, although computer based tools for DFD support and general database packages, such as Easycase [82] and Dbase [107] can be used. The method does not address real-time issues of timing and sequencing of activities. However extensions to the technique have been developed by Ward and Mellor [105] which use finite state machines to coordinate and control the operation of the DFD activities.

The essential modelling technique provides a method for capture of functional requirements which can be easily expressed via a graphic notation and a more detailed textual description. The DFD models produced are easily assimilated by users to facilitate validation of requirements although no procedure for validation is included with the methodology. The inclusion of control-flow description by Ward and Mellor allows the method to be used for real-time control systems by providing methods of describing sequence between cooperating activities and the definition of timing constraints.

2.3.3.4 Finite state machines and Petri-nets

A finite state machine (FSM) provides a method for describing deterministic operation of a virtual machine in terms of inputs, outputs and states. FSMs can be used in all stages of the system development life cycle.

The FSM paradigm defines a virtual machine or automata. A FSM is a single process system. It describes the operation of a virtual machine by defining a set of states (S), a set of inputs (I), a set of outputs (O), a next-state function (f) and an output function (g)

[Birrell 85]. The next-state function given an input and a state gives the new state which the FSM moves to. The output function given an input and a state gives the output that the FSM produces. Avenur [108] terms this a Mealy model FSM since outputs are associated with states and inputs.

FSMs are based on the automata mathematical concept and this mathematical base allows analysis of FSMs if an appropriate model, such as Petri-nets described by Agerwala [109] and Carpenter [110], is used. Thus specification of systems using FSMs allows checking of the correctness of the specification.

The principle role of FSMs is to define or implement the control structure of a system. FSMs are applicable in any system where events trigger each other in a well defined way producing defined sequences of actions.

FSMs can be described by graphical notations of inputs, outputs and states. The use of graphical notations, such as state transition diagrams [105] or Grafcet [43], to describe the states of a system, the transition between states and the interaction between state machines provides a powerful and easily assimilated development method for the designer.

In the functional specification phase FSMs can be used to describe overall system operation before applying another methodology for a more detailed specification. A combination of system activities detailed by DFDs and scheduled by FSMs expressed as state transition diagrams provides a powerful development tool for detailing the operation of real-time systems in the design phase. The method has CASE support in industry, such as the CARDtools of Ready Systems [111]. However the technique does not explicitly perform function abstraction or validation of design.

In the system implementation phase FSMs can easily be coded in a sequential programming language. A single FSM only models a single machine process, in order to describe multiple cooperating concurrent processes, multiple communicating FSMs are required.

Petri-nets, detailed by Peterson [112], are designed to model systems with interacting concurrent components. They are therefore appropriate for multi-process systems since parallelism and concurrency are inherent in the model. A restricted form of Petri-net can model communicating FSMs. Petri-nets have both a graphical and mathematical form, and verification and validation of Petri-net models can be achieved by analysis.

Chang [113] describes how Petri-nets can be used as a tool for functional specification and design, while Carpenter [110] demonstrates their usefulness for simulation modelling. Where a Petri-net simulation embodies requirements in the form of specifications, as described by Murata [114], then a method of dynamically executing a Petri-net results in an executable specification. This form of specification is amenable to verification by Petri-net analysis of characteristics such as completeness, consistency and reachability [88]. Analysis of Petri-nets will be covered in more detail in section 2.4.4.

Petri-nets are also useful as an auxiliary analysis tool, providing analysis lacking in other methods. For example structured analysis may be used to investigate or propose a system, the resulting proposals then being modelled and analysed in the form of a Petri-net. The Petri-net model does not capture the requirements of the control system but provides a means of structuring requirements to form specification models.

The major advantage of Petri-nets is the ability to model a system of concurrent processes in an easily assimilated graphical notation while retaining the capability for mathematical analysis of the model. This analysis facility is lacking in other techniques such as MASCOT or JSD. However Petri-nets principally model control-flow and additional tools are required to describe data-flow and data-structure.

2.3.3.5 Prototypes

Birrell [88] describes how the development of prototypes is not a complete design method. Accordingly prototyping is best used in conjunction with a more complete development method. Prototypes are useful in several phases of the development life cycle to aid design by experimentation.

Prototypes using implementation hardware allows the bottom-up design of interfaces and procedures for real-time operation. High performance may only be possible using certain hardware / software topologies. A prototype can identify these constraints at an early stage to allow the development process to take account of the design constraints and avoid iteration through the development phases of the waterfall model.

The use of prototypes as executable models of requirement specifications, described by Wang [115], facilitates validation. Exercising the prototype can identify errors and omissions in the specifications, allowing recursion to the capture phase to remove the

problem. These prototypes may also be presented to the user since a demonstration of the prototype allows the user to identify incorrect or incomplete functionality, operation may then be corrected by changing the requirements. Prototype validation identifies problems earlier in the development life cycle and avoids costly recursion and redesign in later phases.

To be worthwhile a prototype must be quick to produce and evaluate, Goldsack [91] describes how this requires a development environment which can support definition of the required prototype functionality whilst also allowing experimentation by execution. For an independent drive machine the pertinent functions are real-time operation, multi-tasking and the ability to build distributed control systems. Prototype software will be constructed using a programming language, this language may incorporate the concurrent, real-time facilities required or may rely on a real-time operating system to provide the functions.

Programming languages which incorporate real-time and concurrent functions include Ada [85] and Occam [81]. Speed of creation of prototypes can be improved if the programming language is based on the same paradigm as the controller design and development tools, since a simple route from development model to prototype is available. For example Petri-nets and Occam use similar inter-process rendezvous mechanisms and Draper [116][117] describes how Petri-net model may be transformed into an Occam program of communicating processes, retaining much of the Petri-net model structure.

2.4 Computer controller development for IDM machines

This section discusses the development requirements for an IDM machine controller. A comparison of the development techniques presented in sections 2.3.3.1-2.3.3.5 with reference to the IDM requirements is presented to identify appropriate development methods and tools. Finally a development method for IDM machines is proposed.

2.4.1 IDM machine controller development requirements

An IDM machine controller is a relatively small computer development, an IDM computer controller development team typically comprising less than five engineers, the chosen development method and associated tools should be scaled accordingly.

The machine requirements and functional specification for an IDM machine are subject to change, methods which can identify development problems or help settle design decisions are beneficial. An implementation of the controller may be constrained by available technology, identification and accommodation of these implementation constraints should be taken as early as possible in the development life cycle.

The creation of an IDM machine requires contributions from many fields of engineering and the development of the computer controller will necessarily take place in an inter-disciplinary environment. Easily understood notations are required for communication of specifications and designs between development engineers. To communicate captured machine requirements an intuitive notation, preferably graphical, which can be understood by non-specialists but has the power to describe all necessary functionality is required.

The graphical capture notation should be amenable to analysis by prototyping to demonstrate functionality and by other rigorous methods such as Petri-net modelling to demonstrate safety and correctness.

The capture notation should be capable of being transformed into a requirements specification. Direct or mechanised transforms are preferable to minimise transcription errors. A functional specification for the computer controller should then be developed from this machine requirement specification.

The design method which provides database and process descriptions suitable for implementation from the functional specifications should allow verification of requirements whilst also supporting modular, real-time design.

Implementation will require transformation to an industry standard language, operating system and hardware. An IDM machine developer is likely to use available hardware and software where possible to minimise development time and development risk associated with new technology.

Maintenance and support have been identified as major components of computer controller costs. The controller design should therefore be developed to allow upgrades and addition by using modular design techniques for both hardware and software. Maintenance can be eased by building the machine as a series of independent modules, the components of modules being easily replaced and where possible standardised.

2.4.2 Comparison of development techniques

With reference to the IDM development requirements given in section 2.4.1 an analysis of the development techniques discussed in section 2.3.3 follows to identify an appropriate method for the development of an IDM machine controller .

2.4.2.1 IDM machine requirements capture and specification

None of the development methods discussed explicitly describe capture of requirements of particular relevance to an IDM machine such as requirements for motion, flexibility and synchronisation. This is in part due to the general nature of the methods but also due to the bias of the methods towards capture of software requirements.

A machine requirements capture and specification method specific to IDM machines is necessary. However generic concepts from the described methods may be used in the IDM requirements capture method. These include different viewpoints for requirements capture and hierarchical structure of requirements from CORE. Viewpoints allow requirements to be partitioned into areas of particular interest such as motion, synchronisation, modes of machine operation, and the user interface. Hierarchy can be used within these viewpoints to elaborate the machine requirements for different components describing viewpoint processes by text, database and diagram. Machine context diagrams using the DFD notation from structured analysis provide a method for describing relationships between machine components, particularly with respect to product operations, and manufacturing modules and their components. The sequence of operations can be described by state transition diagrams from structured analysis. Modelling and analysis of these requirements can use Petri-nets.

The use of these notations and viewpoints attempts to cover all pertinent requirements whilst defining standard notations suitable for discussion, validation and verification. The inclusion of Petri-nets for requirements analysis from the initial design phase is seen as a positive step to providing demonstrable system safety with increased controller quality. In addition the requirements will be traceable into the computer controller implementation for further quality assurance. A disadvantage of using mixed methods is the possibility of inconsistency between requirement notations leading to ambiguity by

omission or contradictory requirements. Therefore the analysis of the set requirements for correctness is of great importance.

2.4.2.2 IDM machine controller functional specification

The software development methods described all support computer functional specification of requirements. This specification is required as a deliverable document which forms the foundation of all future development. Figure 2.4 shows a comparison between specification techniques. Figure 2.5 shows the principle modelling capabilities of the methods discussed.

Technique	Method	Notation	Analysis method
Structured analysis	yes	yes	no
CORE	yes	yes	no
JSD	yes	yes	no
FSM	some	yes	yes
Petri nets	some	yes	yes

Figure 2.4. Functional specification techniques

Technique	Control flow	Data flow	Data structure
Structured analysis	no	yes	yes
CORE	yes	yes	no
JSD	yes	yes	yes
FSM	yes	no	no
Petri nets	yes	no	no

Figure 2.5. Principle modelling capabilities

JSD supports data structure, control and data flow modelling, however functional specification using this method is inappropriate due to limitations in the technique for requirements capture and when creating a system design from the real-world model. CORE can express data and control flow, but Looney [94] describes how analysis of requirements is still immature. Structured analysis methods provide good facilities for describing data flow and data structure, and extensions to the technique proposed by Ward and Mellor support real-time operation and provision of a state-transition diagram control flow notation. However this control flow notation is not directly amenable to analysis.

The IDM machine controller development requirement to allow analysis of the functional specification, identified in section 2.4.1 and the presence of deterministic operation of multiple IDMs promotes the use of FSMs for functional specification. An appropriate

Petri-net representation could be used to model these FSMs abstracting control information and facilitating analysis of the specifications. Flow control is insufficient to cover the required IDM machine controller functionality and modelling of data-flow and data-structure is also required, structured analysis methods using DFDs and associated databases are proposed for this purpose.

2.4.2.3 IDM machine controller system design

Given the choice of Petri-nets and DFDs as a basis for functional specification a system design method which can use these notations is required. The method must demonstrably incorporate the required functionality whilst also being suitable for implementation using appropriate languages and hardware. A comparison of the development methods for system design capabilities is given in figure 2.6.

Technique	Method	Notation	Analytic method
Structured design	yes	yes	no
MASCOT	yes	yes	no
JSD	yes	yes	no
FSM	some	yes	yes
Petri nets	some	yes	yes

Figure 2.6. System design techniques

Technique	Control flow	Data flow	Data structure
Structured design	yes	yes	yes
MASCOT	yes	yes	no
JSD	yes	yes	yes
FSM	yes	no	no
Petri nets	yes	no	no

Figure 2.7. Principle concerns of techniques

Real-time systems design, described by Ward and Mellor [105], using hierarchical decomposition of DFDs created in the functional specification stage is proposed as the principle design method. This approach retains consistent notations and supports close transformations from functional specification to design. However the method has no provision for analysis of the design.

Petri-nets provide a rigorous notation and analysis method but do not have an approved design method. A Petri-net modelling method based on state-machines and partitioning

of controller function is proposed which will support the structured design method. Analysis of these nets may be used to verify the specifications in the design.

Prototyping has been identified as a method for demonstrating functionality and this technique will be used during controller specification and design. To prototype cooperating FSM functions, real-time, concurrent processing is required, and to support this the TDS development system for Occam and the Transputer is proposed. This is due to the similarity between the Petri-net and Occam process models and the low cost of Transputer systems compared to other platforms such as Workstation support of an Ada IPSE, or an industrial development environment such as 'C' & OS-9 on VME hardware.

2.4.2.4 IDM machine controller system implementation

An implementation of an IDM machine controller will be based on the software / hardware design document. In industry, standard software tools and hardware will be used in a machine development to ensure stability of products by continuing hardware and software support. Molins together with many other manufacturers use 'C' where possible as a standard programming language. The language 'C' is widely used for industrial control and together with an appropriate real-time operating system can provide the processing functionality required for an IDM machine controller.

Research into IDM controller implementation is not constrained by the commercial pressures indicated above. An implementation of a design should map closely to the design in order that no functionality be lost or implemented incorrectly. The preferred method of generating executable code from IDM controller designs represented by DFDs and Petri-nets would require as few transformations as possible resulting in a close mapping from design to implementation. Many state-machines are anticipated in the design and a language which can support real-time communicating sequential processes without relying on operating system facilities would support this close mapping. If the language can also support multiprocessing on a distributed processor network in an efficient and concise manner then the hardware implementation would also be simplified.

Occam and Ada have been introduced as real-time, concurrent programming languages. Occam is based on the communicating sequential processes paradigm of Hoare [68], supporting the creation of communicating state-machines. The Transputer microprocessor which was specifically designed as a distributed processing element supports Occam definitions of multiple processes on multiple processors efficiently. The

development method proposed will therefore use Occam as a language for implementing designs represented by DFDs and Petri-nets and will execute code on Transputer based processing.

2.4.2.5 Summary of the proposed computer controller development method for IDM machines

To support an IDM machine computer controller development the following set of tools and techniques is proposed.

A graphical notation and associated database will be used for mechanical/process requirements capture representing viewpoints for the machine. Context of the machine controller and external interfaces will be defined using DFDs and associated databases. Petri-nets will be used to model FSM functionality and to facilitate analysis to validate the requirements. The validated machine requirements will be used to create a machine requirements specification document using textual descriptions, diagrams & databases, DFDs and Petri-net notations.

A functional specification for the computer controller based on the machine requirements specification rigorously defined using validated Petri-net models and DFDs will be produced. Refinements will be added to incorporate implementation constraints identified by prototypes of controller functions. Models of critical functions will be abstracted using Petri-nets, analysis of these models allowing validation and providing expected performance of the controller for acceptance tests in later phases. Constraints tables for product and process quality will also be included to support controller verification and quality assurance.

A system design method using finite state machines represented by Petri-nets and data flow diagrams is proposed. It will use finite state machines to specify control flow and sequence over the various functions, transforms and processes that compose the software requirement specification, which are structured hierarchically in data-flow diagrams. FSMs will also be used to elaborate the function of a DFD activity. Executable prototypes in Occam developed from a close transformation of the Petri-net design models will be used to identify errors and omissions in the design and to identify implementation constraints.

An implementation will use the Petri-net FSM and DFD based design to produce code. Transforms from Petri-nets to Occam and structures to implement concurrent, distributed software using Occam and the Transputer will be detailed.

Since DFDs and Petri-nets will be used extensively to describe requirements and detail designs, a description of DFD notation, Petri-net notation and analysis will now be given.

2.4.3 Data Flow Diagrams

Data flow diagrams, detailed by McMenamin [89] allow the description of a system at a logical level without considering the physical environment in which data flows or is stored. The functional processes of the system are detailed in a DFD and the data used and produced by these processes specified. Memory stores of data can be modelled. The boundary of the modelled system to the external system can also be described. The diagrams can be hierarchically structured and therefore support top-down design methods.

DeMarco introduced the DFD notation [103]. To demonstrate the modelling capability of DFDs and the symbols used in the diagrams consider a process which controls the motion of an independent drive axis. The process is required to perform a predefined motion profile. The motion data is to be output to the drive controller (in velocity mode) via a DAC at a suitable sample rate. There is discrete position and velocity feedback available from the drive to support closed-loop control. Figure 2.8 shows a DFD for the system.

Rectangular termination boxes represent an external entity which is outside the modelled system and they define the boundary between modelled and external system. The drive electronics, which are a source and sink of data external to the system, are identified by such a termination box.

Data flows in the modelled system are denoted by directed arcs, the data being represented labelling the arc. Solid lines indicate discrete data flows of information whilst solid lines with two arrows indicate continuous data flows of information. Dashed lines indicate control flows which prompts activity. In this example the motion is started by the control flow prompt 'enable'.

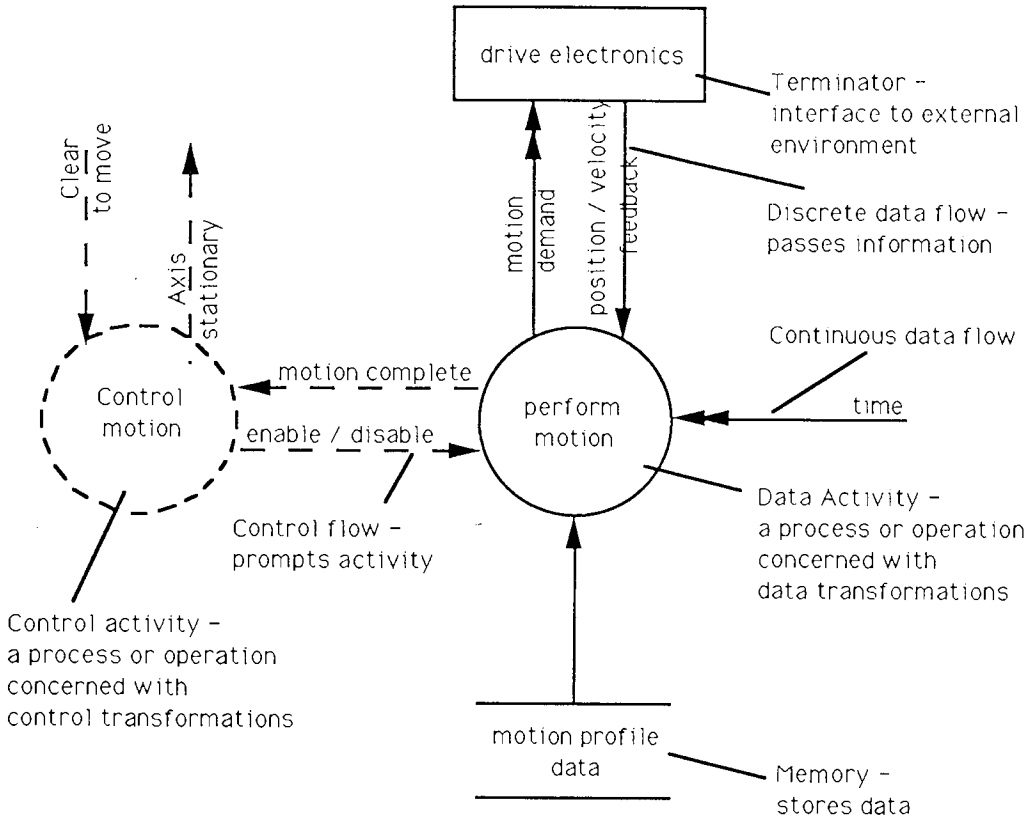


Figure 2.8. Components of a Data Flow Diagram

The solid circle represents an automated or manual activity or process. It transforms input data flows into output data flows. A brief descriptive statement indicates the function of the activity, i.e. 'perform motion'. Two parallel lines denote storage of information or objects, irrespective of the physical storage medium. They represent memory of the system, i.e. 'motion profile data'. The names that are used for data flows, description of activities, source/sink and storage are defined in a database for the system under development.

The dashed circle represents a control activity which transforms input control flows into output control flows, i.e. 'control motion'. Typically the control activity denotes the presence of a state-machine which controls the data transform activities, figure 2.9. shows a state-machine for figure 2.8.

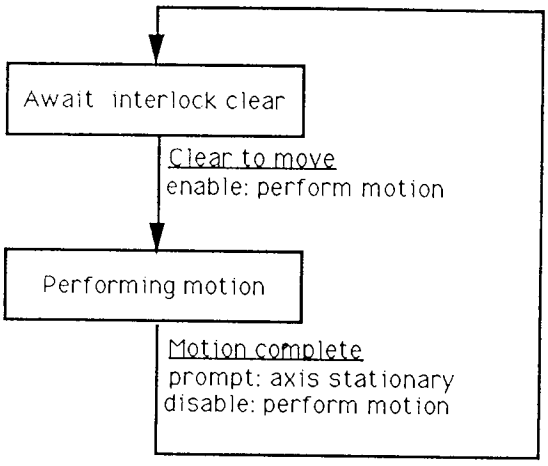


Figure 2.9. State machine expansion of 'control motion' control activity

The boxes represent states of the state machine. Conditions for leaving a state are denoted by an underlined control flow input and actions on leaving a state denoted by labelling the directed arc with a control flow output. Thus in the above example 'clear to move' changes the state-machine state to 'Performing motion' whilst activating the 'enable : motion control' control flow. For initiating a single activation of an activity 'prompt' control flows are used.

Data flows diagrams are hierarchical and are often developed using a top-down approach. A process can be decomposed into sub processes in subordinate DFDs. Detailed guidelines for drawing data flow diagrams can be found in Aktas [86]. Design heuristics for the use of DFDs in the requirements specification and implementation phases of a controller can be found in Ward & Mellor [105].

2.4.4 Petri-net notation and analysis

Peterson [112] describes a Petri-net as a graphical and mathematical representation of a state machine. The use of Petri-nets to model concurrent systems, including manufacturing machines, is well established. Leveson and Stolzy [26] demonstrate how Petri-nets can by used to model and analyse systems which are characterised by concurrent processes constructed from discrete states. Petri-net models being used to abstract control and interaction between the concurrent processes to facilitate safety analysis.

A high-speed placement machine controller constructed using finite state machines described by Petri-nets is detailed by Grotzinger [32]. This machine controller uses discrete positions of the machine actuator as states in the FSM. The Petri-net model using position rather than time as the principle variable for the controller. Willson [114] describes Petri-net tools for the specification and analysis of discrete controllers. The discrete nature of the control allowing a FSM characterisation in the Petri-net model. These applications do not consider timing as an important factor in the modelling process however Sagoo [118] describes how extensions to Petri-net models, timed and temporal Petri-nets can be used in the specification and design of hard real-time control systems.

Peterson [112] shows how a Petri-net consists of four parts. These are a set of places P, a set of transitions T, an input function I and an output function O. The inputs and outputs relate transitions to places. The input function I is a mapping from a transition tj to a collection of places I(tj), known as the input places of the transition. The output function O maps a transition tj to a collection of places O(tj) known as the output places of the transition. The structure of a Petri-net is defined by its places, transitions, input function and output function.

Most theoretical work on Petri-nets is based on the formal definition of Petri-net structures. However the graphical representation of a Petri-net structure as a bipartite directed multigraph is useful for illustrating the theory. A Petri-net is a multigraph, since it allows multiple arcs from one node of the graph to another. Additionally, since the arcs are directed, it is a directed multigraph. The nodes of the graph can also be partitioned into two sets, one for places and one for transitions, such that each arc is directed from an element of one set (place or transition), to an element of the other set, (transition or place).

In a Petri-net graph, conditions are represented by place nodes (circles '0') and events by transitions (bars 'I'). Placing a token on a node constitutes the holding of a condition. Directed arcs connect nodes to bars and bars to nodes. A transition (event) can fire (occur) if all the nodes (conditions) inputting to that transition have tokens (holding). When a transition fires, it removes one token from each of its inputting nodes and places one token on each of its outputting nodes. The state of a Petri-net is termed the marking and is determined by the collection of names of the nodes that have tokens holding.

The nets used in this thesis are restricted so that each node may only contain one token, allowing the representation of cooperating finite-state machines and reducing the complexity of the analysis process. A transition models a primitive event and primitive

events are instantaneous and non simultaneous. The models developed represent concurrent processes. Since each separate task within the Petri-net graph may have only one token within it, the Petri-net represents multiple sequential state-machines. Each state-machine can interact with its neighbour resulting in cooperation between state-machines. In addition the states of the FSM relate to positions of the machine actuators allowing timing constraints to be removed from the model which simplifies the modelling process.

An example of a Petri-net graph is shown in figure 2.10. It represents the motion and synchronisation of the two IDMs of the demonstration machine introduced in Chapter 1 and further described in section 2.5.

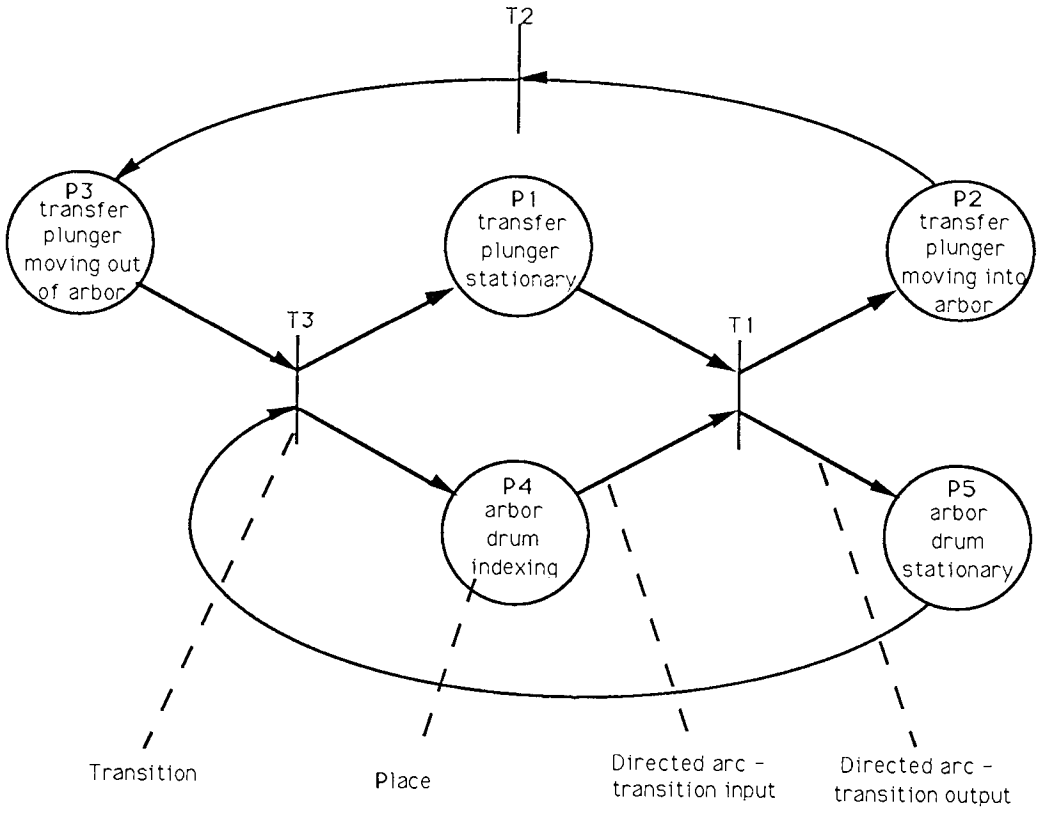


Figure 2.10. Two independent actuator Petri-net showing motion and synchronisation interlocks

The state of each actuator is represented by places in the net (P1, P2, P3 and P4, P5). It represents changes in actuator operation by transitions (T1, T2, T3 and T1, T3).

To initialise the Petri-net, tokens are assigned to places, one token is used for each mechanism. The state of the mechanisms at a particular instant is given by the presence of a token in a place, this is the current marking of the net. The Petri-net of figure 2.10 is marked and exercised by firing enabled transitions in figure 2.11 a,b,c.

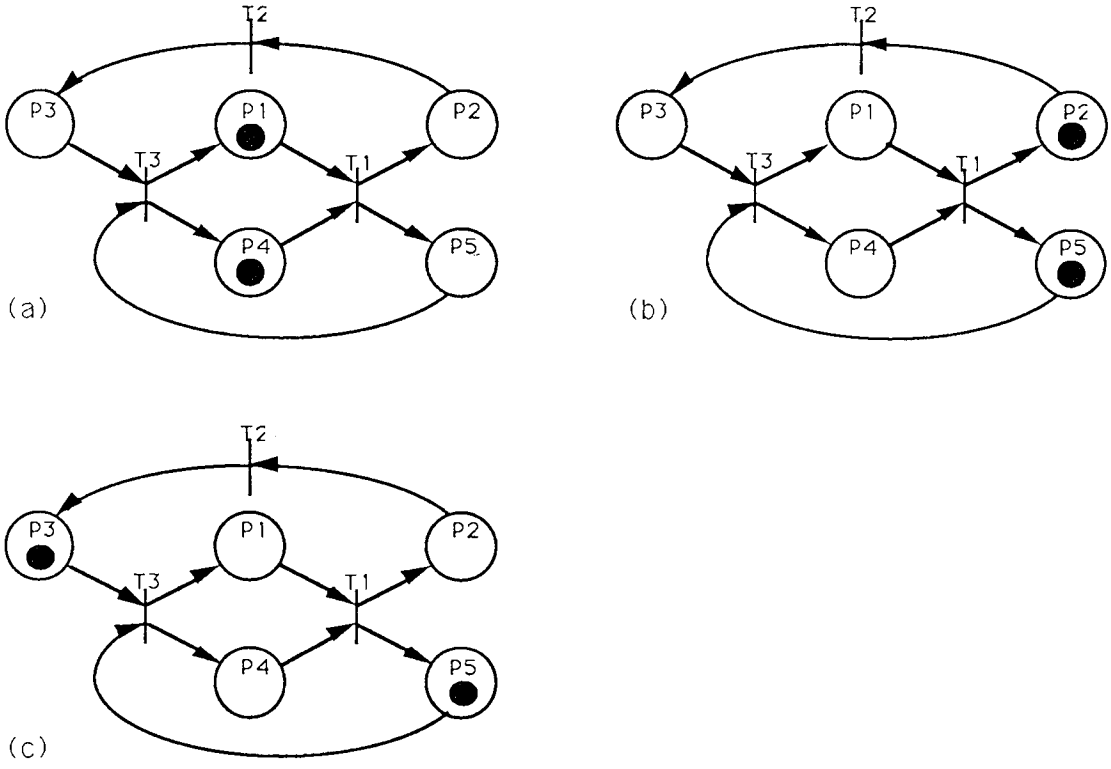


Figure 2.11 Exercising two independent mechanism Petri-net of figure 2.10.

The initial marking of 2.11a, with P1 and P4 holding a token is described, using a state space of (P1, P2, P3, P4, P5), as (1, 0, 0, 1, 0). This net marking enables T1 to fire. When T1 fires, one token is removed from P1 & P4 and instantaneously one token is placed in P2 & P5 as in figure 2.11b.

The new state of the machine is (0, 1, 0, 0, 1), in this state T2 is enabled and when fired results in a state space of (0, 0, 1, 0, 1). To complete the cycle when P3 and P5 are holding tokens T3 can fire returning the net to the original net marking of (1, 0, 0, 1, 0). The number of tokens in any place in the net is bound and does not exceed one.

Willson [118] demonstrates how the graphical form of a Petri-net can be represented as a table or incidence matrix. The vertices of this table are the places and transitions of the

net. The table indicates the action of a particular transition firing by : inserting a '-1' in the intersecting transition-place boxes with the places holding tokens; inserting a '+1' in the intersecting transition-place boxes with the places receiving tokens; inserting a *1 representing a self-loop, tokens being removed and replaced in the same place; and by inserting a #1 for inhibitor arcs from holding places. Table 2.1 represents an incidence matrix for the example Petri-net of figure 2.10.

		Transitions		
		T1	T2	T3
Places	P1	-1	.	+1
	P2	+1	-1	.
	P3	.	+1	-1
	P4	+1	.	-1
	P5	-1	.	+1

Table 2.1 Incidence matrix for simple Petri-net example

When an initial marking is applied to a Petri-net a number of transitions may be enabled. The firing of a transition changes the marking of the net which in turn enables other transitions to fire. Thus the net models dynamic execution by generating a series of state-markings by the firing of enabled transitions. Often multiple transitions will be enabled simultaneously but they cannot execute simultaneously due to a transition firing being a primitive event. An arbitrary choice is taken as to which of the enabled transitions fires. If all avenues of execution are explored then a reachability tree, described by Willson [118], of state markings and fired transitions is generated. A reachability tree of the Petri-net of figure 2.10 is shown in figure 2.12 for the given initial marking of (1 , 0, 0, 1, 0).

The use of Petri-nets to define multiple interacting state-machines allows the generation of models of both requirements and implementations. Further, Petri-nets can be used in scheduling the operations of DFDs. The use of Petri-net modelling applied to IDMs is examined further in Chapter 3.

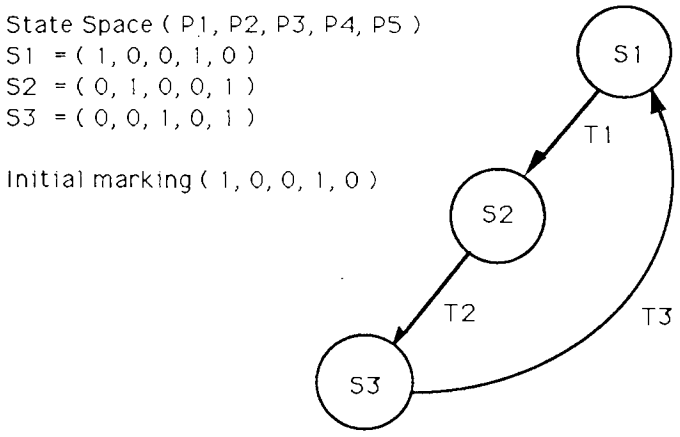


Figure 2.12 Reachability tree for Petri-net of figure 2.10

2.5 Demonstrator for intermittent motion and intermittent synchronisation

To demonstrate the flexibility of independent drive machines and to exercise the design method a study was made which involved fitting independent drives to an existing cam-actuated machine function which was known to be at the performance limit of existing technology. The motions used for this study were those of an incremental arbor drum and an intermittent transfer slider, as shown in figure 2.13.

The example uses two drives which perform cyclic, intermittent motion coordinated by discrete, intermittent synchronisation. The circular drum is required to increment in 22.50 steps and to dwell between increments.

When the drum is at rest the third transfer slider mechanism advances into an arbor mounted on the drum's circumference which contains a pack of variable size. The transfer slider contacts the pack, below the pack crushing speed, then accelerates to push the pack out of the arbor and onto a separate conveyor (not shown).

The transfer mechanism may not enter the arbor if the arbor is moving. In this exceptional condition the transfer slider will return to the rest position outside the arbor and wait until the arbor is stationary. It will then re-perform the insert/retract cycle. Due to the increased transfer slider cycle time when performing this exceptional motion product throughput and machine efficiency is reduced.

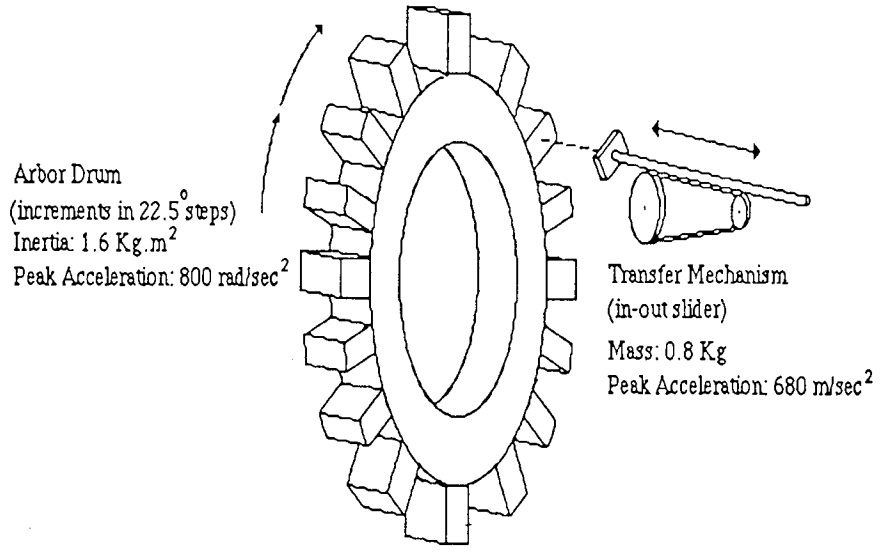


Figure 2.13. Flexible independent drive demonstration machine

The two IDMs will physically clash when the transfer is advanced to 40mm. The pack is contacted at 45-55mm depending on the pack length (30mm, +-5mm). The transfer is fully inserted at 181 mm. The drum increment and slider transfer sequence is required to take place at up to 450 times per minute, whilst maintaining drum and slider point-to-point accuracies of 1 arc minute and 0.4 mm respectively. The motion profiles and mutual synchronisation of the two IDMs are represented in figure 2.14.

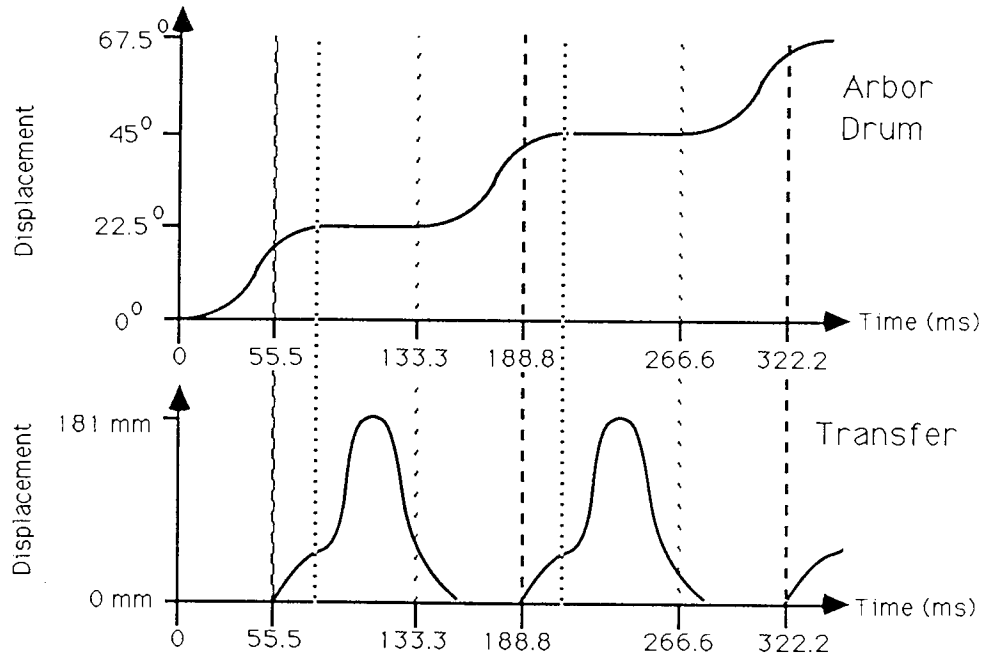


Figure 2.14. The motion profiles and synchronisation of the demonstrator

The system has very large energy requirements per motion cycle because of the large magnitudes and high frequencies of the IDM motion profiles and the large inertia of the arbor drum and transfer slide mechanisms. In the centrally driven, cam-actuated machine configuration from which the loads were taken, these energy requirements are met by the regenerative flywheel effect of the mechanical couplings. However, when the loads are independently driven, the total energy requirement for each mechanism motion must be provided directly by it's driver.

Associated work by Fenny [4] and Holding [19] investigated the selection and control of these independent mechanisms and resulted in the choice of brushless DC machines for use on the demonstrator machine. Having selected appropriate motors and power electronics it was then necessary to develop the control software.

The demonstrator was initially a single-speed fixed-function machine. However the operation of the IDM machine allows many forms of flexibility to be introduced to demonstrate the ability of independent drives to perform in a flexible manner. Each drive motion profile can be operated at different speeds, achieving speed control of the set of motion profiles. The synchronisation points of the two IDMs may be changed, or dynamically altered within a range, this provides phasing between the mechanisms and demonstrates the concurrency that exists in the machine functions. However care must be taken not to introduce unsafe behaviour by allowing hardware clashes due to inappropriate synchronisation. The particular motion of an IDM can be modified to demonstrate flexibility in mechanism function, to accommodate a change in pack length for instance. Further modifications to the motion can deal with other constraints imposed by the product, reducing maximum impact velocity to deal with a change form a hard pack to a soft pack for example.

2.6 Conclusions

This Chapter has discussed the technology and development methods required to construct an IDM machine. Multiple drive applications such as robotics, CNC machines PLC systems and existing IDM machine systems have been discussed. In order to achieve computer control of IDM machines it has been shown that off line programming techniques together with modular software and distributed hardware are necessary. In order to construct a safe and reliable IDM machine computer controller a structured

development strategy for the specification, design and implementation is required, together with methods of validating specifications and designs.

To support an IDM machine computer controller development a set of common tools and techniques is proposed. These are based upon easily assimilated DFD and Petri-net graphical notations together with associated text, databases and diagrams. These notations are proposed to support: machine requirements capture and specification within a multi-disciplinary engineering project; the generation of a validated functional specification for the computer controller; a computer control system design using concurrent FSMs and DFDs; an implementation based on close transforms of Petri-net FSM and DFD designs into real-time, concurrent software written in Occam and executed on distributed Transputer hardware. In addition executable prototypes written in Occam constructed from development models will be used during controller development to identify design faults and implementation constraints.

The principal design and modelling notations, DFDs and Petri-nets have been discussed in detail. A two IDM demonstration machine has been described which will be used to illustrate the application of the computer controller development method and the generic flexibility of IDM machine designs.

----------//----------

Chapter 3

Requirements capture for an independent drive mechanism machine

3.0 Introduction

This Chapter details the capture of IDM machine requirements. Requirements capture is the first stage in the development of such a machine and it's computer controller. The machine requirements specify what class of products the machine will make and the method of manufacture. In particular for an IDM machine the machine requirements address the problem of defining IDM motion, synchronisation and flexible operation.

To give a graphic overview of machine requirement capture and specification, the activities, models and data which are required in the process and their inter-relationships are detailed in figure 3.1. The DFD shows the three sub-phases of capture, validation and specification as activities together with associated functional models and the machine requirements database which appear as memory. The sources of requirements data form viewpoints described in Starts [87], they appear as labelled data flows into the system. The remainder of this Chapter elaborates on the requirement capture component of figure 3.1.

The requirements are drawn from a number of sources. To facilitate development a number of capture phases which reflect a source or viewpoint of requirements is proposed. These phases are sequenced to allow pivotal design decisions to be taken in the appropriate phase, guiding the requirements capture process while imposing detail only when necessary.

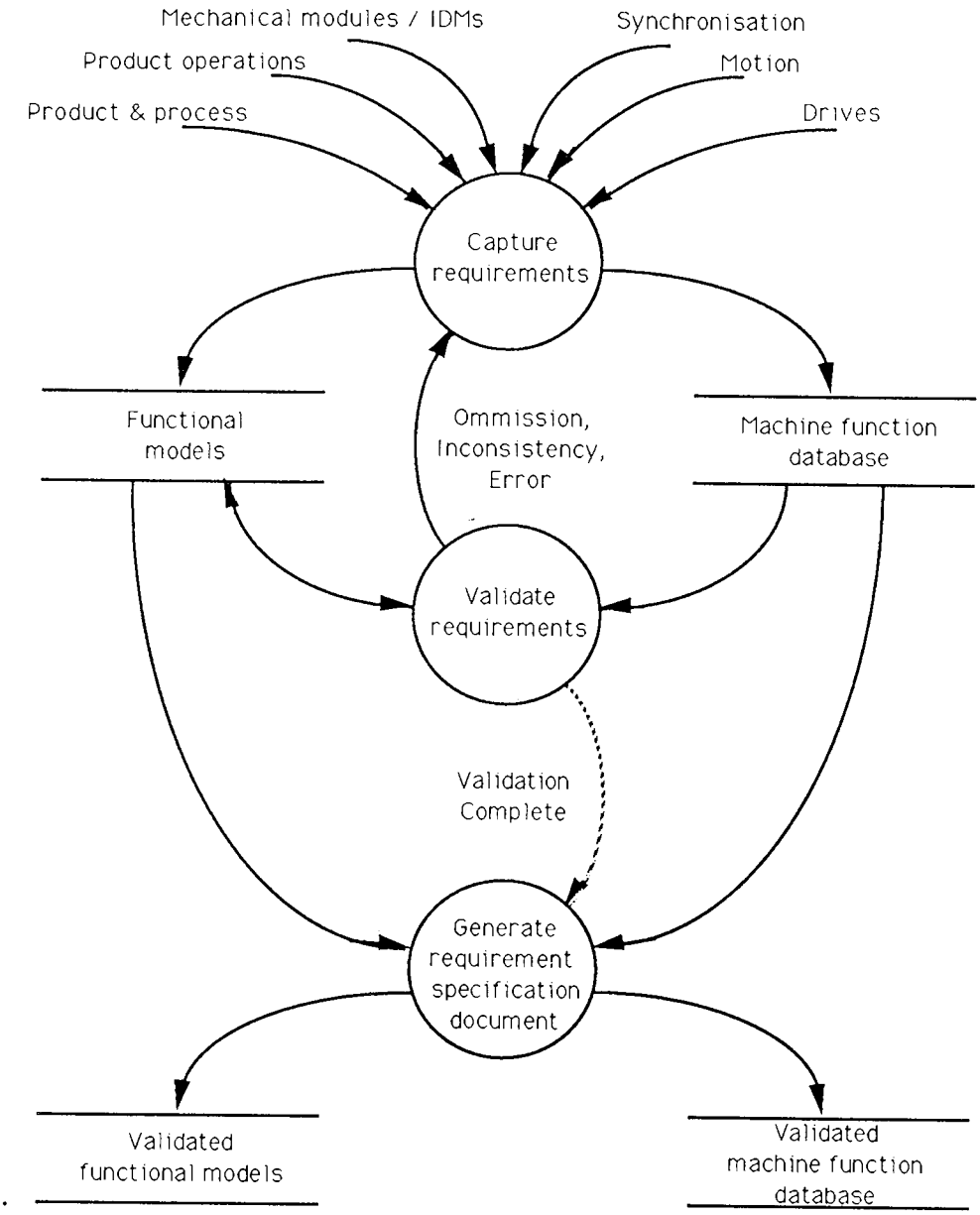


Figure 3.1. DFD of the machine requirements phase

The six phases of functional requirements capture proposed for an IDM machine are: user requirements for product and process; product forming operations; mechanical modules to perform the product forming operations; synchronisation of mechanisms; motion of mechanisms; and independent drive requirements of mechanisms.

The requirements capture process is complex, section 3.1 details the problems faced when deriving requirements for an IDM machine whilst section 3.2 proposes solutions to these

problems. Tools and notations used in the capture process are discussed in section 3.3. A general structure for capturing requirements from a source or viewpoint is proposed in section 3.4., a complete description of each capture stage is given in sections 3.4.1 to 3.4.6.

A summary of the capture phases, and notations for specifying machine requirements is given in section 3.5. Conclusions are drawn in section 3.6.

3.1 Problems of functional requirements capture for IDM machines

The functional requirements for an independently driven machine will be drawn from different sources which relate to many engineering disciplines. These sources may be development engineers, the machine user or existing machine technology and designs. The first problem of requirements capture is to provide a framework for gathering requirements from these many sources.

The requirements should be understandable by the various human sources to support validation and refinement during the capture process. Therefore methods for managing complexity and communicating requirements is necessary. In particular easily assimilated, descriptive notations are needed.

Problems arise from the omission of requirements due to the absence of a clear concept of manufacturing and process goals. Omission of requirements is compounded by the specification changes which occur as a machine development proceeds.

After initial requirements capture it is important to establish the feasibility and correctness of machine requirements to avoid fruitless development of an unfeasible machine. Methods for developing and checking interrelationships in the requirements are necessary.

3.2 Proposed solutions for IDM requirements capture

The problems raised in section 3.1 hamper the collection of IDM machine requirements. However methods and tools introduced in Chapter 2 can be used to aid capture.

The proposed framework for requirements capture for an IDM machine comes from a systematic approach to gathering and annotating machine requirements. An approach based on partitioning requirements capture by information source or viewpoint is proposed. A sequence of these stages captures requirements from all relevant sources leading to a complete machine requirements specification.

The use of documented deliverables in capture phases imposes a discipline that reduces the chance of requirement omission.

A hierarchic structuring of functional requirements allows the complexity of the process to be managed by providing varying levels of detail of requirements, the higher levels abstracting the detail of lower levels. This structure also displays the parent-child relationships between requirements in the hierarchy allowing inheritance of requirement relationships to be appreciated and examined. This hierarchical structure of machine requirements forms a layered specification of machine function.

To facilitate ease of understanding and communication of requirements a limited number of notations and data formats are proposed. Notations that support checking will be used to provide validation of requirements at an early stage. Static forms of annotation, such as data tables, will be supplemented with models to describe dynamic operation and interrelationships between requirements. The tools used are detailed in section 3.3.

The feasibility of requirement specifications can be verified by developing models and prototypes of the requirements. Petri-net models will be used to abstract requirements for dynamic testing from the overall machine requirements specification.

To limit the uncertainty caused by ill defined requirements it is proposed that parameterisation of unknown components be used, constraints on the component being defined to produce an envelope of possible functionality.

Where requirements are subject to change the impact of these changes on the requirements specification can be minimised by concentrating on generic parts of the functional requirement and avoiding the inclusion of unnecessary detail.

3.3 Tools for capture & description of requirements

The tools used for capturing machine requirements from process, mechanical and electrical viewpoints are those already used in industry: engineering drawings, written descriptions, timing-diagrams and product flow/fabrication operation diagrams. These established forms of notation will be extended, using notations and techniques introduced in Chapter 2, to support the capture of control requirements of an IDM machine. In particular computer database systems are advocated as a suitable storage medium for tabular requirements, facilitating automated cross referencing of requirements and report generation. They also form a single repository of requirements reducing the number of obsolete or replicated requirements defined. Microsoft Excel [119] has been used in the demonstrator project whilst dbase 3 [107] is in use by Molins for IDM machine development.

To annotate models showing relationships and dynamic operation DFDs [89] are proposed as a general notation. For describing state-machines, sequential operation, concurrent operation and control flow Petri-nets will be used [109][112].

The different types of independent drive machine function need different tools for annotating their requirements. Methods for annotating motion, synchronisation, flexibility and safety are necessary. A static database of machine requirements from different capture viewpoints will hold unstructured specification data, while functional models of requirements will give structure to database entries. The models, by combining components of the database into coherent activities, impose structure and facilitate analysis.

The following sections detail how the machine requirements can be captured, verified and specified in a rigorous manner.

3.4 Overview of capture sequence and deliverables in a capture phase.

The proposed method of requirements capture partitions requirements according to source or viewpoint and sequences the gathering of requirements data from each of the partitions. For each partition or phase, capture sources can be defined. The inputs to a phase are the requirement deliverables from the previous capture phases. The output

requirements are annotated as deliverables and passed to the next phase in the requirements capture sequence. In this way a hierarchy of requirements is formed with increasing requirement detail down the hierarchy. Figure 3.2 shows the sequence of requirements capture phases for an IDM machine.

Requirements may be interrelated at a given level in the requirements hierarchy and a cross reference may be included to show the relationship. In addition requirements relationships crossing phase boundaries can be defined as parent-child relations, facilitating inheritance of requirements between phases. Traceability of requirements, a key facility identified by Birrell [88], through the phases and the resulting hierarchical requirement specification is therefore supported.

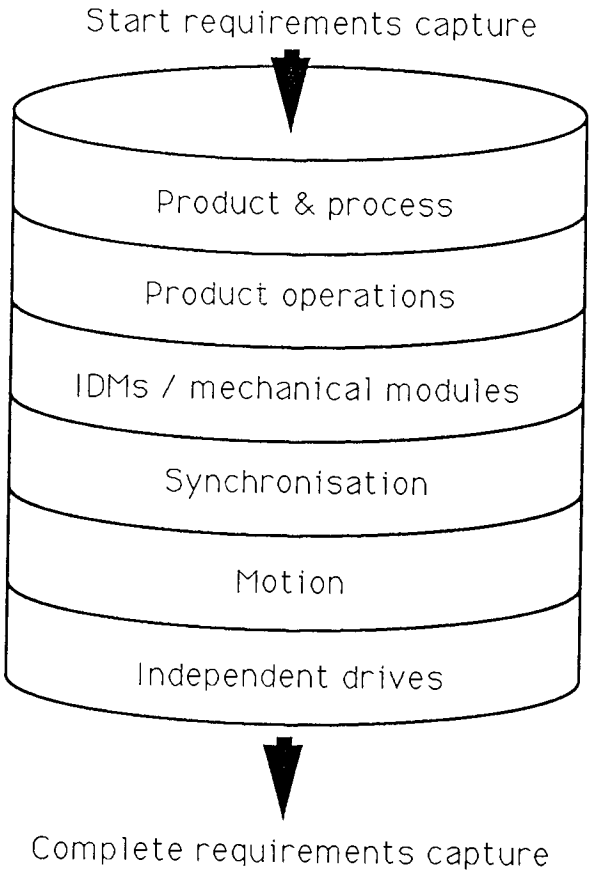


Figure 3.2 Sequence of requirements capture phases for an IDM machine

The four phase components of input requirements, capture sources, output requirement deliverables and cross references of requirements can be used in each phase of the capture process.

The requirements capture sequence of figure 3.2 will now be discussed to examine the capture phases in detail.

3.4.1 User requirements for product and process

The intended user of a machine under development will be the initial source of two types of machine requirement. The first type of requirement will define the class of products to be made by the machine, incorporating details of variation in construction necessary to produce different products (product flexibility). The second type of requirement addresses the manufacturing process by defining operating limits and constraints for the process such as required efficiency, product tolerances, speed ranges and required machine configuration times (process flexibility).

Product requirements can be captured from the user's development engineers, existing product lines and prototype developments of new products. Process requirements can be captured from factory managers, machine operators, production engineers and existing machinery.

Requirements for product and process are not based on previously specified requirements since this is the initial capture phase. The phase depends upon the capture sources for requirements information.

The captured requirements pertaining to product and process form the highest level of requirements in the machine requirement specification hierarchy. They are fundamental in leading future phases of requirements capture. The deliverables of this phase are detailed below.

The primary deliverable is a description of product to be produced including variations in manufacturing to produce different classes of the product. To facilitate this a series of intermediate products which when fabricated and combined lead to the final complete product can be defined. In addition constraints on product should be defined. These may include physical constraints such as maximum manipulation speed, or process constraints such as mechanical handling suitable for food manufacture.

The secondary deliverable is a description of the operating characteristics of the manufacturing process. These include maximum and minimum operating speeds and

capability of implementing flexibility in manufactured product. An overview of the user interface and requirements for connections to other manufacturing machines or factory control systems should also be provided.

The product and process requirements can be annotated using the following techniques. Engineering drawings can be used to graphically show the complete product and specify dimensions. These dimensions may be stored on a suitable database with the inclusion of flexibility in dimensions to produce different classes of product. The complete product may also be described as a number of discrete components and sub-assemblies. These intermediate products being described in the same way as the finished product. A textual description of the assembly sequence of the product may be included. The product requirements will specify the range of products manufactured by the machine. Assemblies can be defined as groups of components, using bill of materials notations. Constraints tables for operating parameters of the machine can be drawn up and entered into the machine requirements database. Diagrams and templates for the user interface, based on process requirements, can be drawn.

At this level cross-referencing of requirements is limited. However many of the requirements captured can be used in verification of requirements introduced in later phases. For example the product dimensions, tolerances and flexibility can be used in the verification of correct operation of mechanisms and motion profiles. Thus parent-child relationships between requirements in the hierarchy are important in assuring the quality of the final machine requirement specification.

An example of a product and process requirement for the demonstration machine follows.

The product involved in the demonstrator is a rectangular pack of varying length, but constant width and depth as shown in figure 3.3. The pack has a maximum manipulation speed of 4.0 m/s and has a crushing impact speed of 2.0 m/s.

The process requirement for the demonstrator is simple. The machine must transfer product at varying speeds. The maximum speed of the demonstrator is to be 450 ppm (packs per minute) the minimum speed is 0 ppm. The speed is to be dynamically variable between these two limits. There is a manufacturing constraint that the mechanical operations be performed to an accuracy of 2mm on the pack.

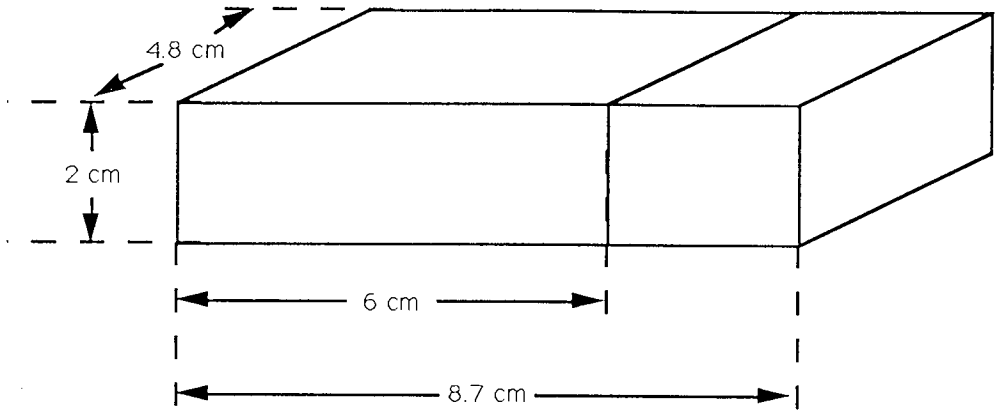


Figure 3.3 Product specification for the demonstration machine

3.4.2 Product forming operations

The second phase of requirements capture for an IDM machine identifies the various operations that must be performed to form the product. There are three main types of operation on product. The primary operations are those associated with fabrication of a product component. These operations are the essential activities of the manufacturing system defined by McMenamin [89]. Secondary operations are those associated with transporting raw material and partly complete product between fabrication operations. A separate set of operations may be required to accommodate abnormal situations, ensuring the quality of manufacturing by removing faulty product and safe operation of the machine by responding to unexpected behaviour of machine or process.

An individual product forming operation may involve manipulating continuous or discrete raw material and components. The manufacturing process will often commence with continuous operations on material stock and finish with discrete operations on individual products. For example the Molins maker converts continuous flows of paper web & food product into individual bags.

The set of product forming operations, the sequence of these operations and their interrelationship defines the overall manufacturing process. In order to describe these relationships data flow diagrams may be used. Hierarchical grouping and decomposition of the product operations, a technique described by Ward & Mellor [105], will be used in later phases leading to a description of the physical and logical layout of the manufacturing process and hence the machine.

The required product operations can be captured from a number of different sources. The primary source of information will be the process design engineers who will take requirements for products from the previous phase. Existing machine designs will often accomplish similar manufacturing operations which may also be incorporated into new designs.

Proof of concept test rigs may be constructed to assess the performance of a product operation and experimental design used to refine the prototype product operation. The demonstrator machine example is a prototype rig for intermittent operations on discrete product taken from an existing packing machine described by Fenny [4].

The product operations are necessary to transform raw material into the intermediate products defined in the product and process requirements capture phase. The method of achieving the transformation is not required in this phase but the relationships between operations should be specified. The relationships should specify hierarchy, sequence and concurrency in operations and will form a basis for identifying mechanical partitions of operations.

The main deliverable of this phase will be a diagram showing the product forming operations of the machine, their spatial relationships, and the sequence and concurrency in the operations.

This diagram can be used to define a list of product forming operations delimited by operation type: fabrication, transportation or quality/safety. The concurrency and sequence of operations will lead to a preliminary physical layout of the machine. In addition a diagram showing the requirements for flow of raw material and product in the machine can be created which may be annotated with the material throughput required to achieve specified machine performance. Thus the constraints on machine operation can be refined to include throughput of material for the various flows in the machine.

The operations defined in this phase can be described and related using data flow diagrams. An example for the demonstrator being given in figure 3.4. The DFD notation allows the description of raw material and product flows in the machine as data flows and the product operations as activities on data. This diagram therefore describes the functional requirement for material flow in the machine and the interrelationships of product operations. It forms the basis of information for partitioning operations into modules in the next phase.

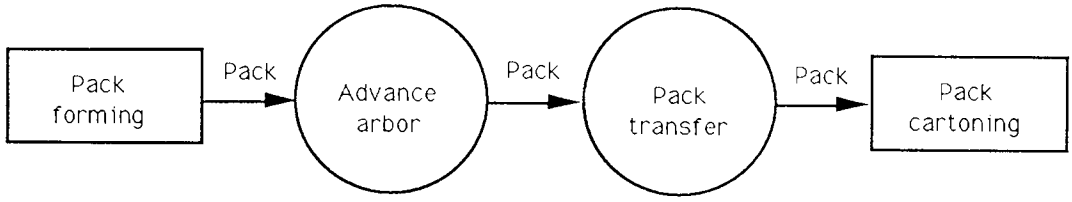


Figure 3.4 Product forming operations in the demonstration machine.

A list of all machine operations required in the machine can be generated using a database package. Constraints on products specific to a particular operation can be included. Sequence and concurrency of operations will be implicit in the DFD of the product operations however more detailed notations including Petri-nets and timing diagrams may also be used.

Refinements to the specifications of the intermediate products can be made and methods of achieving the manufacturing operation defined using a textual description of the dynamic process.

There exists a child-parent relationship between product operations and overall manufacturing process operation. Cross references within product operations are required to ensure that the set of operations can achieve the required product throughput defined in the previous capture phase. Any deficient operations identified will require further requirements capture of suitable operations.

The demonstrator performs the function of a single mechanical module present on an existing machine.

The demonstrator has only two product operations. These are both transport operations. The first operation is the rotary indexing of an arbor drum to move a pack filled arbor into a predefined position related to the pack transfer mechanism. The second is the movement of the pack out of the arbor by the translating motion of the transfer mechanism. An initial definition of these operations can be seen in table 3.1

Product operation	Type	Rate (ppm)
Advance arbor	Transport	450
Pack transfer	Transport	450

Table 3.1 Product operations in the demonstrator

3.4.3 Mechanical modules to perform product forming operations

The mechanical module phase of requirements capture groups the previously identified product operations into discrete modules. These modules are then examined to identify where IDMs may be used and if the application of an IDM is appropriate. This partitions the machine into mechanical modules to facilitate modular machine construction and defines where IDMs will be used.

The goal of the phase is to define a list of mechanical modules and their constituent product operations and to list the number of IDM to be used in the machine. The relationship between IDMs and conventionally driven mechanisms (CDMs) can be specified.

The mechanical mechanisms which perform the product operations in a module can be defined by mechanical engineers using conventional engineering-drawing notation. Groups of product operations can be annotated as DFDs.

The sources of requirements for this phase include: mechanical and process design engineers, who can specify mechanisms suitable to achieve the required product operations; existing machines for proven IDMs and conventional mechanisms; mechanical component almanacs such as Naylor [46], for engineers when tackling standard mechanism problems. Prototypes of the mechanical modules can be made to examine requirements and as proof of concept rigs for both users and managers of the machine development.

In order to define a set of mechanical modules the capture method requires that a module's constituent product operations and a method of providing these operations by

suitable IDMs and CDMs be specified. To facilitate this the requirements of the previous phase can be utilised. In order to group a set of product operations into discrete mechanical modules the research has identified a number of design heuristics:

Grouping product operations required to produce an intermediate product can be used as a first step in partitioning into mechanical modules. This is termed intermediate product partitioning.

The identification of common modes of synchronisation, an important factor when implementing IDM controllers, related to product operations can be used to group operations and to define links between mechanical modules, the capture method denotes this as common synchronisation partitioning. This is particularly appropriate for transport operations since the flow of material through a machine is often continuous requiring continuous motion and continuous synchronisation between transport operations.

The sequence of operations required to make a product can be used to define sequence relationships between mechanical modules in a DFD. This is termed fabrication sequence partitioning.

A preliminary assessment of the IDM requirements of a mechanical module can be initiated by examining the presence of intermittent or continuous motion operations within the module. Product operations which are intermittent or which require flexibility are especially suitable for implementation using programmable IDM's. This can also be an indicator for the best type of inter-IDM synchronisation method to be adopted, continuous synchronisation for close coupling between IDMs, discrete synchronisation for loose coupling.

An example of a DFD for mechanical module partitioning for the demonstrator can be seen in figure 3.5.

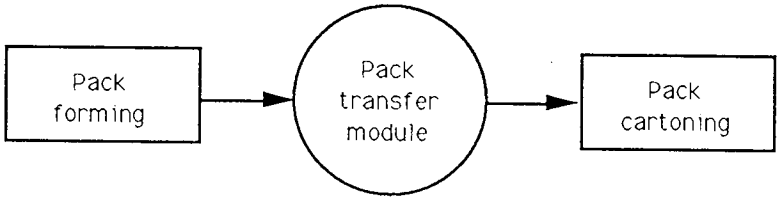


Figure 3.5 Mechanical module partitioning for the demonstrator

The identification of a number of mechanical modules, their component operations and the mechanical modules interrelationships will be used as a basis for mechanical design of the machine and process. The mechanical design of product operations using conventional drive techniques does not influence the design of the IDM computer controller, since a conventional drive train can be treated as an IDM comprising a single drive together with a complex mechanism. Mechanical design of mechanisms however is outside the scope of this thesis.

To facilitate requirements capture of IDM operation and synchronisation, it may be necessary to define a particular product operation as a machine datum reference point. This point will be a physical position of a drive in a mechanical module, for use in datuming product when setting up the process. It may also be a position master of use when synchronising a number of IDMs. If a continuous motion is present in an IDM which represents a complete cycle of the machine, this motion should be used to define the datum point since it will provide a unique repeatable datum position. This datum point may also be used when describing phase relations between IDMs.

Mechanical modules can be described by hierarchical partitioning of the product operations DFD described in the previous phase. This results in a simplified higher level view of the manufacturing operation. This view is created by a hierarchical abstraction of machine function based on intermediate product, common synchronisation and fabrication sequence partitioning.

Relationships between modules such as common synchronisation can be denoted by control flows to the appropriate modules together with an appropriate CFD.

The product operations specified as being independently driven can be listed in an appropriate database. The entry for each IDM will have a number of additional requirements added in later phases as refinements are made to the IDM specification.

Mechanical designs for product operations will be produced using engineering drawing techniques. The mechanisms may be exercised using an animation tool such as Camlinks [120], to validate the operation of the mechanism.

The type of mechanism operation used in a multiple-drive machine will influence the method and content of the requirements capture phase since different motion generation and synchronisation schemes provide for different types of machine operation.

For the demonstrator the defined product operations will be achieved by a single mechanical module. This module requires two product operations working as part of a product fabrication sequence. The mechanism motions for these product operations are intermittent. The transfer motion requires flexibility to accommodate variations in pack size and process speed, in order to provide this flexibility an IDM will be used. The arbor drum motion is not modified by changes in product however speed flexibility is still required. To support process flexibility the arbor drum will also be independently driven.

The mechanical layout of the mechanisms of the demonstrator is shown in figure 2.13 of Chapter 2. Mechanical specifications for the two mechanisms are given in the database entry of table 3.2.

IDM	Max position (ec)	Max velocity (ec/s)	Max accel	Mass	Pulley Radius (m)
Transfer	8000	50,000	680m/s^2	0.8kg	0.1
Arbor	---	5,000	800rad/s^2	1.6kg.m^2	0.4

Table 3.2 Mechanical specifications of IDMs

A preliminary sequence of activities for each product operation is given in the two FSM Petri-net in figure 3.6. The Petri-net represents each IDM by a separate FSM containing a single token. Each FSM describes the physical state of the IDM (p1->p4 represents the transfer slider operation, p5 & p6 the arbor operation). Sequencing of states are denoted by directed arcs and transitions. A single path of transitions impose sequence (t1,t2,t3 & t5), return transitions (t4, t6) represent cyclic operation.

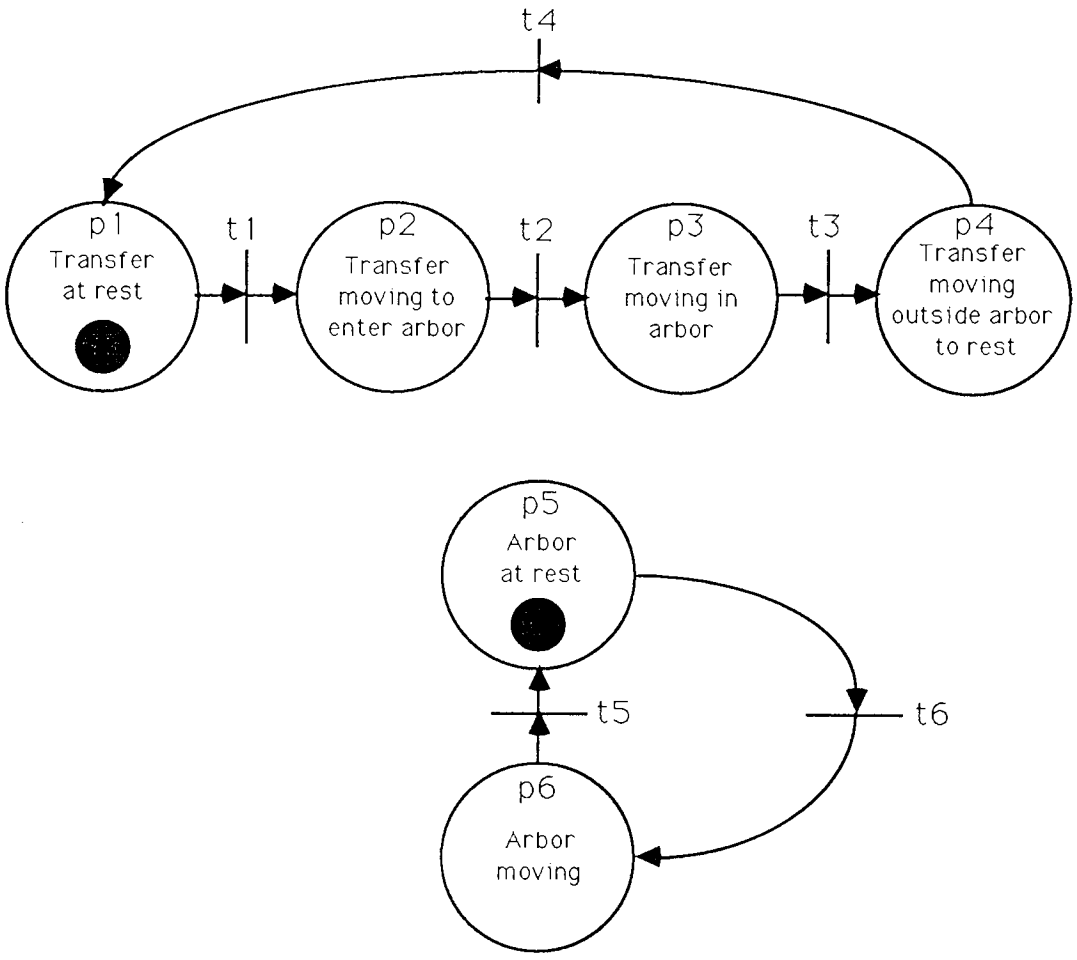


Figure 3.6. Petri-net FSM sequence of activities for demonstrator IDM operations

3.4.4 Synchronisation of mechanisms

There are two types of mechanism synchronisation, for continuous synchronisation between mechanisms the set of mechanisms is synchronised at all times and positions, for discrete synchronisation individual cooperating mechanisms synchronise at discrete positions.

Conventionally driven mechanisms utilise a single prime-mover and drive shaft. The fixed mechanical linkage of this shaft to the machine mechanisms provides continuous synchronisation, other mechanisms such as geneva wheels described by Fogarasy [6], provide discrete synchronisation. The synchronisation is inherent in the mechanical design and does not require feedback of mechanism operation.

Discrete synchronisation of an IDM however, uses a particular event or condition in the dynamic operation of the machine to trigger a mechanism, requiring feedback of machine state. For example a photoelectric sensor may detect presence of product to initiate an eject mechanism. Discrete IDM synchronisation requires closed loop control whereas conventional synchronisation uses open loop control.

The research has identified three forms of inter-mechanism synchronisation for IDM machines denoted by the following classifications. Hierarchical synchronous mechanisms rely on a master motion/synchronisation profile. This master profile is broadcast to individual IDMs in real-time and hence embodies the continuous synchronisation requirement, Tal [50] describes how this will be generated in software.

Parallel synchronous mechanisms execute independent motions using a fixed inter-mechanism timebase, implemented in software as a common clock between IDM controllers.

Parallel asynchronous mechanisms execute independent motions using a timebase local to the mechanism controller. The synchronisation between mechanisms is discrete. Each discrete synchronisation point being coupled to the mechanism motion.

Hierarchical synchronous and parallel synchronous synchronisation methods utilise open-loop synchronisation between mechanisms whereas parallel asynchronous synchronisation requires feedback of mechanism state between IDM controllers and is a closed-loop synchronisation technique.

In an IDM machine implementation failures in synchronisation can be detected for all three synchronisation methods. However to achieve recovery of failed synchronisation requires dynamic control. Closed-loop feedback of mechanism state supports dynamic control by allowing alternate motion profiles to be executed after a synchronisation failure. To support synchronisation recovery parallel asynchronous synchronisation should be used in association with alternative motion profiles.

The machine designer will be responsible for specifying synchronisation between CDMs and IDMs. Existing mechanisms and mechanical modules identified in the previous capture phase can be analysed to determine the synchronisation method used.

The capture of synchronisation method between machine mechanisms will depend upon the required motion of the mechanisms. Continuous motion with simple transforms from a master motion profile, such as gear-ratios, are most suited to hierarchical synchronous synchronisation. Continuous motion with unrelated complex motions which have a common timebase, such as cams, can use parallel synchronous synchronisation. Intermittent motion or motions with dynamic profile changes are best suited to parallel asynchronous (discrete) synchronisation.

The synchronisation method of a mechanism will impact the implementation of the final computer controller. The logical coupling between mechanisms will be a principle partitioning heuristic when designing a modular computer controller. A diagram showing this logical coupling between mechanisms can be used as a primary source of information for software design. Thus the logical partitioning of the machine controller does not have to follow the mechanical module partitioning introduced in section 3.3. This emphasises the difference between machine requirement specification and computer controller design.

In order to annotate synchronisation relationships between mechanisms a hierarchical DFD will be used since continuous and discrete DFD data flows can represent continuous and discrete synchronisation. The data flows show synchronisation method and the activities denote machine mechanisms. This context diagram of machine synchronisation can be elaborated to show the synchronisation data required for the three types of mechanism introduced. Hierarchical synchronous mechanisms will be related to software master motion/synchronisation profiles. Parallel synchronous mechanisms will be related to common-clocks and parallel asynchronous mechanisms will use data flow prompts elaborated as Petri-nets to show discrete synchronisation between cooperating mechanisms.

In the demonstrator example the two IDMs perform intermittent motions, the motions are synchronised to provide motion sequence and safety interlocks at discrete positions. The IDMs are parallel asynchronous mechanisms. The transfer arm IDM must pass into the arbor IDM and a mechanical clash is possible if the transfer attempts to enter a moving arbor or the arbor starts to move while the transfer arm is still inserted. The synchronisation must therefore be both correct and fail safe. In addition there is also a requirement for recoverable operation of the module if synchronisation should fail. Discrete synchronisation based on position feedback of mechanisms will be used together with alternative motion profiles for synchronisation recovery. A DFD of the synchronisation is given in figure 3.7.

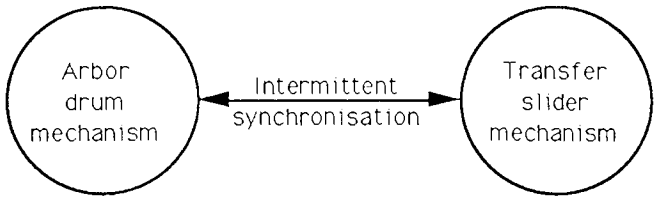
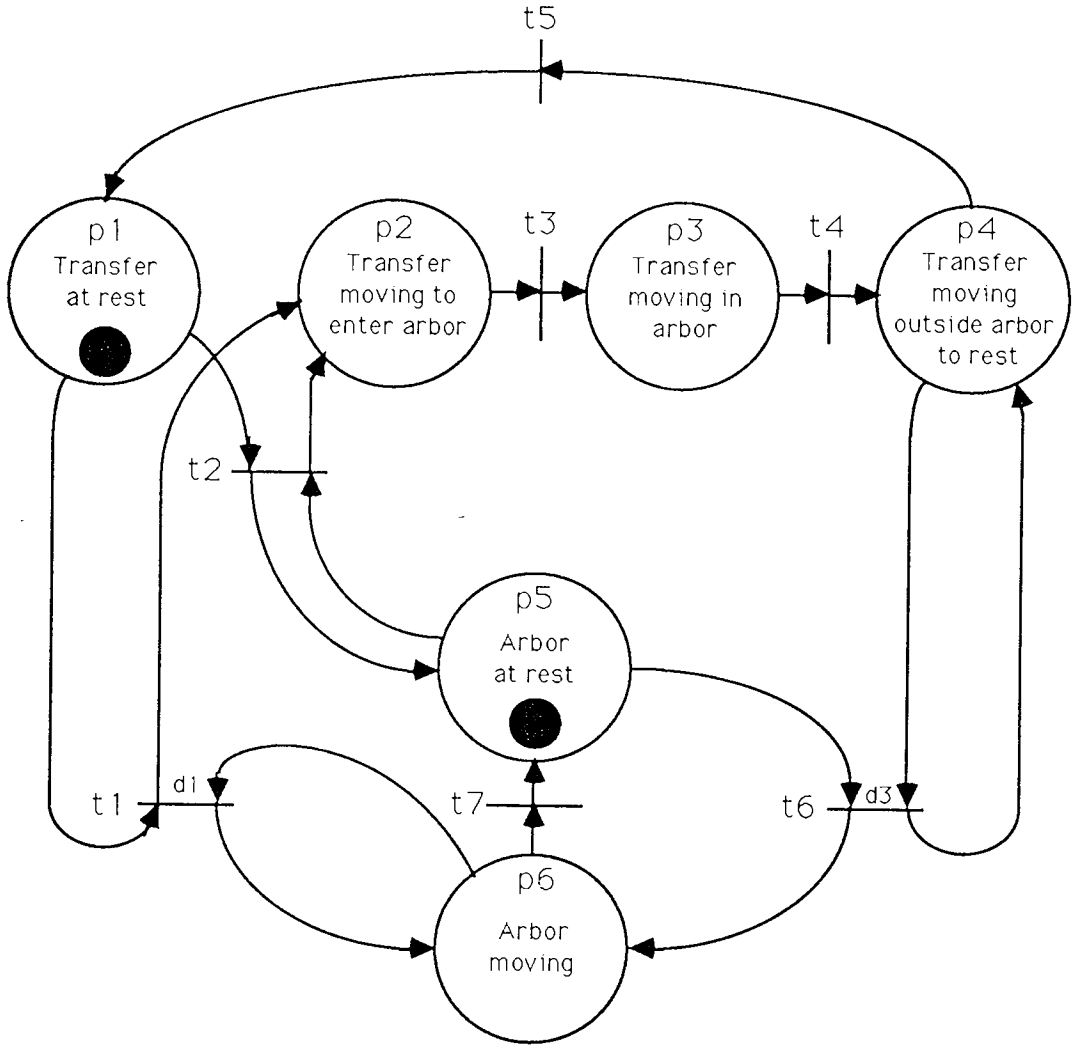


Figure 3.7 DFD of Logical connection between mechanisms (synchronisation method) for demonstrator

An elaboration of the discrete synchronisation introduced in figure 3.7 is given in the Petri-net of figure 3.8 which is in turn based on the sequence of operation Petri-net of figure 3.6. Each mechanism is annotated as a series of places and transitions. Places in the net are associated with position states of the mechanisms, transitions denote the presence of a motion or synchronisation event for the mechanism. Only one token is allowed per mechanism, thus the places and transitions form a FSM description of the mechanism operation.

Figure 3.8 show transitions to support inter-IDM sequencing but not inter-IDM safety interlocks, which will be identified during validation analysis in Chapter 4 section 4.1.2.3. The discrete synchronisation events for motion sequencing between IDMs are represented as rendezvous transitions and are annotated to detail the initiating parameter, this is mechanism displacement for the demonstrator. The model details the initial conditions of each IDM (arbor & transfer at rest, denoted by places p1 and p5 holding tokens) and also describes the starting sequence (arbor and transfer at rest, transfer moves first, denoted by rendezvous transition t2).



d1 : arbor at displacement to initiate 'commence transfer move' rendezvous
d3 : transfer at displacement to initiate 'arbor clear to proceed' rendezvous

Figure 3.8 Petri-net of required inter-IDM sequencing synchronisation for the demonstrator

3.4.5 Motion control of mechanisms

This phase of requirements capture defines the motion of each of the component mechanisms in the machine. Annotation of motion for IDMs and CDMs is necessary together with details of required flexibility in the motion of an individual mechanism and interrelationships in motion trajectories between mechanisms.

There are two basic types of end-effector motion, for a continuous motion mechanism the motion is ceaseless, and for an intermittent motion mechanism the motion contains dwells. Both intermittent and continuous motion may be repetitive to perform the same operation on discrete products in the machine. This repetition of a mechanism motion related to product has been termed a 'mechanism cycle'. The set of mechanism cycles, which together produce a processed product, is termed by Fenny [15], a machine cycle.

To refine the description of mechanism motion the classification of mechanisms introduced in section 3.4.4 will be used.

'Hierarchical synchronous' mechanisms follow a master motion profile generated outside the mechanism. The externally generated motion profile may be modified by the mechanism before execution, for example to generate a gear-ratio. The master motion profile provides synchronisation and motion trajectory generation for the set of mechanisms relying on it. A mechanical example of a master motion profile provided by Strandh [5] is the motion of a conventional machine's main drive shaft, hierarchical synchronous mechanisms attached to this shaft perform motion according to the speed of rotation of the shaft and the set of mechanisms is synchronised by the fixed mechanical connection to the shaft. For a set of IDMs the master motion profile will be provided in real-time using software.

'Parallel synchronous' mechanisms use a common method of synchronisation but each individual mechanism generates it's own motion profile. Thus the set of 'parallel synchronous' mechanisms perform individual motion profiles which have fixed spatial and temporal relationships. A mechanical example of parallel synchronous mechanisms can be found in a cam-box where the cams are synchronised by the common cam-shaft but the individual mechanism motions are determined by the profile of the cam and follower of the mechanism.

For 'parallel synchronous' IDMs a common timebase used across the set of individual IDM trajectory generators provides the common synchronisation. This method is used for the Molins maker machine introduced in Chapter 2, due to the simplicity of implementation and a limited machine requirement for flexibility in motion profiles.

Finally 'parallel asynchronous' mechanisms are characterised by having mechanism motion requiring limited inter-mechanism synchronisation. The 'parallel asynchronous' mechanism generates it's own motion profile and synchronises with other mechanisms at

discrete positions. The synchronisation between mechanisms does not rely on temporal information.

The three types of mechanism will often be used together in one machine. To facilitate the annotation of the interrelationships between types a combination of data flow and Petri-net diagramming techniques can be used, to allow the description of continuous and discrete synchronisation respectively.

The flexibility in the operation of an IDM machine is directly attributable to the programmable motion control of the machine's independent drives. In order to utilise this flexibility it is necessary to capture the required motions with the flexibility included. The motion profile of an individual drive in the machine will be modified for two principle reasons. The first will be a change in operating speed to facilitate a machine-wide speed change in the manufacturing process. The second will be a motion modification to accommodate a change specific to the product operation associated with the IDM, such as a phase advance relative to the machine reference.

The primary motion profile for a given mechanism is the desired motion trajectory of it's end effector. The motion trajectory requirements can be captured from a number of sources. The principle source will be the mechanical engineers who will specify the required motion trajectory of the mechanism's end effector. To determine this motion trajectory prototypes of mechanisms, mechanism almanacs and existing machine mechanisms can be used.

The phase should capture the motion trajectory of an IDM's end effector incorporating all the variations in trajectory relating to product and process flexibility. The motion trajectory of the end effector should then be related to the mechanisms drive or drives to produce a drive motion trajectory. In the final machine this motion trajectory will be generated by the IDM computer controller to drive the IDM through the required manufacturing operation.

To capture motion profiles a graphical method of depicting motion is proposed. This is supported by a table of motion parameters, which will detail position, velocity and acceleration at critical points on the required motion trajectory. An entry in the motion specification table will define a simple motion segment of the profile. Typically initial and terminating position, velocity and acceleration will be used when defining such a segment. The sequencing of the set of segments produces complex motion profiles, completion of one segment initiating the execution of the next in sequence. Dynamic

changes in the sequence allows alternative motion profiles to be generated. Petri-nets will be used to annotate sequences of motion segments.

The motion specification table will include details of flexibility required in motion segment parameters, typically maximum and minimum values of parameters are presented associated with a flexibility identifier. This identifier allows dynamic control of motion profile flexibility in an implementation. Often a single identifier will be used across a set of IDMs to provide global control of IDM flexibility, for example a speed identifier may be defined which is used by all IDMs to support machine speed control in an implementation.

To annotate interrelationships between motion profiles data-flow diagrams will be used for describing hierarchical synchronous and parallel synchronous mechanisms. To annotate motion and synchronisation of parallel asynchronous mechanisms Petri-nets will be used.

The demonstrator IDMs both require intermittent motions. The arbor performs circular indexing motions whilst the transfer performs cyclic translating motions. The motions are both too complex for a simple transformation from a software master profile. The motions may be decoupled from each other therefore a common clock is not required. Flexibility is required in both motions to support speed changes in the process. The motion profile views of figures 3.9 to 3.12 give the motion profiles required. They are derived from the demonstrator product operations described in section 2.5. Simple motion segments are labelled, the normal segment sequence being shown by the motion profile. The transfer slider also has an abnormal motion profile (segments 1,7 & 8), this is defined to allow recovery after a synchronisation failure. The arbor mechanism shows position and velocity flexibility in motion segments 1 to 4. The transfer mechanism shows position flexibility in motion segment 2 and velocity flexibility in segments 1 to 8.

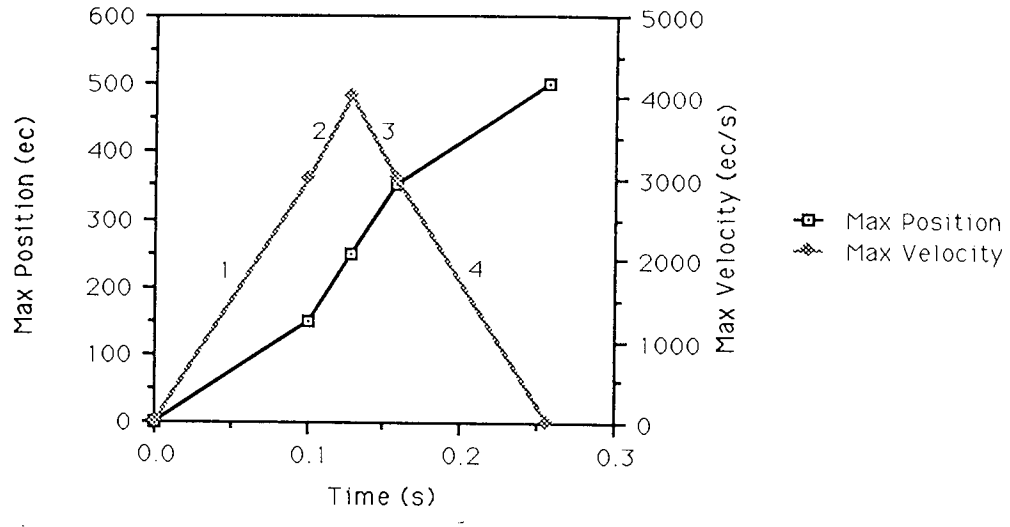


Figure 3.9 Arbor maximum position / velocity graph

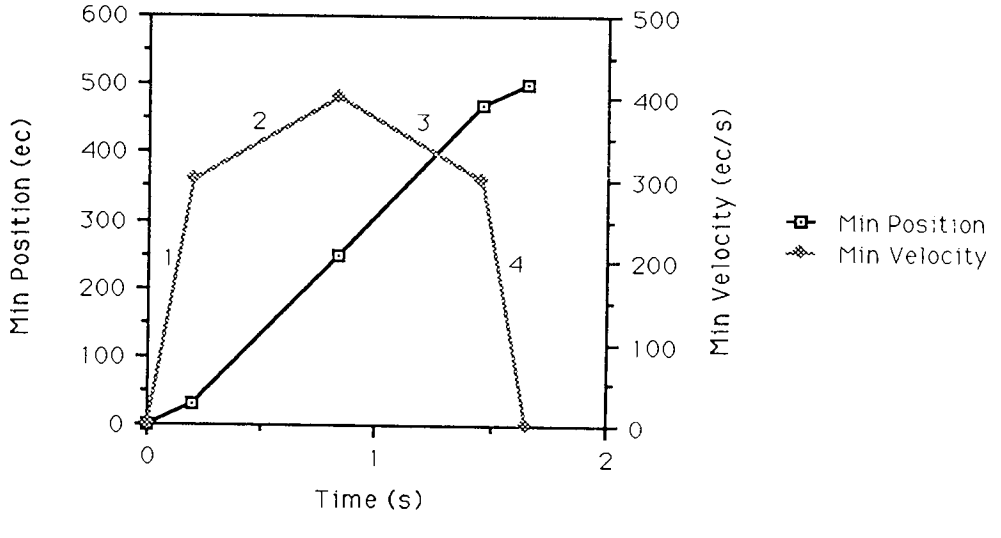


Figure 3.10 Arbor minimum position / velocity graph

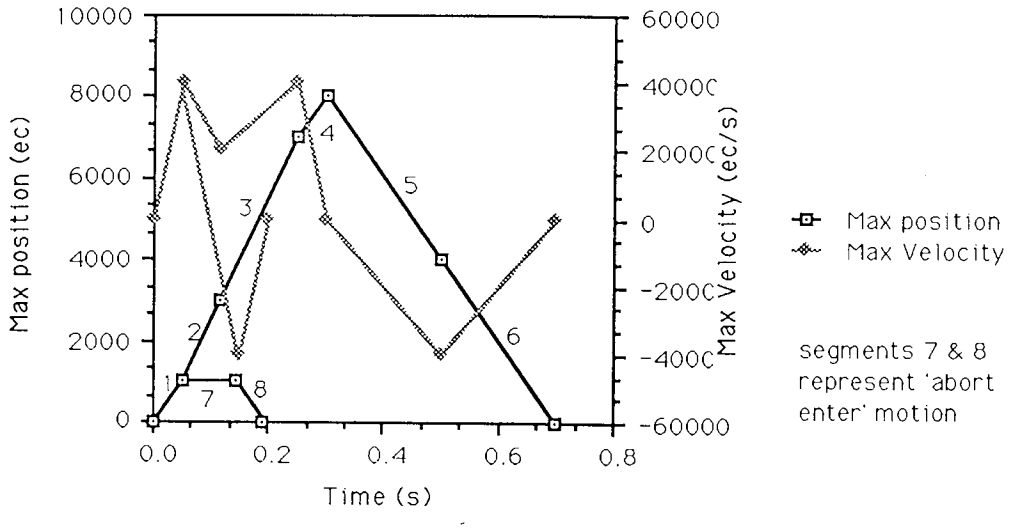


Figure 3.11 Transfer maximum position / velocity graph

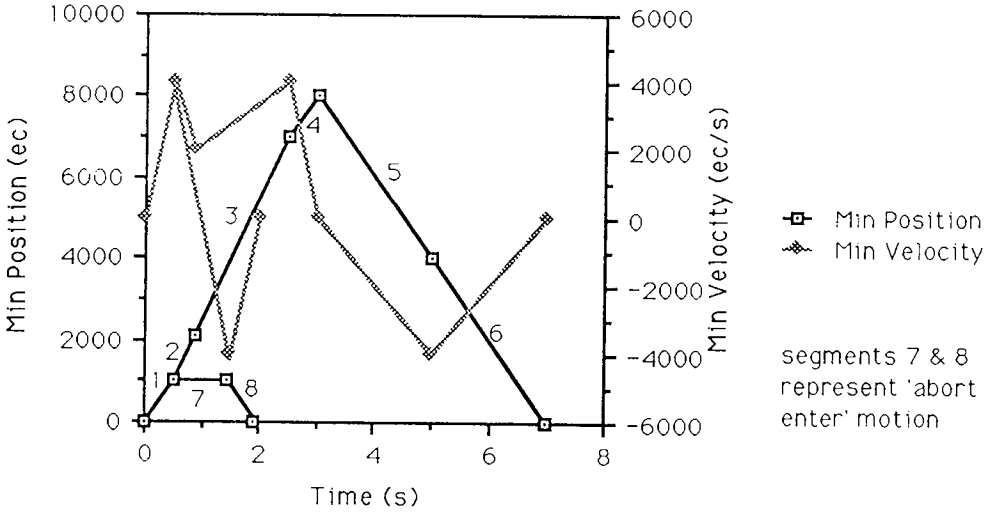


Figure 3.12 Transfer minimum position / velocity graph

The motion profile segment specification tables of 3.3 and 3.4 show the complex motion profiles of figures 3.9 to 3.12 as a series of motion segments. Each segment has defined terminating conditions and associated variable parameters.

Motion segment definition: **Arbor Drum Mechanism**

Segment	Max position	Min position	Max velocity	Min velocity	Posn flex var	Vel flex var
1	150	30	3000	300	2	1
2	100	220	4000	400	2	1
3	100	220	3000	300	2	1
4	150	30	0	0	2	0

Table 3.3 Motion segments for arbor drum mechanism

Motion segment definition: **Transfer slider mechanism**

Segment	Max position	Min position	Max velocity	Min velocity	Posn flex var	Vel flex var
1	1000	1000	40000	4000	0	1
2	2000	1100	20000	2000	2	1
3	4000	4900	40000	4000	2	1
4	1000	1000	0	0	0	0
5	-4000	-4000	-40000	-4000	2	1
6	-4000	-4000	0	0	0	0
7	0	0	-40000	-4000	0	1
8	-1000	-1000	0	0	0	0

Table 3.4 Motion segments for transfer mechanism

To sequence the motion segments the Petri-nets of figure 3.13 have been defined. The sequences define a mechanism cycle for each IDM. Two sequences are required for the transfer slider to describe both normal and abnormal motion profiles.

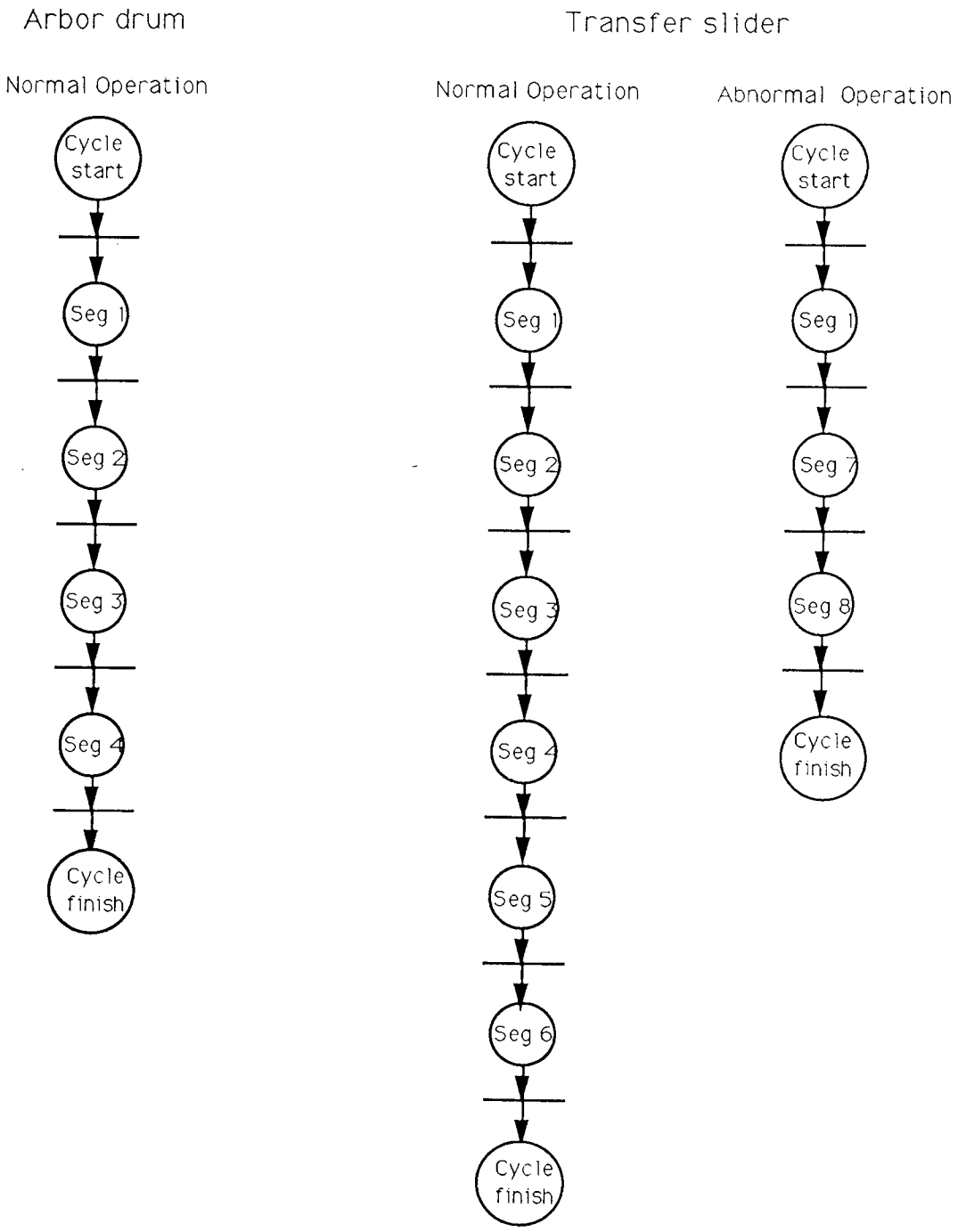


Figure 3.13 Demonstrator segment sequence Petri-net

3.4.6 Independent drive requirements of mechanisms

The motors and actuators used in an IDM machine provide flexibility in the operation of the machine. The drives provide power and motion profiling to each mechanism therefore it is necessary to establish the size and specification of the drive to be used with each IDM.

Electrical specifications associated with controlling the drive may be included. Additional requirements for physical dimensions and wiring may be incorporated if a mechanical layout of the machine is being prepared.

The capture sources for drive requirements include: electrical and electro-mechanical engineers developing the independent drive mechanisms; drive design handbooks and product catalogues of suitable drive systems with associated performance data.

Prototypes of mechanisms and mechanical modules can be used to experiment with different drive configurations. Existing machines may have IDM modules which can be incorporated into new designs, the concept of reusable modules introduced in Chapter 1. Seaward [14] and Fenny [15] both advocate the use of simulation for examining the suitability of drives for an IDM application.

The information necessary to define drives requirements include: The specification of the performance of the drives on the machine, including maximum and minimum operating speed, maximum acceleration, and a measure of the drives ability to drive the load such as power ratio desscribed by Fenny [15]. The drives requirements should also detail repeatability of drive performance, accuracy and reliability. Torque / speed characteristics for the required operating speeds of the machine may be defined. For high torque slow speed operation the inclusion of gearboxes may be necessary, however Seaward [14] demonstrates how direct drive is preferable. Stability and response times of the drive may also be detailed if specific control algorithms are to be implemented. If precise motion requirements are not available then an envelope of required performance may be used.

The main destination of drive requirements will be the database of IDM requirements. The drive requirements are added into the existing database entries for each IDM. Constraints on the operation of the drives will already be present by cross referencing with motion / synchronisation and product requirements held in the database.

Many cross references exist in the requirements database pertinent to the drives requirements. The cross referencing with motion and product requirements is essential to validate that the drive is capable of the specified motions and can operate within the specified tolerances. Cross referencing mechanical size and wiring requirements with physical machine designs allows the feasability of the mechanical layout of the machine to be validated.

An example drive specification for the demonstrator is given in table 3.5 it is derived from drive selection data for the SERC phase one project at Aston produced by Fenny [15]

IDM drive	Max velocity (rad/sec)	Max accel (rad/sec^2)	Power rate (Kw/sec)
Arbor	12.44	800	2033
Transfer	39.98	6840	2542

Table 3.5 Drive specifications for demonstrator

3.5 Summary of notations for specifying machine requirements

The requirements for an IDM machine have been partitioned into a sequence of phases. The first phase identifies the principle parameters of product and process. These parameters are annotated using tables of process requirements, technical drawings and tables of dimension data for intermediate and final product.

The second phase defines product operations necessary to form the various intermediate products which lead to the finished product. Data flow diagrams are used to define and relate the operations in a context diagram of the manufacturing process.

The third phase groups the product operations into mechanical modules and identifies how each of the operations will be achieved by defining appropriate mechanical mechanisms. The mechanisms which require an independent drive are identified. A database record for each of the mechanisms is defined and a DFD of mechanical modules and their interrelationships is created.

The fourth phase defines synchronisation types for each of the mechanisms of the machine. A context diagram of synchronisation relationships is drawn using DFD notation and discrete synchronisation described using Petri-nets.

The fifth phase specifies motion profiles for each of the machine mechanisms and relates these end-effector motions to the mechanisms drive. Motion profile views are generated for each mechanism showing flexibility in motion profiling, extensions for synchronisation type may be included. The motion views are decomposed into sequences of motion segments which are used to create tables of motion profile data. Petri-nets are used to describe sequences of motion segment execution

The sixth and final phase of requirements capture uses the motion profile and mechanical mechanism data to specify drive requirements for the machine. IDM database entries pertaining to drive performance are generated which will be used both for drive selection and for quality assurance of the final product.

3.6 Conclusion

This Chapter has discussed requirements capture for an IDM machine development. It has identified six phases necessary to gather sufficient information to allow the machine development to proceed. These phases capture: user requirements for product and process; product forming operations; mechanical modules to support the product operations; synchronisation of mechanisms; motion of mechanisms; and independent drive requirements of mechanisms.

Methods for partitioning and labelling components of these phases have been proposed. In the product operation phase manufacturing operations have been labelled as fabrication, transportation or safety/quality operations according to function. The product operations are grouped into mechanical modules using the concepts of intermediate-product, common-synchronisation and fabrication sequence partitioning. The logical relationships between mechanisms are defined by describing motion and synchronisation between mechanisms as hierarchical synchronous, parallel synchronous and parallel asynchronous relationships.

Data flow diagrams and Petri-nets have been used as the principle diagrammatic modelling tools to limit the number of notations whilst allowing description of control and data flow.

The requirements data captured is unstructured and unvalidated. The next step in machine development is to validate the requirements by modelling and cross-checking. In particular requirements captured as Petri-nets support validation of requirements by analysis. This is discussed in the next Chapter.

----------//----------

Chapter 4

Validating the machine requirements & creating the machine requirement specification

4.0 Introduction

The previous Chapter described how IDM machine requirements could be captured from a number of sources. In this Chapter these requirements will be used to create a requirements specification for the machine. This specification will be used to drive the design and implementation stages of the IDM controller development and as Anderson [90] states must be correct, complete, unambiguous and feasible.

In order to ensure that the requirements specification document complies with the above constraints the captured requirements must be validated and structured before being included in the machine requirements specification. To achieve this goal methods of requirements validation based upon static and dynamic checking such as Spade [121] or Malpas [122], will be described. Real-time requirements will be examined by using executable requirement specifications to exercise dynamic constraints.

The result of these activities are validated requirements in the form of diagrams, models, database tables and textual descriptions. This data will be structured to form the machine requirements specification.

4.1.1 Requirements validation

Validation defined by Starts [87], is the process of checking for correctness. The initial captured machine requirements will contain errors, omissions, ambiguities and unfeasible performance. To uncover these deficiencies the requirements must be validated, further requirements capture, recasting or alteration may then take place to rectify any deficiencies. Validation is particularly important for the requirements phase since propagation of errors from the requirements specification into future development phases

has serious consequences. Boehm [12] and Gerrard [123] identify the time and cost penalty of removing errors later in the development life cycle.

Validation, the process of checking for correctness, is different from verification. Verification defined by Starts [87], is the process by which consistency of function is established after a transformation of the function. An example of verification would be checks to ensure that an implementation derived from a design must accommodate all the functionality of the design.

4.1.2 Methods of validation

Two classes of validation method, static and dynamic, have been identified by Malpas [122]. Static validation uses redundancy in captured requirements to allow cross-checking of parameters, and standard data formats to check for omissions and syntax errors. The checks identify ambiguous and inconsistent requirement components.

Dynamic validation, described by Birrell [88] requires that models of requirements be constructed. These models can then be exercised to produce details of the dynamic behaviour of the set of modelled requirements. Constraints can be specified to define operational bounds on the variable parameters of requirements. Judgements on the performance of these models can then be taken to check for correct, complete and feasible requirements.

For IDM machines the use of defined constraints on parameters of machine operation allows the dynamic requirements to be checked for breaches of constraints during machine operation, thus highlighting any faults. The constraints on parameters may also be used for real-time performance checks in an implementation.

An additional method of proving correctness is the construction of executable specifications. Wang [115] describes how such specifications can be exercised not as a model but directly, producing the same behaviour as that expected of a direct implementation of the requirements. The executable form of a requirement specification demonstrates the feasibility of the requirements at an implementation level.

A transformation of validation models into executable form provides a route to creating executable specifications. Such an executable specification can be used as a basis for

preliminary controller design. A prototype controller may be developed quickly by adding implementation details to an executable specification.

Two forms of models will be used in the requirements validation procedure. Data flow diagrams will be used to generate context and connectivity diagrams while Petri-nets will be used for modelling specific requirements and for describing control-flow. Occam, a concurrent programming language will be used to build executable forms of the requirements, leading to the development of executable specifications.

4.1.2.1 Database validation

The database which holds the unstructured data of the requirements can be validated using both static and dynamic methods. The captured requirement data for motion is particularly amenable to validation using static methods.

The first static validation method is to uncover ambiguity by cross-checking related requirements. An example from the demonstrator project is the comparison of desired operating speed of the product operation defined in section 3.4.1 of 450 ppm with the maximum speed of the individual IDMs. The duty cycle of the individual IDMs can be derived from the maximum speed graphs of figures 3.7-3.9. The combined graphs can be seen in figure 4.0 this shows a minimum product operation cycle time of 0.7s or 85ppm.

This speed is well below the required maximum speed defined in the process specifications, and the validation process has identified an inconsistency. However for the purposes of demonstrating the IDM flexibility speed of operation is not significant and 85ppm maximum speed will be used in the final implementation.

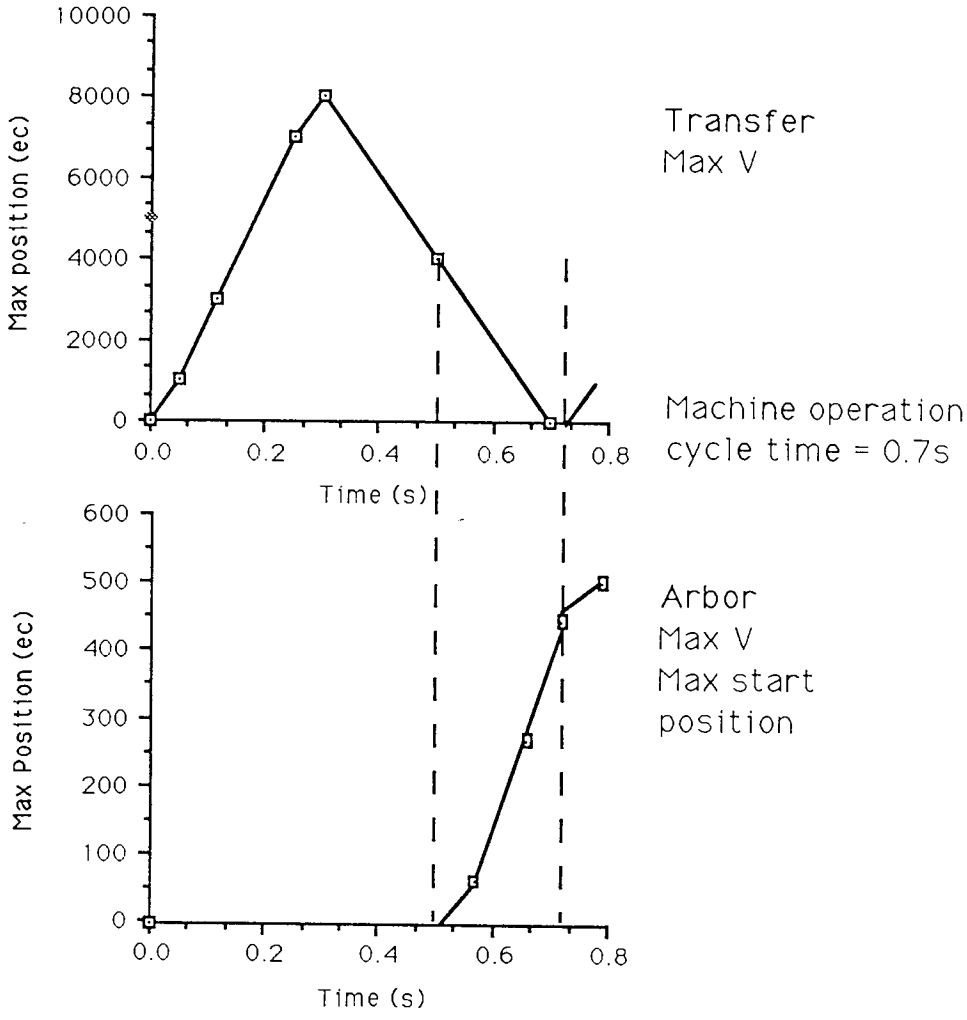


Figure 4.0 Minimum product operation cycle time for demonstrator

The second static validation method uses constraints defined within the requirements. For instance a defined product requirement can be used as a constraint checks on IDM operation. The demonstrator for example has a maximum impact velocity of 2 m/s defined as a product constraint in section 3.4.1. The speed of impact of the transfer mechanism with the product is given in the motion profile table of 3.3 as 20,000 ec/s, where 8,000 ec = 1 revolution of the drive, given a pulley radius of 0.1m this is an impact velocity of :

$$\text{Product impact velocity} = \frac{\text{Maximum Velocity (ec/s)} \times 2 \times PI \times \text{Pulley Radius}}{\text{Encoder counts per revolution}}$$

$$= \frac{20{,}000 \times 2 \times PI \times 0.1}{8{,}000} = 1.57 \text{ m/s}$$

Similarly for the maximum speed of manipulation of the product, maximum transfer velocity is 40,000 ec/s therefore

$$\text{Product max velocity} = \frac{40{,}000 \times 2 \times PI \times 0.1}{8{,}000} = 3.14 \text{ m/s}$$

These are both within specification and therefore pass the validation test.

The incorporation of flexibility into motion profiles to facilitate actuator function changes and speed changes can be problematical for constraints checking. This is due to the many possible profiles the flexibility embodies. However the motion profiles can still be constraint checked. This is achieved by exercising the motion specifications through the envelope of motion profiles defined by the flexibility parameters and referring to the constraints tables for breaches of limits. A similar approach can be taken to the flexibility in synchronisation. However additional care must be taken due to the safety critical nature of synchronisation activities particularly if the sequence of synchronisation is changed by a flexibility parameter.

The relationships between data in the database is dependent upon synchronisation methods and mechanical module partitioning. This makes specification of general tests difficult. The result is that the system developer should identify and carry out the appropriate cross-reference and constraints validation tests individually for each machine development.

4.1.2.2 Validation of DFDs

Data flow diagrams are a powerful tool in describing the processes and flows of data in a system. The annotation of requirements as DFDs allows the functionality of the system to be expressed succinctly. They form a useful tool as the basis of discussions between the systems analyst and the human sources of requirements.

However Birrell [88] describes how DFDs are not amenable to direct validation since there is no structure for exercising and annotating the state of data as it passes through the DFD. Validation of DFDs is possible on the minispecifications of data transforms. If the transform algorithms are available and the scope of input data can be specified then an envelope of possible output data can be generated. This allows constraints checking of the algorithm's output to validate the data transformation minispecification.

4.1.2.3 Validation of Petri-nets

Petri-nets were introduced in Chapter 2 as a tool for modelling concurrent behaviour and control flow. Petri-net descriptions of requirements for synchronisation and motion have been described in Chapter 3, they are direct representations of requirements. In this section these models of synchronisation and motion will be combined to produce a Petri-net model which defines the overall function of the IDMs in a form suitable for validation.

Additional models using Petri-nets can be constructed which model pertinent aspects of the controller requirements. For parallel asynchronous IDMs in particular a Petri-net model of IDM interaction to avoid hardware clash points can be created. This provides a safety model for IDM interaction, and forms the basis of inter-IDM synchronisations for safe and fault-tolerant operation. An example for the demonstrator based on the sequence of activities of each IDM operation is shown in figure 4.1.

The Petri-net is based on the IDM sequence of activities model of figure 3.6. The two FSMs are connected by transitions t2, t3 & t6 which represent safety critical decisions affecting the motion of the IDMs. These decision points are derived from the mechanical requirements of the product operations given in section 2.5 and represent mechanism displacements where a mechanical clash is possible. The inter IDM synchronisations modelled by transitions t2, t3 & t6 allow the IDM controller to react to avoid unsafe

machine operation. In particular the transition t3 forces the transfer IDM to abort entry of the arbor if the arbor drum is moving. The safety decisions are made at points on the motion profile defined by mechanism displacements d2 & d3, thus time is not used in the specification of IDM safety.

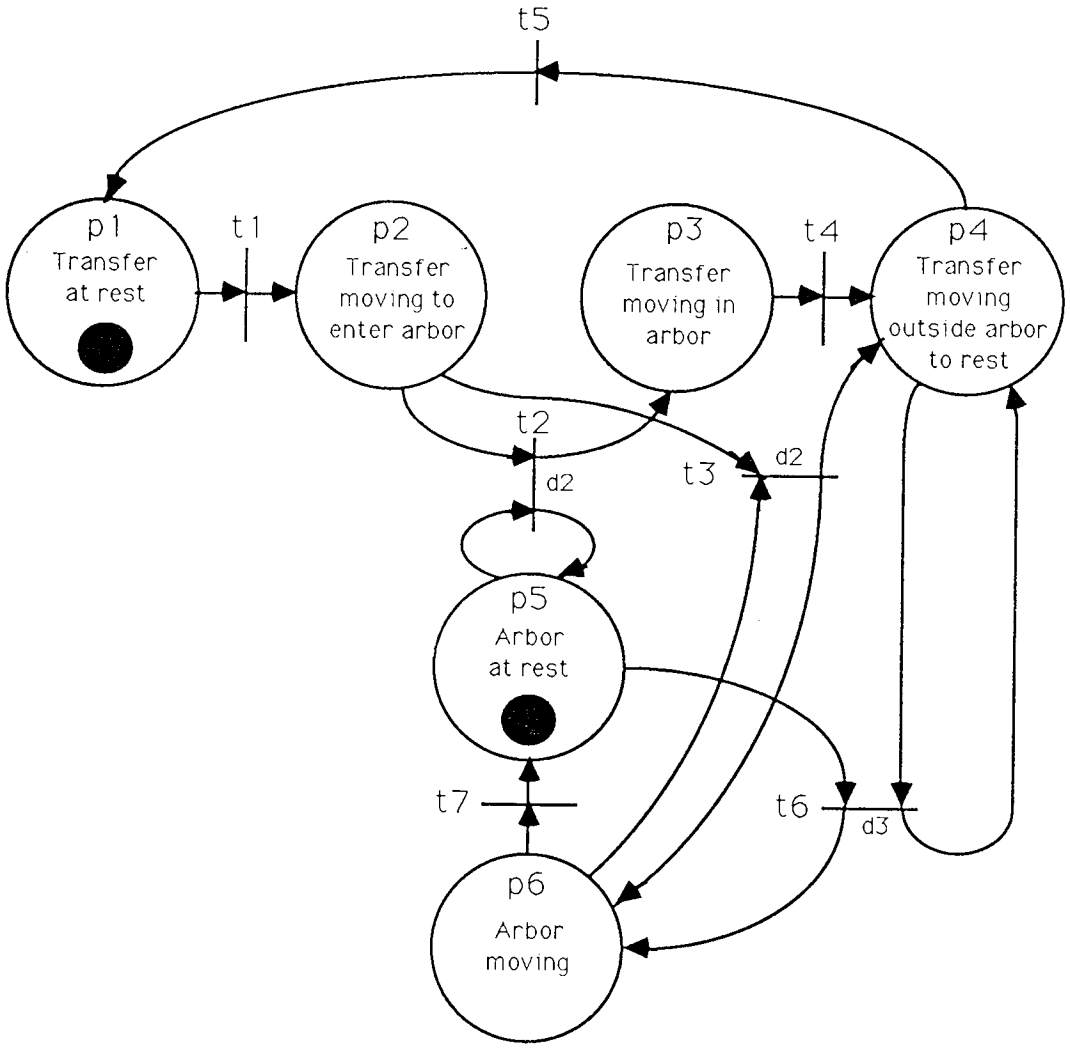


Figure 4.1 Petri-net safety model for demonstrator

The safety and synchronisation models of figure 4.1 and figure 3.8 can be combined to form an IDM interaction model, an example for the demonstrator being seen in figure 4.2. This model specifies interaction between IDMs for all IDM conditions, it can be used to

form the basis of executable prototypes which demonstrate the correctness and feasibility of the IDM interactions.

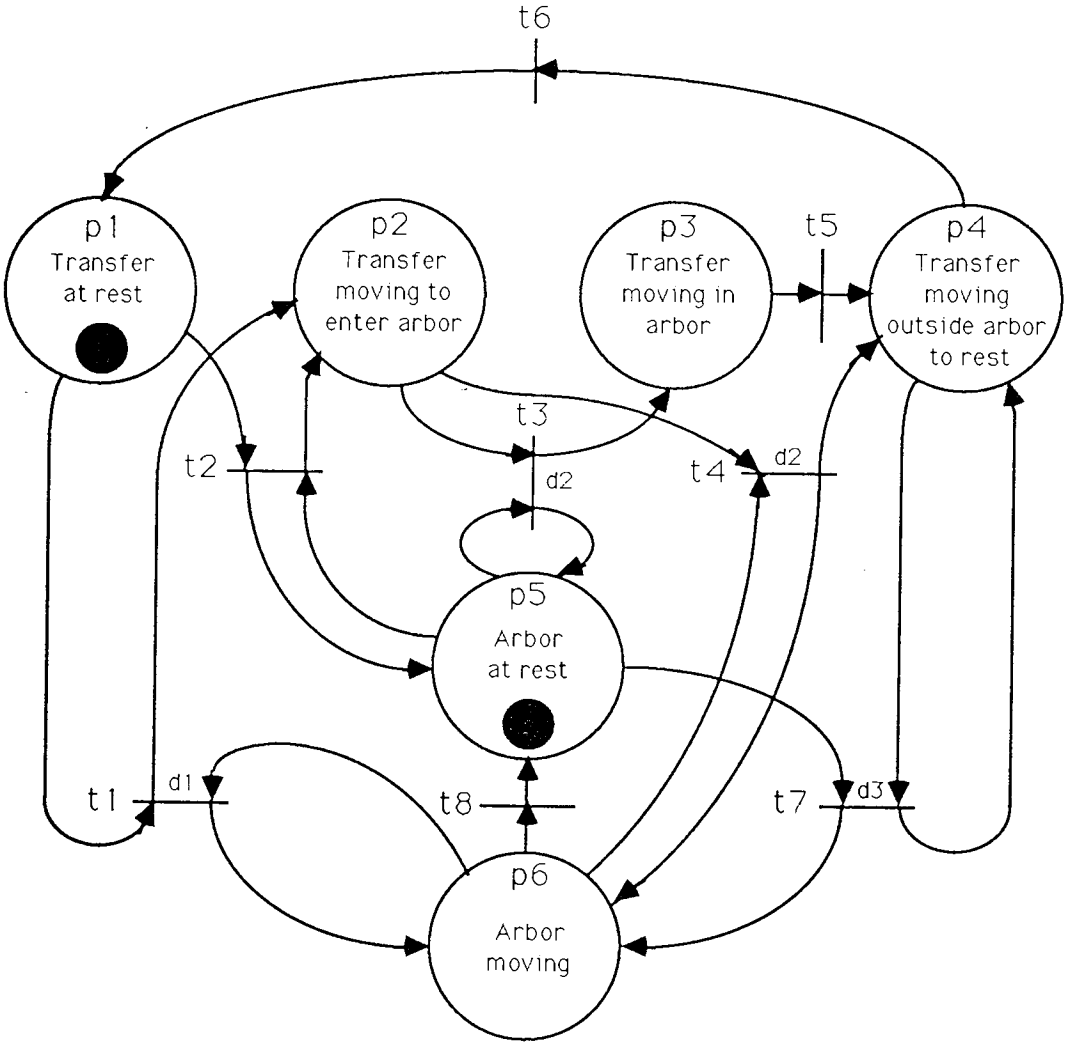
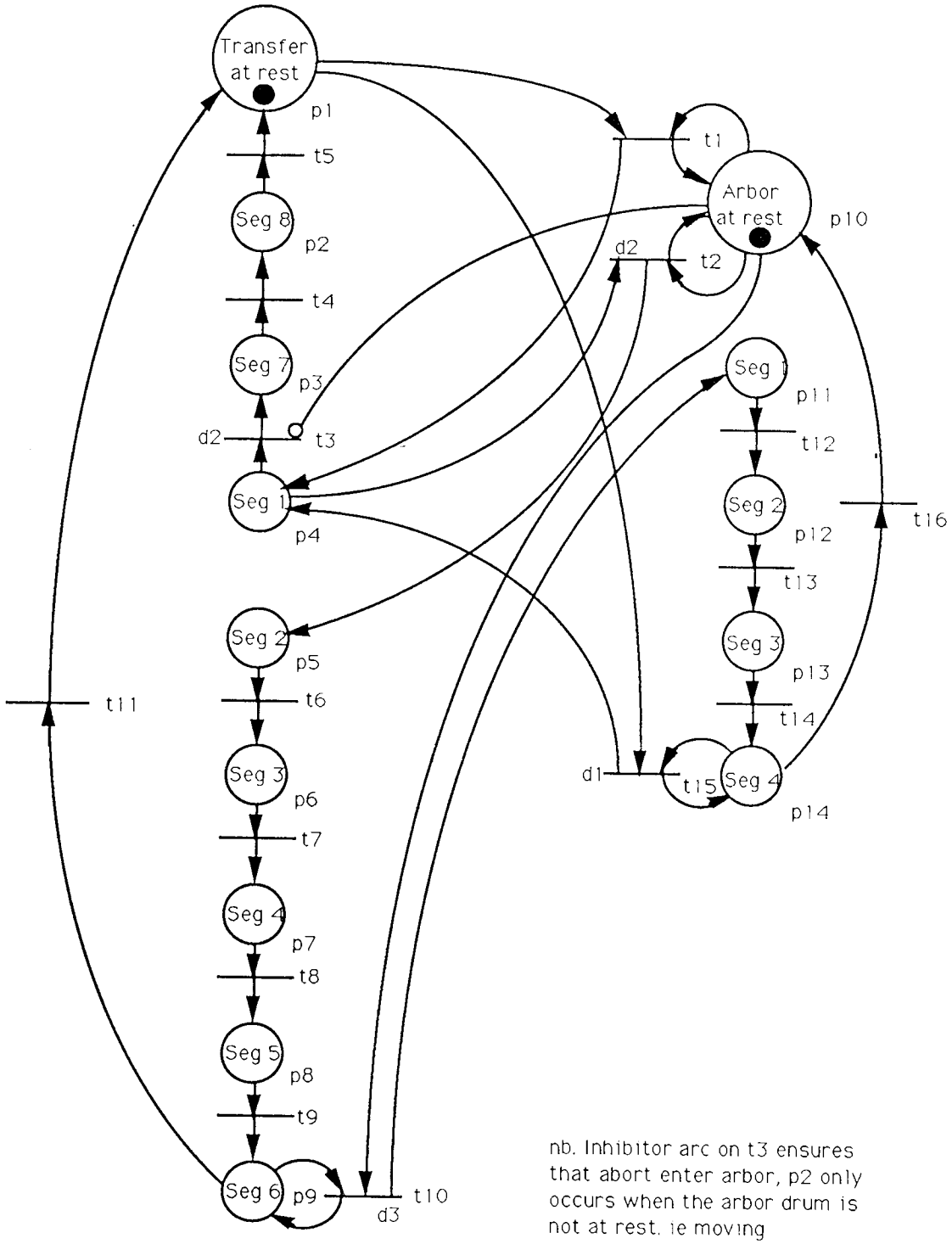


Figure 4.2 Petri-net IDM interaction model for demonstrator

Finally a complete controller model which incorporates the IDM interaction model of figure 4.2 and the motion sequence models defined in figure 3.13 can be created. A Petri-net IDM interaction/motion model for the demonstrator can be seen in figure 4.3.



d1 : arbor at displacement to initiate 'commence transfer move' rendezvous
d2 : arbor at displacement to initiate 'transfer clear to proceed' rendezvous
d3 : transfer at displacement to initiate 'arbor clear to proceed' rendezvous

Figure 4.3 Petri-net IDM interaction/motion model for the demonstrator

In this Petri-net places in the net correspond with states of the IDMs, the Petri-net, using Avenur's [108] definition, describes two Moore type FSMs. The transitions connecting the two concurrent FSMs, representing synchronisations for motion sequence and safety, are detailed in table 4.1 below:

Transition	Represents
t1	initial start condition interlock
t2	transfer entering arbor safety interlock
t3	transfer abort enter arbor safety interlock
t10	arbor commence move safety interlock
t15	transfer commence move interlock

Table 4.1 Motion and safety synchronisations in demonstrator

Transitions relating to a single FSM represent changes in operation of the IDM modelled by the FSM. For figure 4.3 these are changes in the motion segment being executed by the IDM.

The inhibitor arc from p10 to t3 ensures that t2 fires when the arbor is at rest and the transfer has completed motion segment 1.

The controller model forms the most comprehensive Petri-net model of the machine requirements and analysis of this model will result in information on the inter-relationships between safety, synchronisation and motion. The interaction/motion model is suitable for use as the basis for an executable specification of IDM requirements for the demonstrator.

4.1.2.3.1 Exercising Petri-net models

The models which tie together the requirements data into a structured format can be validated using dynamic methods. These methods, described by Willson [118], exercise the models to infer system behaviour and performance. The actual and expected behaviour can be compared and errors uncovered accordingly.

The analysis of Petri-nets to deduce system behaviour using incidence matrices and reachability trees was introduced in section 2.4.4. Petri-net models are particularly useful

in the dynamic validation process. The Petri-net models derived for safety, synchronisation and motion can be transformed from the graphical form to a logical notation suitable for analysis for reachability. The results of the analysis can be examined to see if the Petri-net model of the interaction/motion requirements demonstrate safe fault-tolerant IDM operation. Agerwala [109] describes how the incidence matrix together with the initial marking of the net and the state word describe the operation of the transformed graphical Petri-net.

An analysis for the IDM interaction/motion model Petri-net for the demonstrator serves as a practical example for generating a reachability tree and reasoning about the dynamic behaviour of the Petri-net.

An example analysis for the demonstrator Petri-net model of figure 4.3 follows. The incidence matrix of figure 4.4 is a tabular form of the input and output functions of transitions and places shown in figure 4.5a.

		Transition															
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Place	1	-1	.	.	.	-1	.	.	.	.	.	+1	.	.	.	-1	.
	2	.	.	.	+1	+1	.	.	.	.	.	.	.	.	.	.	.
	3	.	.	+1	-1	.	.	.	.	.	.	.	.	.	.	.	.
	4	+1	-1	-1	.	.	.	.	.	.	.	.	.	.	.	+1	.
	5	.	+1	.	.	.	-1	.	.	.	.	.	.	.	.	.	.
	6	.	.	.	.	.	+1	-1	.	.	.	.	.	.	.	.	.
	7	.	.	.	.	.	.	+1	-1	.	.	.	.	.	.	.	.
	8	.	.	.	.	.	.	.	+1	-1	.	.	.	.	.	.	.
	9	.	.	.	.	.	.	.	.	+1	*1	-1	.	.	.	.	.
	10	*1	*1	@1	.	.	.	.	.	.	-1	.	.	.	.	.	.
	11	.	.	.	.	.	.	.	.	.	+1	.	-1	.	.	.	+1
	12	.	.	.	.	.	.	.	.	.	.	.	+1	-1	.	.	.
	13	.	.	.	.	.	.	.	.	.	.	.	.	+1	-1	.	.
	14	.	.			.	.	.	.	.	.	.	.	.	+1	*1	-1

Transfer slider FSM

Arbor drum FSM

+1 : Insert token in place
-1 : Remove token from place
*1 : Self loop (Remove/Insert token) in place
@1 : Inhibit transition if token in place

Figure 4.4. Incidence matrix for demonstrator IDM interaction / motion model Petri-net

The incidence matrix may be exercised to simulate performance of the demonstrator interaction/motion requirements. As transitions fire tokens appear in different places on the net and this models the changes of state in the requirements. The states of the net & state mnemonics are defined in figure 4.5b & c. A state-transition diagram can be drawn which graphically shows the different paths that the state machine may follow, as in figure 4.6. Unsafe instances of the state-word may be flagged and an occurrence of a prohibited state indicates the presence of an error in the requirements. A knowledge of the operation of the system is required since judgements on dynamic performance to identify safe, unsafe and incorrect states, is necessary.

Transitions

		Input function	Output function
t1	:	{p1,p10}	{p4,p10}
t2	:	{p4,p10}	{p5,p10}
t3	:	{p4,NOT(p10)}	{p3}
t4	:	{p3}	{p2}
t5	:	{p2}	{p1}
t6	:	{p5}	{p6}
t7	:	{p6}	{p7}
t8	:	{p7}	{p8}
t9	:	{p8}	{p9}
t10	:	{p9,p10}	{p9,p11}
t11	:	{p9}	{p1}
t12	:	{p11}	{p12}
t13	:	{p12}	{p13}
t14	:	{p13}	{p14}
t15	:	{p1,p14}	{p3,p14}
t16	:	{p14}	{p10}

Figure 4.5a Transition function definition for demonstrator IDM interaction/motion model Petri-net

State word (p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12, p13, p14)

State representation of places in Petri-net
p1: transfer at rest in home position
p2: transfer returning to home position (abnormal operation), segment 8
p3: transfer decelerating to abort arbor entry (abnormal operation), segment 7
p4: transfer accelerating, motion segment 1
p5: transfer decelerating to contact pack, motion segment 2
p6: transfer accelerating into arbor, motion segment 3
p7: transfer decelerating to rest - fully inserted in arbor, motion segment 4
p8: transfer accelerating out of arbor; motion segment 5
p9: transfer clear of arbor, decelerating to rest, motion segment 6
p10: arbor at rest
p11: arbor accelerating, motion segment 1
p12: arbor moving at constant velocity, motion segment 2
p13: arbor moving at constant velocity, motion segment 3
p14: arbor decelerating to rest, motion segment 4

In places p6, p7 and p8 the transfer and arbor kinematic envelopes overlap with the associated possibility of mechanism clashes.

Figure 4.5b. State word definition for demonstrator IDM interaction / motion model Petri-net

Reachability tree, state word mnemonics

s1: (1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0)	- initial state
s2: (0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0)	- transfer commence move
s3: (0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0)	
s4: (0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0)	
s5: (0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0)	
s6: (0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0)	
s7: (0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0)	
s8: (0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0)	- arbor commence move
s9: (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0)	
s10: (0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0)	
s11: (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0)	
s12: (0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0)	
s13: (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0)	
s14: (0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1)	
s15: (1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1)	
s16: (0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1)	- transfer commence move
s17: (0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1)	- transfer abort enter arbor
s18: (0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0)	
s19: (0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1)	
s20: (0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0)	

Figure 4.5c. State word mnemonics for demonstrator IDM interaction / motion model Petri-net

The state-transition diagram for the demonstrator shown in figure 4.6 has two error transitions flagged (t11 & t16). The error is detected since the performance of the model is not the required performance of the control system.

In the demonstrator model the solution to correct the error is the same in both instances. Consider the error of transition t11 firing in state s7, if this occurs the arbor will not move and the transfer will repeatedly enter the stationary arbor. This is incorrect operation. The remedy is to impose firing constraints on transition t11, to ensure t10 fires first - t10 enabling the arbor to move. This can be done using inhibitor arcs to prevent the firing of t11 (p10 successfully inhibits t11), the method used is the same for the interlock safety decision of t3.

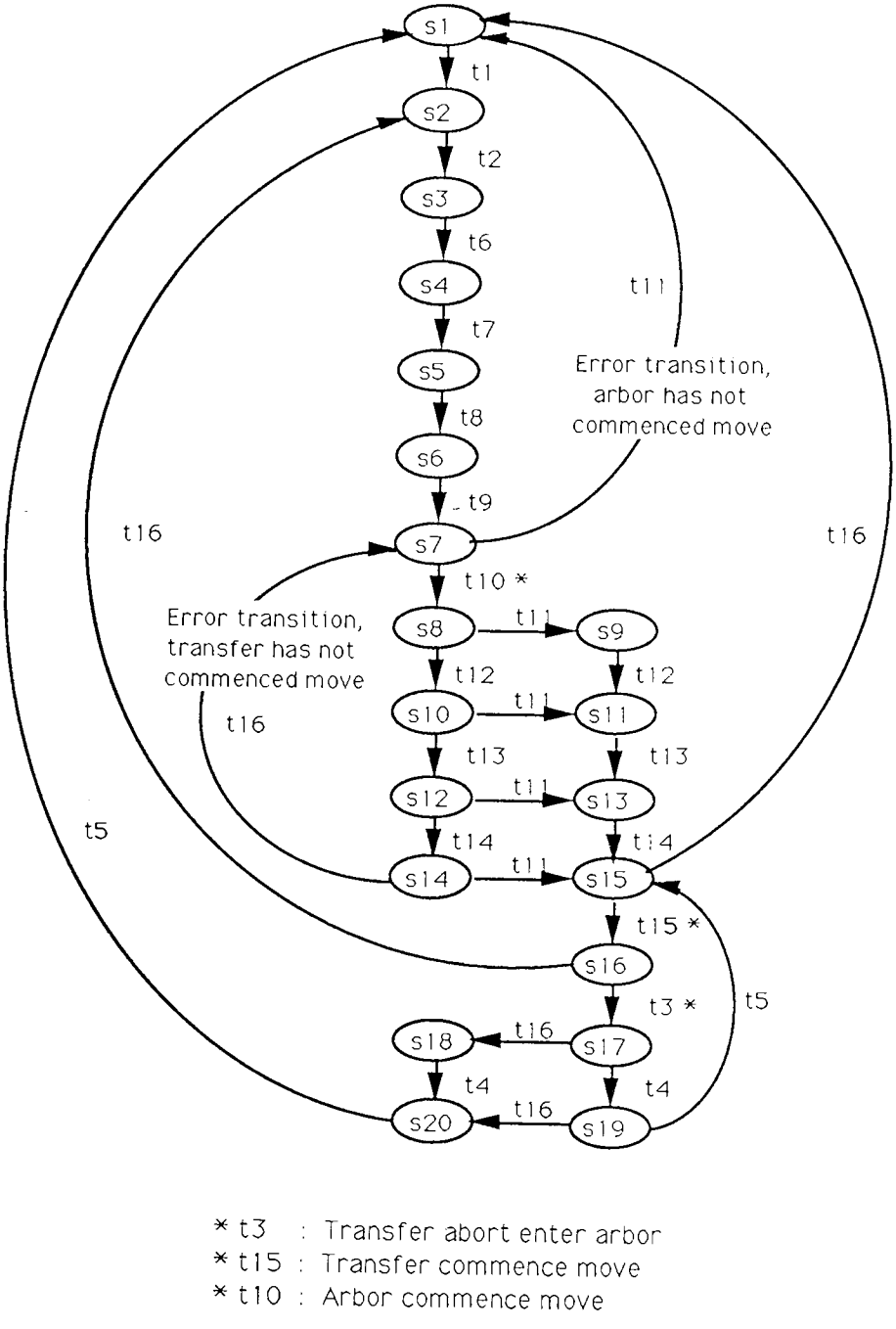


Figure 4.6 State reachability diagram for demonstrator Petri-net of figure 4.3

Other methods of scheduling transitions, such as timed Petri-nets introduced by Merlin [124] and generalised by Holliday [125] or the temporal Petri-nets of Sagoo [126], can provide the required performance at the expense of greater model complexity.

The physical firing of synchronisation transitions in the Petri-net model for the demonstrator relate to motion displacements in the IDM motion requirements for the machine. Defined flexibility in the motion profiles may cause abnormal events to occur. For example if the arbor is moving much more slowly than the transfer mechanism, which is defined motion flexibility, then the interlock to abort the transfer will enable on every cycle due to the disparate cycle times of the individual IDM motions.

The flexibility of the motion is not at fault, the problem lies with the possible implementation of flexibility between the IDMs, since running the arbor more slowly than the transfer while possible is not an intended mode of operation. An example of the demonstrator implementation exhibiting abnormal operation due to inappropriate use of flexibility in IDM operation is given in section 6.2.

Petri-net models have some drawbacks. Peterson [112] shows how the larger the size of the original Petri-net model the more susceptible to state explosion the state-transition diagram becomes. A large number of Petri-net states makes the results of the state-transition exercise difficult to assimilate and incorrect operation or unsafe states hard to identify. Large Petri-nets are unweildy in both development and analysis. However net reduction can be used, this remodels the Petri-net into a smaller subset of the main Petri-net. Analysis of particular requirements may then proceed on this reduced size Petri-net.

4.1.2.4 Executable specifications

The functional models may be used to construct executable specifications which can be used to both exercise models to highlight dynamic performance errors and enforce compilation of the executable form of the requirements specification to expose consistency and completeness errors.

To create an executable specification a notation which allows the description of real-time, concurrent behaviour allowed by Petri-net and DFD models is required. The notation must be supported by a method of generating executable code which can be run on an appropriate computer system to produce information on the dynamic operation of the requirements specified.

To describe requirements for IDM machines, the concurrent programming language Occam [63][80] has been used. This language can be applied as an executable specification notation since it supports real-time operation (TIMERS), concurrent

operation of many processes (PAR), sequential operation of single processes (SEQ), races between concurrently executing processes (ALT), decisions (IF), iteration (WHILE) and communications between cooperating processes (!-put, ?-get) [116][127]. The language is also compact and succinct, this allows requirements to be expressed concisely and interactions between concurrent processes, such as IDMs, to be annotated in a visible and understandable form. An Occam executable specification for the inter IDM synchronisation for the two IDM motion/synchronisation Petri-net of figure 2.10. is given in figure 4.7.

In figure 4.7 two individual FSMs are denoted by two sequential processes operating concurrently. Occam uses the PAR and SEQ primitives to achieve this. States of the FSM are represented by a sequence of comments or functions. Petri-net transitions between FSMs are represented by inter process communications using 'put' (t1 ! go) and 'get' (t1 ? any) primitives. Transitions internal to a FSM are not explicitly represented but can be implied as being present between states.

The use of executable specifications as a basis for implementing IDM controllers will be discussed further in Chapter 5. A description of an Occam specification and it's use in developing a fast prototype controller is given in Chapter 6. In addition a description of a fast-prototyping exercise based on Occam executable specifications of synchronisation for the demonstrator can be found in the "The specification and fast-prototyping of a distributed real-time computer control system for a modular independently driven high-speed machine" a paper by the author which is included in the appendices.

```
-- two IDM synchronisation model for figure 2.10

-- inter IDM Petri-net transitions modelled by Occam channels
CHAN OF ANY t1,t3
int go EQU 1:

-- start arbor and transfer IDMs concurrently
PAR                                          -- define concurrent processes
    SEQ                                      -- transfer process
        -- transfer stationary               -- start condition for transfer
        WHILE TRUE
        SEQ
            t1 ? any                         -- 'get' awaits arbor start
            -- transfer moving into arbor
            -- transfer moving out of arbor
            t3 ! go
            -- transfer stationary

    SEQ                                      -- arbor process
        -- arbor rotating                    -- start condition for arbor
        t1 ! go                              -- 'put' starts transfer motion
        WHILE TRUE
        SEQ
            t3 ? any                         -- 'get' awaits transfer clear
            -- arbor moving
            t1 ! go                          -- 'put' starts transfer motion
            -- arbor stationary
```

Figure 4.7 Example Occam executable specification

4.1.3 Summary of validation tools

The principle validation tools for IDM machine requirements are static and dynamic checks on machine database entries, dynamic modelling of system requirements using Petri-nets and executable specifications of modelled requirements. The different validation tools, the requirements they validate and their purpose are summarised in table 4.2.

The discovery of errors makes further requirements capture and validation necessary. When all the requirements data and models have been validated satisfactorily the next phase, requirements specification, can be initiated.

Tool	Applied to	Validates
Cross-reference tests	Product Dbase Process Dbase IDM mechanics Dbase IDM motion Dbase	Consistency
Constraints tests	Product Dbase Process Dbase IDM mechanics Dbase IDM motion Dbase	Correctness
Petri-net	Safety model Synch model Motion model	Correctness Feasibility Completeness
Executable specifications	Safety model Synch model Motion model	Correctness Feasibility Completeness

Table 4.2 Summary of validation tools

4.2 Machine requirements specification

The thesis proposes a machine requirements specification document structured as a hierarchical decomposition of functional requirements for the IDM machine computer controller. The highest levels of the hierarchy describe machine operation and product to be manufactured. The lowest levels describe the motion profiles and drive specifications of individual IDMs in the machine.

The requirements specification will be based upon the deliverables of the requirements capture phases identified in Chapter 3. These deliverables undergo validation procedures to ensure correct, complete, unambiguous and feasible requirements before being included in the machine requirements specification document.

The capture stages used a database to hold unstructured requirements and models to structure and relate requirements. This format is retained in the specification document. The document is readable by various users due to it's hierarchical decomposition of complexity. Thus an overview of the requirements can be gained by studying higher levels of the specification whilst lower levels have sufficient detail to support controller design.

The document provides the basis for the design and implementation of the IDM machine computer controller.

4.2.1 Formats for specification

Textual description of requirements are used at the top level of requirements specification. Product, process and mechanical requirements may use technical drawings to describe components, assemblies, the complete product, product flow and mechanical components in the machine. These will be supported by database entries to hold unstructured requirements data, common data formats and templates being used to impose consistency.

Data flow diagrams and associated minispecifications can be used to describe hierarchy of control in the machine, relationships between machine operations and IDMs, and material flow.

Petri-nets can be used to describe rigorous safety and synchronisation requirements and control flow. Analysis of the performance of these Petri-nets will also be given to facilitate reasoning about the dynamic operation of the Petri-nets and the requirements they model.

Occam code can be used as a minispecification notation for algorithms and to develop executable specifications of pertinent requirements.

4.2.2 Content of machine requirements specification

The content of the machine requirements specification will be based on the validated deliverables of the sequence of capture phases defined in Chapter 3. Accordingly a table of capture phase, deliverable and format follows

.

Requirements Capture phase	Deliverable	Format
Product specification	Product description Components Assemblies	Text Technical drawing Database
Process specification	Process parameters Assembly sequence	Database DFD
Product operation	Operation description Operation definition Operation relations context diagram Material flow	Text Technical drawing Database DFD
Mechanical modules & IDM	Mechanical partitioning Module definition IDM definition	Text Technical drawing DFD Database
Synchronisation	IDM/CDM synchronisation classification Synchronisation hierarchy Safety synchronisation Inter IDM synch	Text Database DFD Petri-nets
Motion	Motion profile view + flexibility + alternative profiles Motion profile components Motion profile sequence	Technical drawing Database Petri-nets
Independent drives	Drive sizing Drive specification Operating parameters	Text Technical drawing Database

Table 4.3 Requirements capture phase deliverables

The principle diagrams which describe the system are: a context diagram of the physical machine showing product flow, mechanical modules, IDMs within modules and their connection; and a context diagram of hierarchical control showing logical connectivity within the machine.

4.3 Conclusions

This Chapter has discussed methods of validating captured machine requirements in order that a complete, correct and feasible set of requirements be presented in the form of a machine requirements specification document.

Validation analysis of the demonstrator requirements using cross-referencing and constraints testing on the machine database has been introduced. The creation and analysis of Petri-net models of machine requirements has been presented.

The requirements have been brought together into a machine requirements specification document using the validated deliverables of the sequence of requirements capture phases introduced in Chapter 3. This document is a hierarchical decomposition of machine requirements. The highest levels of the hierarchy describe machine operation and product to be manufactured. The lowest levels describe the motion profiles & drive specifications of individual IDMs in the machine.

There are several problems associated with validation of requirements. One of the most significant is the lack of automated procedures for requirements testing. This results in the system developer being responsible for the thoroughness of the validation process. Different representations of requirements may make checking for consistency between representations difficult, however the use of Petri-nets as a notation for combining and structuring requirements into coherent models has been demonstrated. Inappropriate use of flexibility in IDM requirements have been shown to create implementation problems, if machine safety is not compromised these flexibility difficulties may be overcome by careful implementation strategies.

In order to annotate some requirements design decisions often need to be taken, this may constrain the computer controller design. For example annotating an IDM motion as following a master profile generated outside the IDM will require close-coupling of profile-generator and IDM controller in an implementation.

The design of the IDM computer controller is based on the machine requirements specification document and is discussed in Chapter 5.

----------//----------

Chapter 5

The design of a computer controller which embodies the machine requirements

5.0 Introduction

This Chapter discusses the design of the computer controller for an IDM machine. It uses the machine requirements specification of Chapter 4 to develop the design. In particular an essential model of a generic IDM machine controller, constructed using the data flow concepts of DeMarco [103] and the real-time extensions of Ward & Mellor [105], is detailed together with an implementation model developed for the distributed control demonstration machine proposed in Chapter 1.

The aim is to demonstrate how the machine requirements specification deliverables are used in developing the controller design. In addition pertinent aspects of practical real-time control systems which constrain system design are discussed.

5.1 Software development

The thesis will concentrate in this Chapter on software development for the IDM computer control system. This is due to the tendency for machine manufacturers to use proprietary computing hardware but to develop control software in-house.

As a result the hardware described has been selected by it's suitability for real-time, distributed processing and it's real-world interfacing capability. Many standards exist, such as Multibus, VME, and Bitbus, but the Transputer standard has been chosen since it has the facility to support multiprocessing and to directly execute concurrent Occam code without the need for additional operating systems. Occam has been selected as a language due to it's support for concurrency, it's close representation of Petri-net finite state machines and capability as a concise specification/prototyping language.

5.2 Control system design

Project goals of modularity, reusability and safety require that constraints be applied during the design process. In particular real-time constraints are prevalent and require technology based design to provide a suitable implementation. This technology-led approach is at odds with the problem-oriented approach advocated by McMenamin [89]. To achieve a feasible system design when faced with technological constraints in a predominantly problem-oriented design environment, prototypes of controller designs and proof of concept IDM test-rigs are proposed. Since these prototypes can be used for validating the suitability of technology for a particular purpose [87].

Modularity of computer hardware requires distributed processing based on processing nodes connected by a communications network. Partitioning to minimise interfaces and connectivity is appropriate in reducing this real-time communications overhead. Due to time constraints and hardware limitations this may require the grouping of related IDM controllers into tasks on a single processor, such as the multi-axis motion generation for the Molins maker machine.

Modular software also requires that interfaces be created. Regulating these interfaces by rigorous definition of module function and connectivity associated with minimal coupling and strong cohesion between modules will be used to achieve a robust design. The use of defined interfaces also facilitates testing by delineating functions whilst also providing a visible interface which can be used to monitor software performance.

The partitioning of the controller functions will follow the hierarchy of control identified in the specifications to provide a close match between required and implemented controller functions.

5.3 Tools

As introduced in Chapter 2 and expanded upon in Chapter 3 & 4 the Ward-Mellor real-time software design method, based on developing an essential system model, will be used to derive a generic IDM machine controller. DFDs are the principle tool for detailing the design.

Database design notation will be restricted to data declaration with comments. Pseudo-code minispecifications and textual descriptions will be used for detailing DFD

transforms. Prototypes of real-time and safety critical control system components will be described using Petri-nets and implemented in Occam.

Where possible Occam programming conventions will be used in database and minispecification definitions to allow a close correspondence between database specifications, design and implementation. This has been influenced by the use of Occam and the Transputer as the computing development environment for prototypes, executable specifications and the demonstrator implementation.

5.4 Essential model for a generic IDM machine controller

In order to generate an essential model for a generic IDM machine controller a series of design steps must be undertaken. These steps take requirements from the machine functional requirements specification document as input data for the design.

5.4.1 Environmental model

The first step is to develop an environmental model which describes the environment in which the system will operate. Ward & Mellor [105] describe how this is achieved by describing the boundary that separates the system from it's environment in a context diagram and by describing the external events to which the system must respond in an event list. An event list and general context diagram for an IDM machine can be seen in figures 5.1 and 5.2.

The Ward Mellor event classification denotes the source of the event. A flow-direct event is initiated by the presence of data on a data flow; a flow-indirect event requires an event recognition activity; a temporal event is time dependent. The events of an IDM machine relate to asynchronous operator commands ('New user command' & 'New MIS command'), to physical events in the machine ('New IDM cycle' - for completion of one IDM operation, 'New module cycle' - for completion of one manufacturing module operation and 'New machine cycle' - for completion of all module operations resulting in a finished product), and to temporal events associated with machine elements at given sample rates ('Machine clock', 'Module clock', and 'IDM clock').

Event	Classification
New user command	flow-direct
New MIS command	flow-direct
New machine cycle	flow-indirect
New module cycle	flow-indirect
New IDM cycle	flow-indirect
Machine clock	temporal
Module clock	temporal
IDM clock	temporal

Figure 5.1 Event list for a generic IDM machine

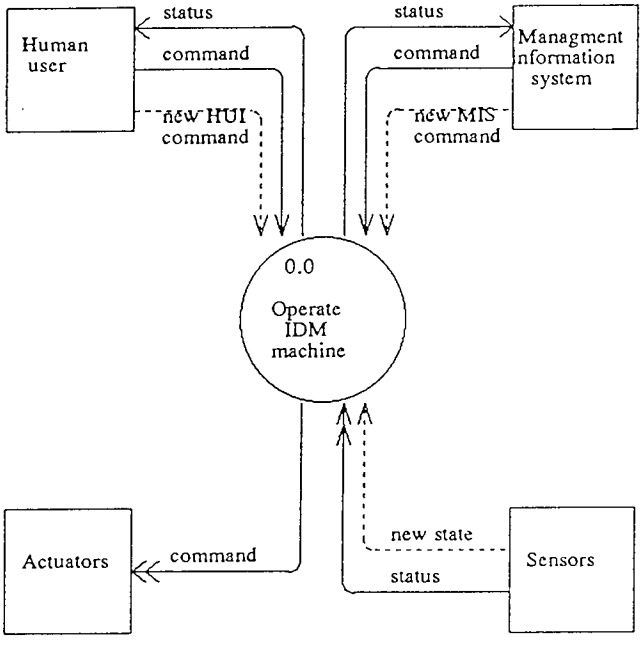


Figure 5.2 Environmental model for a generic IDM machine

The context diagram shows a coarse grained representation of the machine, it contains details of controller function, physical mechanisms and interfaces to users and other systems. The tangible annotation of controller, plant and external interfaces details the boundary of the computer controller in relation to plant and external systems and provides a basis for top-down decomposition of controller function.

The list of external events is refined by detailing an event-action list of expected machine responses prior to the creation of the first level of the behavioural model.

Event	Action
New user command	validate & perform if appropriate
New MIS command	validate & perform if appropriate
New machine cycle	perform machine speed related (asynchronous) logic
New module cycle	perform module speed related (asynchronous) logic
New IDM cycle	perform IDM speed related (asynchronous) logic
Machine clock	perform machine time related (synchronous) logic
Module clock	perform module time related (synchronous) logic
IDM clock	perform IDM time related (synchronous) logic

Figure 5.3 Event list for a generic IDM machine

5.4.2 Behavioural model for generic IDM machine controller

The behavioural model is an expansion of the operate IDM machine data transform of figure 5.2 which details data transforms whose actions provide responses to the events identified in figure 5.1 & 5.3.

5.4.2.1 Operate IDM machine DFD

A generic IDM machine behavioural model, based on hierarchical decomposition of machine function, obtained from the machine context diagram of the requirements specification can be seen in figure 5.4.

This diagram is based upon the logical hierarchy of control identified in the machine specification, which has been partitioned to identify levels of control within the machine.

The three levels of hierarchy in the controller are denoted by data transforms. The lower transforms being outlined to denote multiple instances of the transform activities. The partitioning follows the logical grouping of a set of interacting IDMs into mechanical modules, to perform product operations, and a group of mechanical modules to fulfil the manufacturing task. The partitioning also follows Saridis' [70] heuristic of increasing data rates with decreasing responsibility since the IDM control transforms will operate at higher sample rates than the machine control transform. In addition the closest coupling between IDMs will occur within a single machine module.

The levels are partitioned by data stores which represent well-defined, visible, communications paths between levels of the hierarchy.

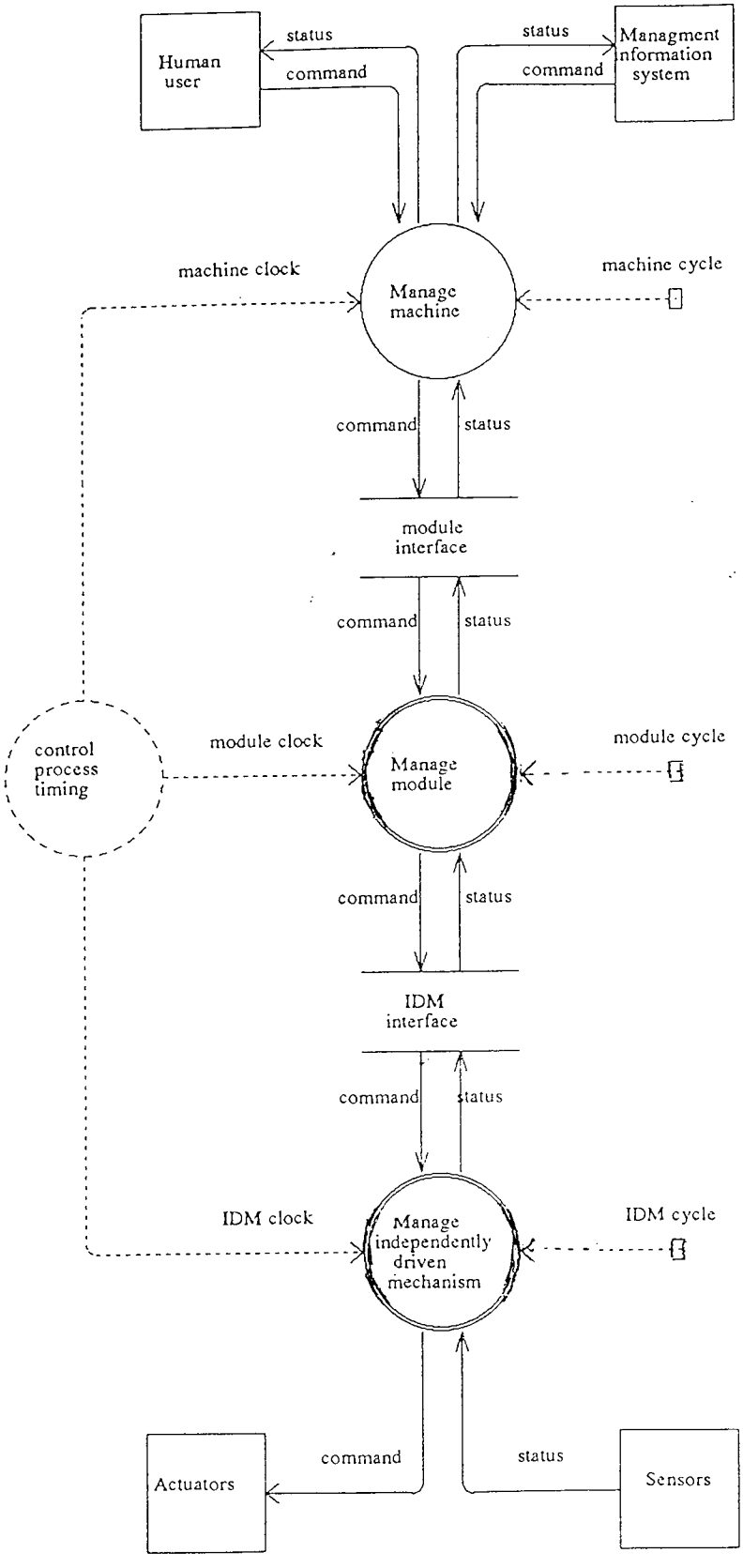


Figure 5.4 Behavioural model for a generic IDM machine controller : Operate IDM machine

The propagation of commands serves different purposes down the control hierarchy. At the top level these commands relate to modes of machine operation such as STOP & RUN. At the intermediate level commands convey mode and possibly motion and synchronisation data. At the lowest level commands relate to motion and synchronisation of the cooperating IDMs.

Data stores and data transforms within a level will be detailed in the next level of DFD modelling.

5.4.2.2 Manage machine DFD

The diagram in figure 5.5 describes the custodial activities which allow commands and status data to be passed up and down the hierarchy of control. The 'validate commands' transform accepts data from higher levels and passes on the commands to the control transform if appropriate. The status process takes data from local status and data from lower levels in the hierarchy. So unprocessed data can be passed from the lowest level to the highest in the hierarchy if appropriate.

The control process transform takes status data from the lower level, which may contain motion and synchronisation data, and control data from the higher data area which may contain mode of operation (start, stop, flexibility) commands as well as motion/synchronisation data. The action of the control process transform supports the events identified in figure 5.1. A further DFD decomposition demonstrates the required functionality.

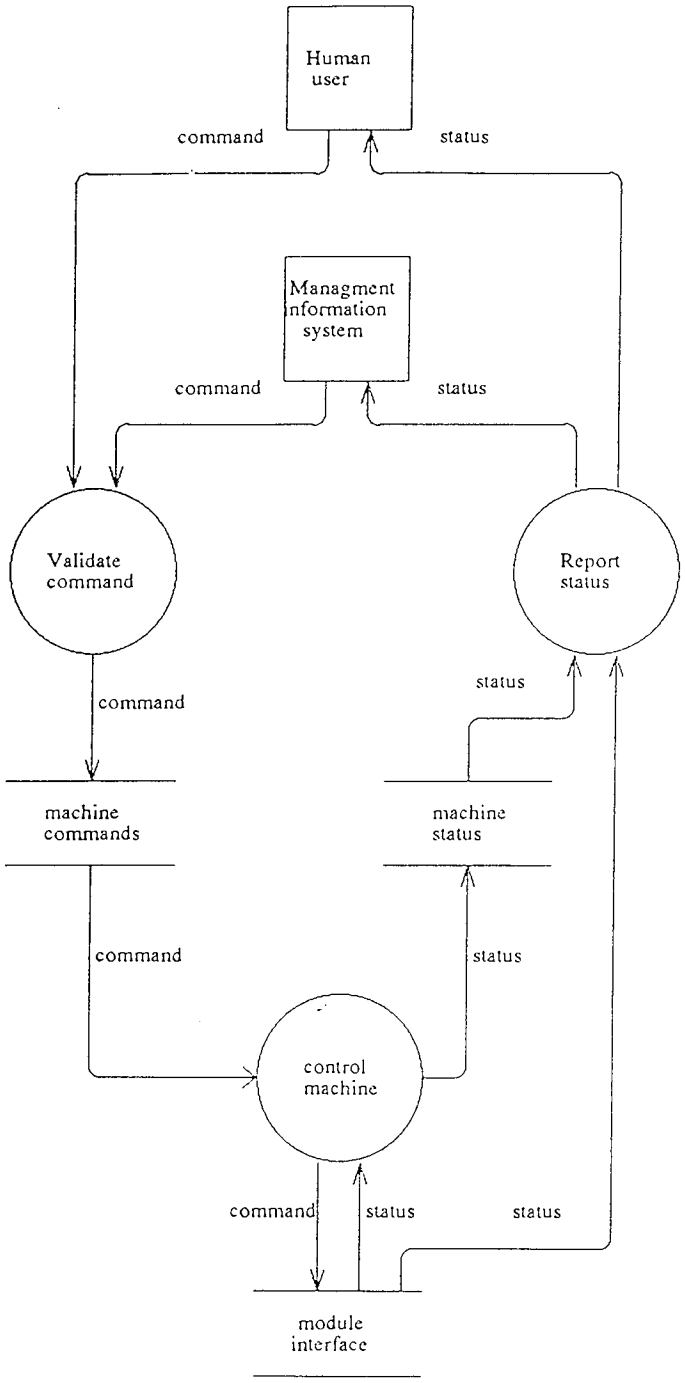


Figure 5.5 Behavioural model : Manage machine DFD

5.4.2.3 Manage module DFD

The diagram of figure 5.6 shows the same custodial functions as detailed in figure 5.5 for the machine but applied to the module control. The data transforms perform the same validating function but the sources of data are internal controller data stores, the module interface and IDM interface.

5.4.2.4 Manage IDM DFD

The control IDM DFD of figure 5.7 shows the custodial functions of figures 5.5 & 5.6 applied to IDM control. The data stores used by the transforms detailing connection to the physical plant (actuators, sensors) and the IDM interface.

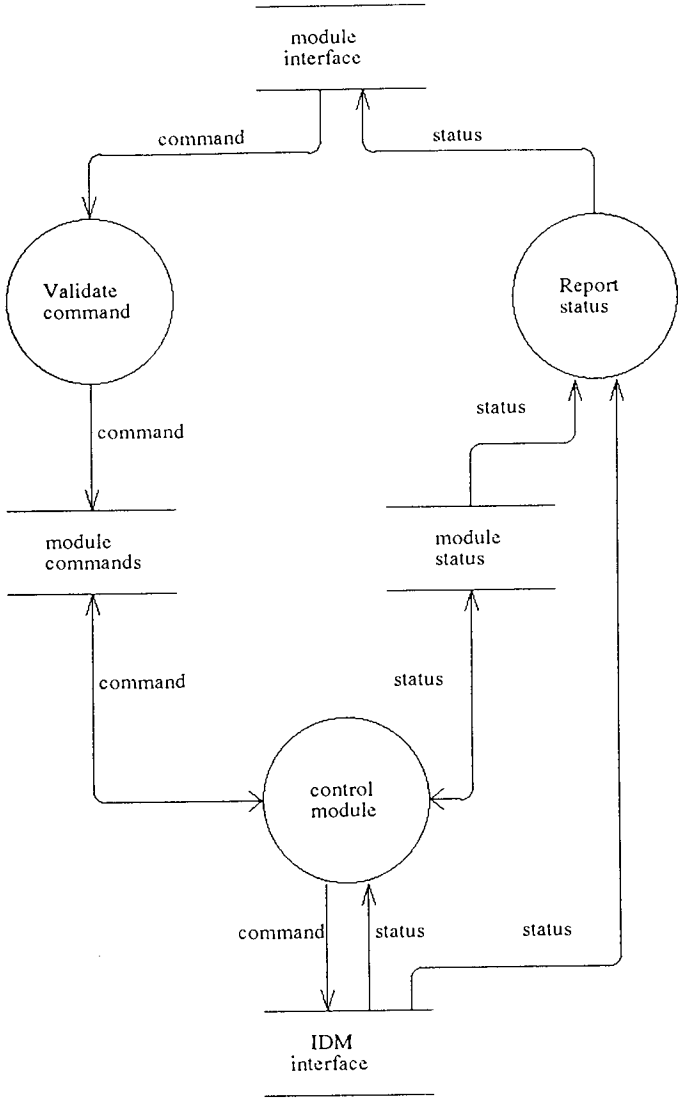


Figure 5.6 Behavioural model : Manage module DFD

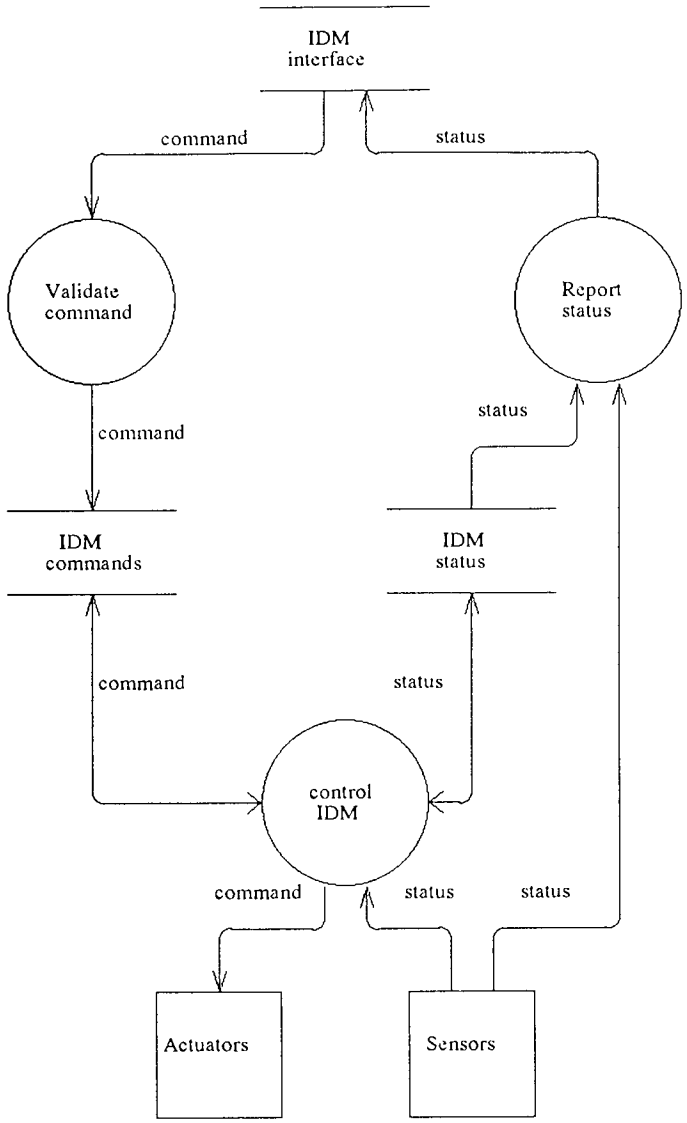


Figure 5.7 Behavioural model : Manage IDM DFD

5.4.2.5 Control machine timing CFD

The inclusion of a control transform indicates the scheduling of controller operation by events that are generated internally. The clock function prompts the three data transforms to which it is connected to perform 'samples' on the data held in the data stores. Sample rates increase down the hierarchy of control in the machine. An Occam pseudo-code minispecification of the process showing the three output prompts as channel communications is shown in figure 5.8.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--      MACHINE, MODULE & IDM control timing pseudo code
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

  --- OCCAM clock channels
 TIMER machine.timer, module.timer,  IDM.timer
 VAL INT machine.interval 4000
 VAL INT module.interval 1000
 VAL INT IDM.interval 100:

 SEQ
  -- set up next events

  machine.timer  ? machine.time     -- current machine time
  module.timer   ? module.time      -- current module time
  IDM.timer      ? IDM.time         -- current IDM time

  WHILE timer.enabled
   --  monitor for events
   ALT
    machine.timer ? AFTER (machine.time+machine.interval)
     SEQ
      machine.timer ? machine.time
      -- send event  to machine controller
      machine.clock ! prompt
        SKIP
    module.timer ? AFTER (module.time+module.interval)
     SEQ
      module.timer ? IDM.time
      -- send event  to module controller
      module.clock ! prompt
        SKIP
    IDM.timer ? AFTER (IDM.time+IDM.interval)
     SEQ
      IDM.timer ? IDM.time
      -- send event  to IDM controller
      IDM.clock ! prompt
        SKIP
```

Figure 5.8 Occam pseudo-code minispecification of control machine timing activity

5.4.2.6 Machine process logic

The diagram of figure 5.9 shows the machine level data transforms which deal with supervision of multiple manufacturing module processes. The principal role of this DFD is to manage the mode of operation of the machine and the user interface.

The 'machine process logic' data transform responds to events from the user interface and the module interface to control the set of manufacturing modules. The set of commands issued to the module interface is dependent upon the type of machine operation. A machine with a large number of continuously synchronised IDMs may require a software master motion which incorporates synchronisation to be transmitted to a number of modules. Intermittent motion modules may require only mode commands, synchronisation coming from other modules. Some IDM actuators may require control in circumstances that only the highest control levels can detect, the initiation of a response to these circumstances requires machine-level commands to the IDM passed via the appropriate module. For example a product-eject mechanism may require knowledge of machine state outside it's physical module.

The mode of operation transform is the primary mechanism for controlling the machine in response to user commands and module status. A FSM representation allows deterministic response to commands in different machine operating modes. Thus commands in the system can be classified as mode of operation commands (such as RUN, STOP commands) or process logic commands (such as flexibility commands for motion profile changes).

Detection of machine state events is accommodated by the 'generate machine cycle' control transform.

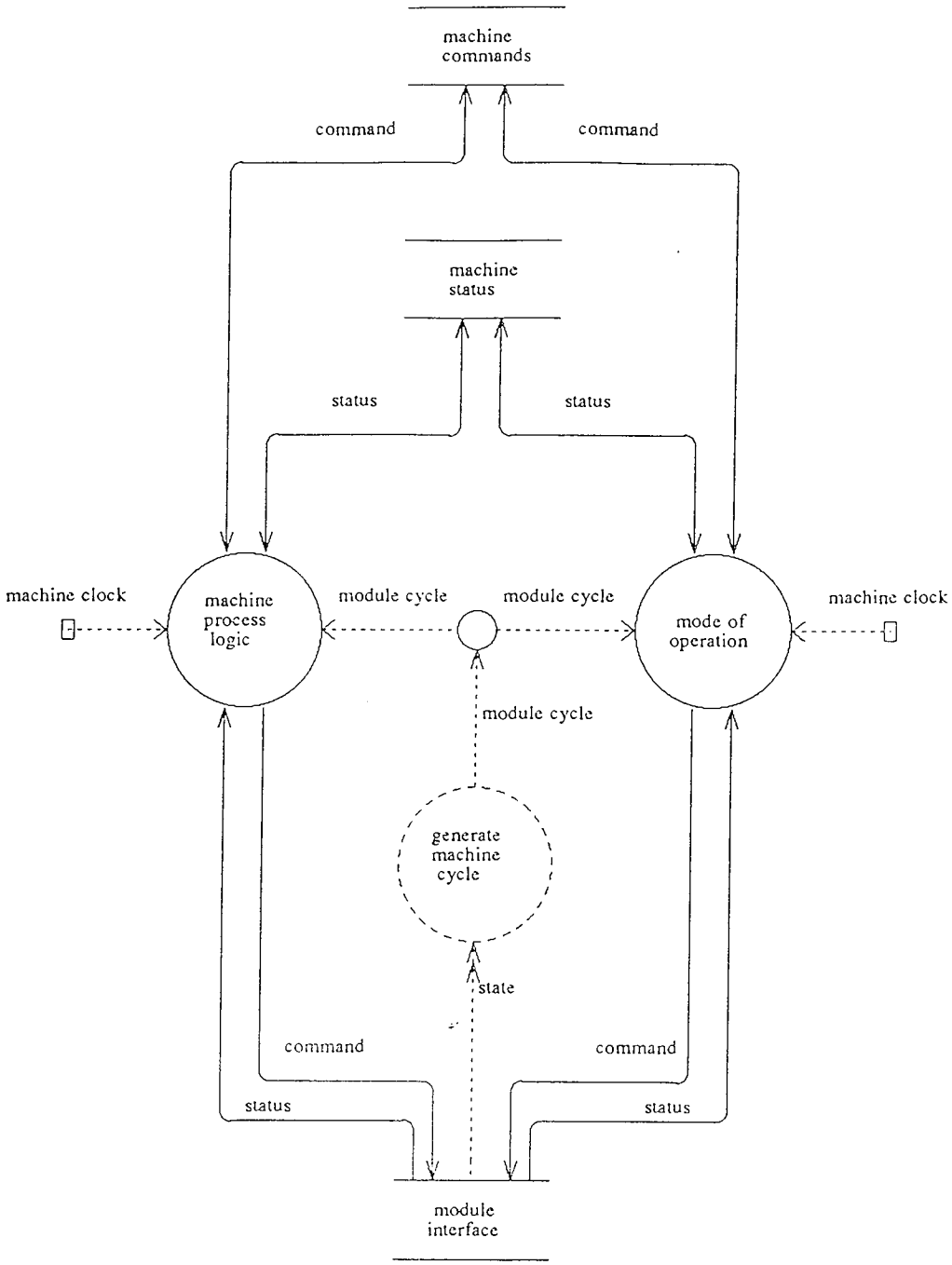


Figure 5.9 Behavioural model : Machine process logic

5.4.2.7 Module process logic

Figure 5.10 shows the lowest level of decomposition of manufacturing module controller function. The data transforms deal with control of multiple IDMs by issuing motion/synchronisation data/events and mode commands to IDMs.

The 'module process logic' data transform responds to events from both machine controller and IDM interface to supervise a set of independent drive mechanisms via the IDM interface. Events may be flow-direct such as 'user commands' or flow indirect via control transforms such as 'module cycle'. The set of IDM commands issued to the IDM interface is dependent upon the mode of operation of the mechanical module held in the module status data store. For continuous motion IDMs motion/synchronisation data may be passed, for intermittent motion IDMs synchronisation only may be passed.

The 'mode of operation' transform controls the mode of operation of the module using the module status data store to provide information on current mode and requested modes of operation via the 'module commands' data flow. Typical modes for a module will include STOPPED, RUNNING. The 'mode of operation' transform may be represented by a FSM.

The generate module cycle data transform takes the set of IDM cycles together with current IDM states to determine the start/finish of a module cycle. Module cycles typically being repetitive.

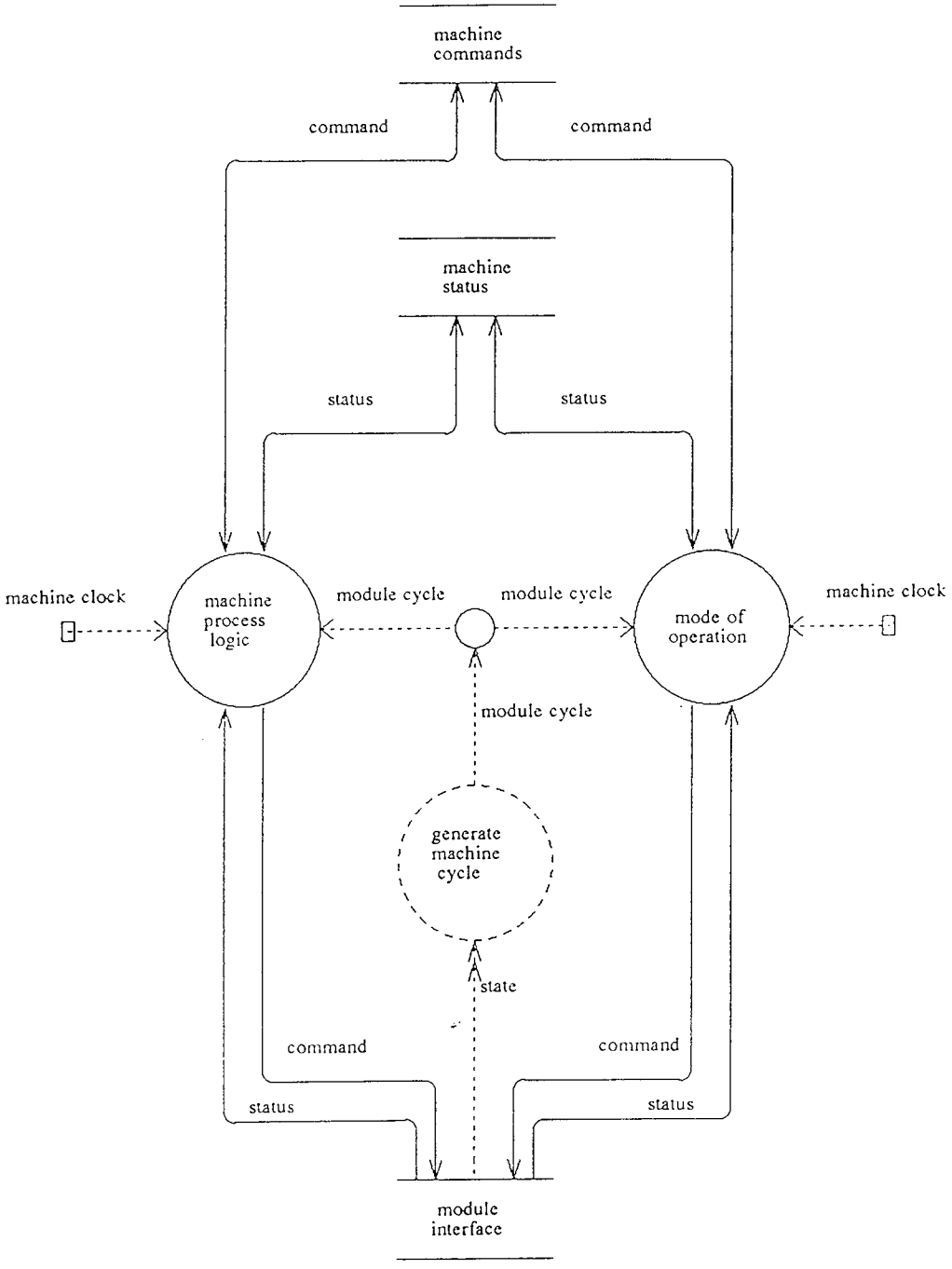


Figure 5.10 Behavioural model : Module process logic

5.4.2.8 Control independent drive mechanism

The diagram of figure 5.11 shows the lowest level of control for an IDM.

The 'mode of operation' transform controls the mode of the mechanism, using the 'IDM status' data store to provide information on current and requested modes of operation via the 'IDM commands' data flow. The 'mode of operation' transform may be represented by a FSM since the IDM concept of a mode is primitive, typically a sequence of motion segments and synchronisations. The consequences of a particular mode may require control data to be passed to the 'IDM process logic' via the 'IDM status' data store, for example to execute a new motion segment requires motion parameter data to be sent. The 'mode of operation' transform also supports inter-IDM synchronisation, the data being passed into and out of the transform into the 'IDM interface' data store. Flexibility in motion and synchronisation will be handled by this transform. The local 'IDM status' data store may be used to hold motion profile segment declarations for use by the DFD transforms.

The 'IDM process logic' transform responds to events from physical plant, IDM status and IDM interface to control the independent drive mechanism. Typically the transform provides a closed loop feedback control process, additional motion generation processes may also be necessary. The set of IDM commands issued to the actuators is usually limited to analog control, typically -10,+10v for motors, or logic switching, typically 0-24v for pneumatic cylinders. These outputs are required continuously requiring high sample rates. If accurate feedback control is required 1ms samples are typical.

The 'generate IDM cycle' data transform uses plant feedback and mode of operation to determine the start/finish of a motion segment and a complete IDM cycle.

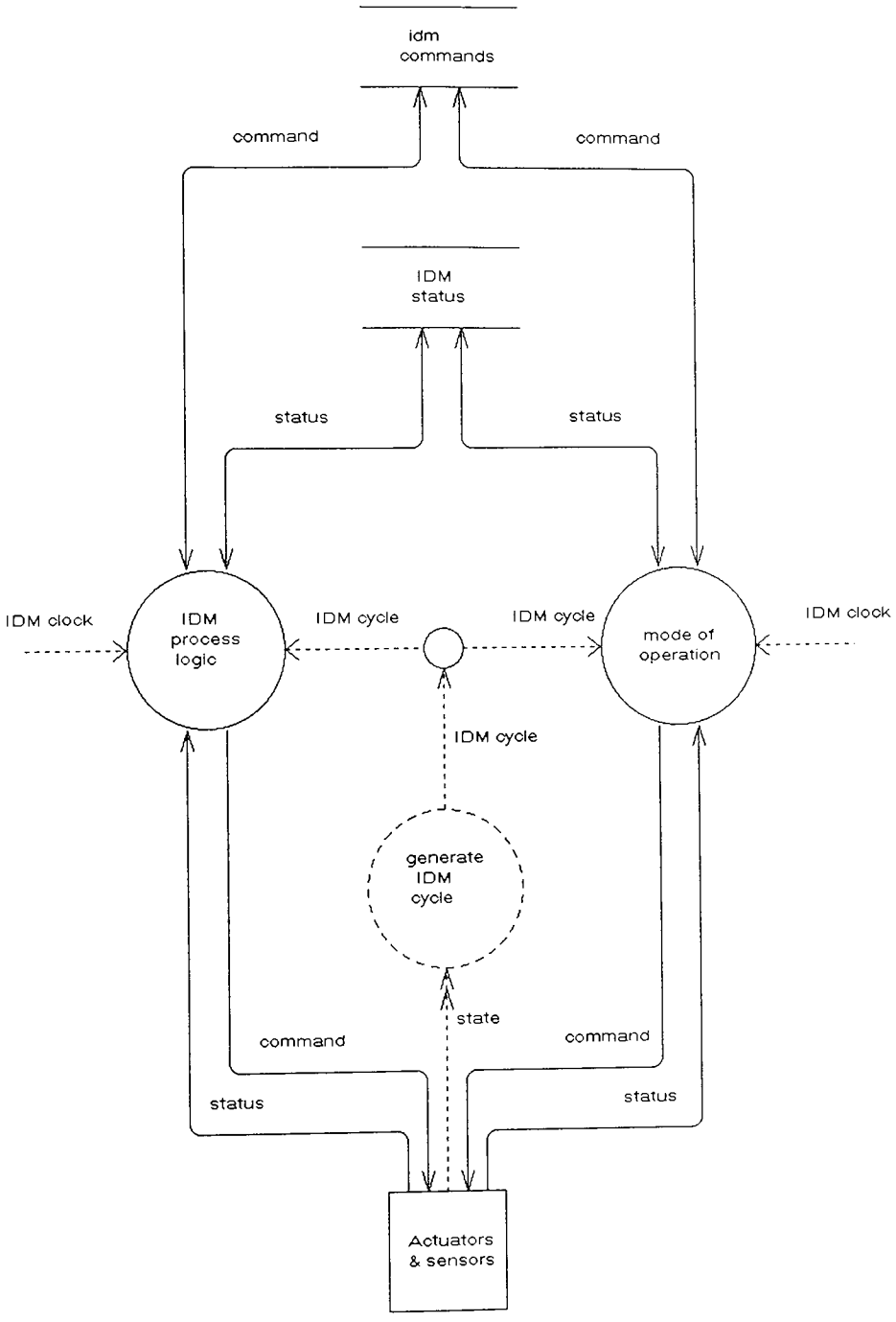


Figure 5.11 Behavioural model : IDM process logic

5.5 Controller design for the demonstrator project

A particular example of the generic IDM machine control design will now be created for the two axis demonstrator. The demonstrator is a module of an existing machine and the controller reflects this by having few high-level 'mode' operations, a single module control data transform and only two IDM controllers. However the demonstrator controller can be constructed using the procedure outlined in section 5.4.

5.5.1 Context

The first phase is to determine the external interface to the IDM machine controller. For the demonstrator this will be a simple user interface of keyboard and screen. The interface will support limited modes of operation (single shot run, continuous run, stop, emergency stop) and mechanism flexibility (speed, pack size, arbor start position synchronisation), a single machine status screen being provided to display the last command and a summary of arbor speed, arbor 'transfer start' synchronisation position, transfer speed and transfer 'pack size' variable parameters. The interface to the plant will also be simple, a two channel DAC for outputting velocity demands to the mechanism motors and a two channel ADC for returning actual velocity. The environmental context diagram for the demonstrator based on the generic design of figure 5.2 is shown in figure 5.12.

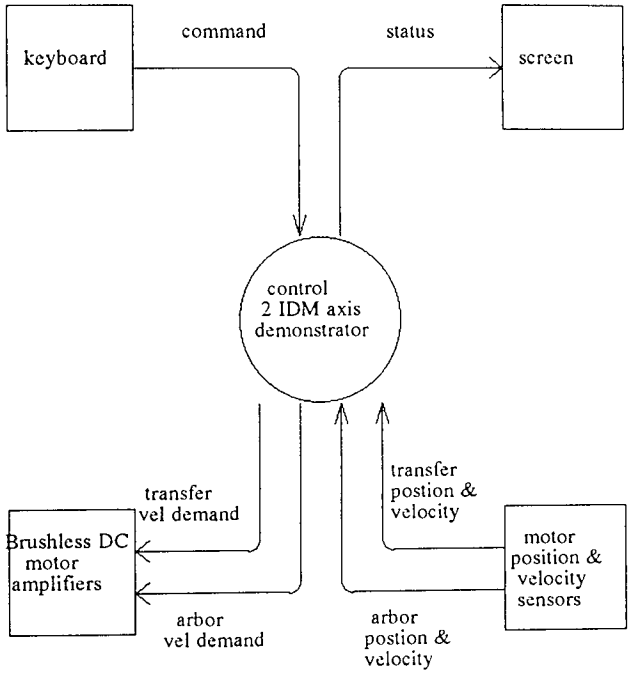


Figure 5.12 Environmental context diagram for the demonstrator

5.5.2 Demonstrator design ; behavioural model

The data transform expansion of figure 5.13 shows the highest level of design for the demonstrator IDM controller model. The connections to the external environment (keyboard, screen, motor amplifiers and motor sensors) are shown together with a hierarchical partitioning of the controller. The diagram has two levels of hierarchy, the highest represents the 'manage machine' and 'manage module' functions of the generic model of figure 5.4. This is a result of the demonstrator being a single manufacturing module taken from a production machine.

The various required modes of operation of the manufacturing module can be accomplished without a separate 'manage machine' transform of the generic model, however the human user interface (HUI) to the system must be provided. An examination of the required commands and status from the specifications shows limited requirements which can be met in the 'manage 2-axis module' transform. The individual IDMs of the demonstrator each require a motion/synchronisation controller and these are shown as 'manage arbor IDM' and 'manage transfer IDM' data transforms. These IDM controllers receive and transmit mode commands and status to the higher level module by a data store which represents the IDM interface.

In addition the IDMs must pass motion and synchronisation data to coordinate their operation. This is achieved via the 'IDM interface' data-store, providing a visible and structured connection between IDM controllers. These IDMs have been identified in the specifications as intermittent motion / intermittent synchronisation mechanisms. The motion generation function is performed locally within an IDM controller. The data passed between IDM controllers via the IDM interface is therefore used to provide synchronisation between the mechanisms. The top level behavioural model is decomposed further in the following sections.

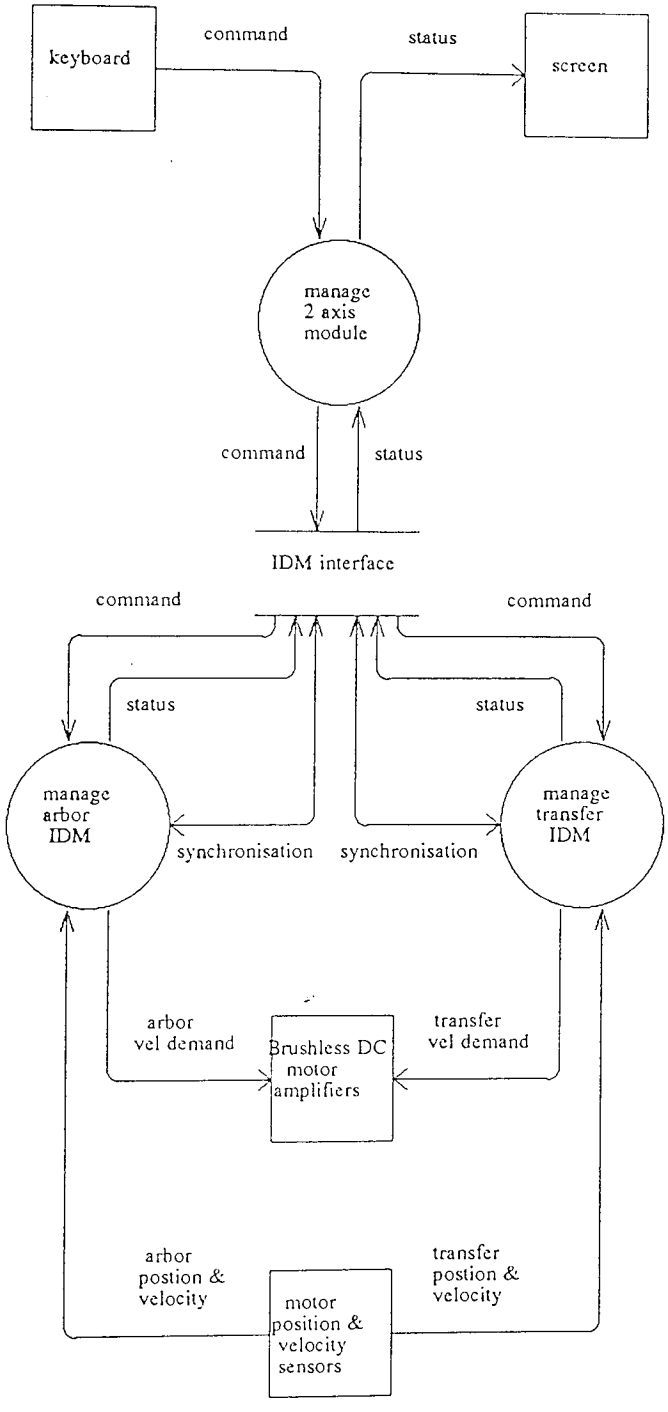


Figure 5.13 Demonstrator design - top level behavioural DFD model

5.5.2.1 Control 2 axis module

The control 2 axis module data transform combines the function of HUI and mode of operation / flexibility command dispatcher to the two 'manage IDM' transforms. An expansion of this transform can be seen in the DFD of figure 5.14. This diagram shows a decomposition of the IDM interface data-store into data-stores of particular relevance to the axis controller, these being: flexibility commands for IDM speed, position & synchronisation flexibility; mode of operation commands for single shot, continuous run, stop and emergency stop; and IDM status reporting mode of operation, speed, position & synchronisation flexibility. Table 5.1a & b defines entries in the IDM interface data store subdivided into categories of mode of operation and flexibility, each entry describes source, destination and purpose of the data.

Mode of operation IDM data store entries

Data entry	**Single shot motion**
Source	manage 2 axis module
Destination	manage arbor IDM & manage transfer IDM
Purpose	initiates a single IDM cycle from each IDM resulting in one manufacturing module cycle.
Data entry	**Continuous motion**
Source	manage 2 axis module
Destination	manage arbor IDM & manage transfer IDM
Purpose	initiates repetitive IDM cycles from each IDM resulting in repetitive manufacturing module cycles.
Data entry	**Stop motion**
Source	manage 2 axis module
Destination	manage arbor IDM & manage transfer IDM
Purpose	initiates a cessation of IDM cycles from each IDM resulting in a machine stop after current manufacturing module cycle is complete
Data entry	**Emergency stop**
Source	manage 2 axis module
Destination	manage arbor IDM & manage transfer IDM
Purpose	initiates an immediate fast deceleration to stop for both IDMs

Table 5.1a Contents of IDM interface data store for manage 2 axis module DFD mode of operation data

Flexibility IDM data store entries

Data entry	**Arbor speed**
Source	manage 2 axis module
Destination	manage arbor IDM
Purpose	variable motion profile parameter applied to all motion segments dynamically to achieve speed changes
Data entry	**Transfer start**
Source	manage 2 axis module
Destination	manage arbor IDM
Purpose	variable position synchronisation parameter - changes motion profile and alters phase at which 'transfer start' synchronisation occurs
Data entry	**Transfer speed**
Source	manage 2 axis module
Destination	manage transfer IDM
Purpose	variable motion profile parameter applied to all motion segments dynamically to achieve speed changes
Data entry	**Pack size**
Source	manage 2 axis module
Destination	manage transfer IDM
Purpose	variable motion profile parameter applied to a single motion segment to achieve a pack size change

Table 5.1b Contents of IDM interface data store for manage 2 axis module DFD flexibility data

Raw keystrokes from the operator are decoded and passed to data transforms (process flexibility, process mode of operation) which can set up the appropriate command data in the IDM interface. Returned IDM status and requested commands are taken as input to the update screen transform which generates a data flow of screen reports. To compare the specific demonstrator design of figure 5.14 with the generic design of figure 5.5, the 'decode keys' transform corresponds to the generic transform 'validate command', 'process flexibility & mode of operation command' transforms correspond to 'manage machine' and 'report state' corresponds to 'report status'.

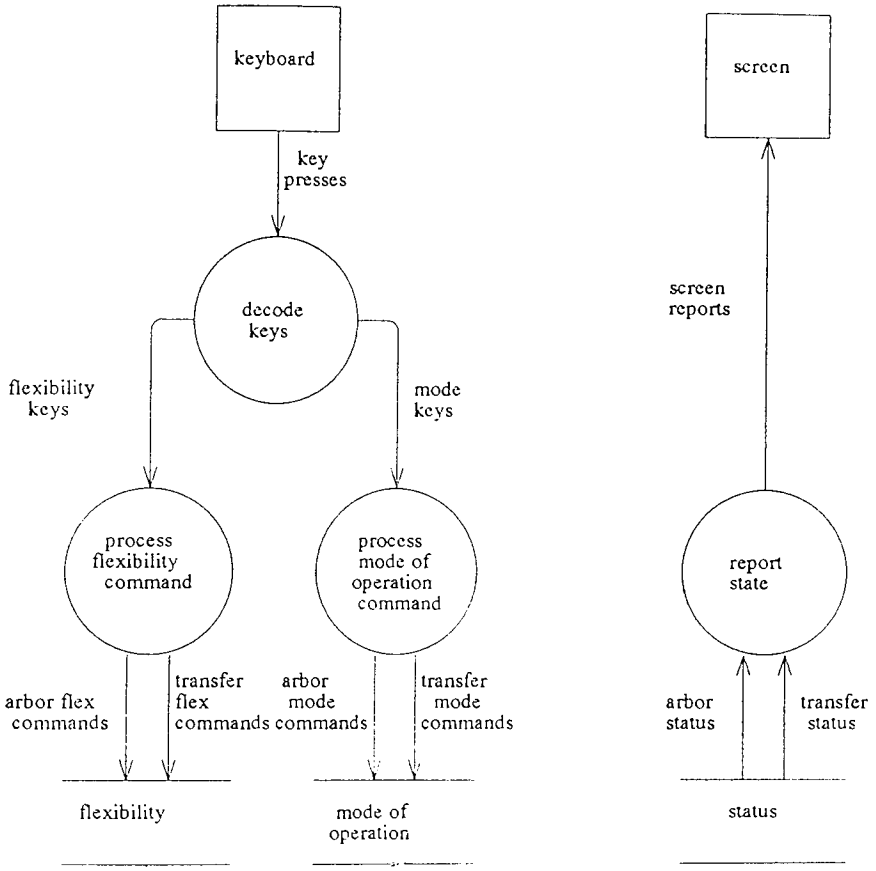


Figure 5.14 Manage 2 axis module DFD

5.5.2.2 Manage arbor DFD

An expansion of the 'manage arbor IDM' data transform can be seen in figure 5.15. This DFD shows connections to the module controller via 'mode of operation', 'flexibility' and 'arbor state' data stores (components of the 'IDM interface' data store of figure 5.4), and connection to the transfer axis via an 'inter IDM synchronisation' data store (also a component of the IDM interface store of figure 5.4). Internal data stores for controlling the IDM (the 'IDM status' data store of figure 5.11) are denoted by the 'IDM FSM state' and 'motion parameters' stores.

The 'process mode command' transform supplies mode parameters to the motion and synchronisation FSM transform. The 'process flexibility command' transform supplies required flexibility parameters to the motion trajectory generator via the 'motion parameters' data store. The 'update arbor state' transform takes FSM state and motion data to provide current axis status to the module controller (a component of the 'status' data store of figure 5.14). These three transforms provide the custodial functions of 'validate command' and 'report status' in the generic control IDM DFD of figure 5.7.

The 'arbor motion and synchronisation FSM transform' supports a finite state machine based on the required motion and synchronisation established in the requirements, particularly from executable specification prototypes. The purpose of the FSM is to determine which motion segment is to be executed, to schedule the execution of the segment and to synchronise this motion with the transfer IDM by intermittent communication via the 'inter IDM synchronisation' datastore, table 5.2 details the arbor synchronisation entries in this data store. The 'arbor motion and synchronisation FSM transform' represents the generic 'mode of operation' transform of figure 5.11.

The 'motion trajectory generator' transform uses motion segment identifiers and dynamic values of the specified motion flexibility parameters to take motion data from the data-store and produce a continuous arbor mechanism velocity demand which will make the motor move through the required motion. Feedback of motor position and velocity is used to allow closed loop control of the motor demand and to allow segment completion to be detected. The motion parameters for each motion segment come from the requirement specifications and include details of position and velocity flexibility. The 'motion trajectory generator' transform represents the generic 'IDM process logic' transform of figure 5.11.

Data entry	**Arbor at rest**
Source	manage arbor IDM
Destination	manage transfer IDM
Purpose	initiates start of transfer motion if transfer stationary enables transfer to enter arbor kinematic envelope
Data entry	**Arbor moving**
Source	manage arbor IDM
Destination	manage transfer IDM
Purpose	initiates start of transfer (when arbor decelerating to rest)

Table 5.2 Arbor entries in the 'inter IDM synchronisation' datastore

5.5.2.3 Manage transfer IDM

The 'manage transfer IDM' transform expansion uses the same functionality as the 'manage arbor IDM' transform and can be seen in the DFD of figure 5.16. However internal data and FSM action will be unique, the transfer entries in the 'inter IDM synchronisation' datastore are shown in table 5.3. Thus modularity is applied to the IDM controllers with customisation of operation being required only in motion data, motion sequence & synchronisation state machine.

Data entry	**Transfer cleared arbor**
Source	manage transfer IDM
Destination	manage arbor IDM
Purpose	initiates start arbor (when transfer decelerating to rest)

Table 5.3 Transfer entries in the 'inter IDM synchronisation' datastore

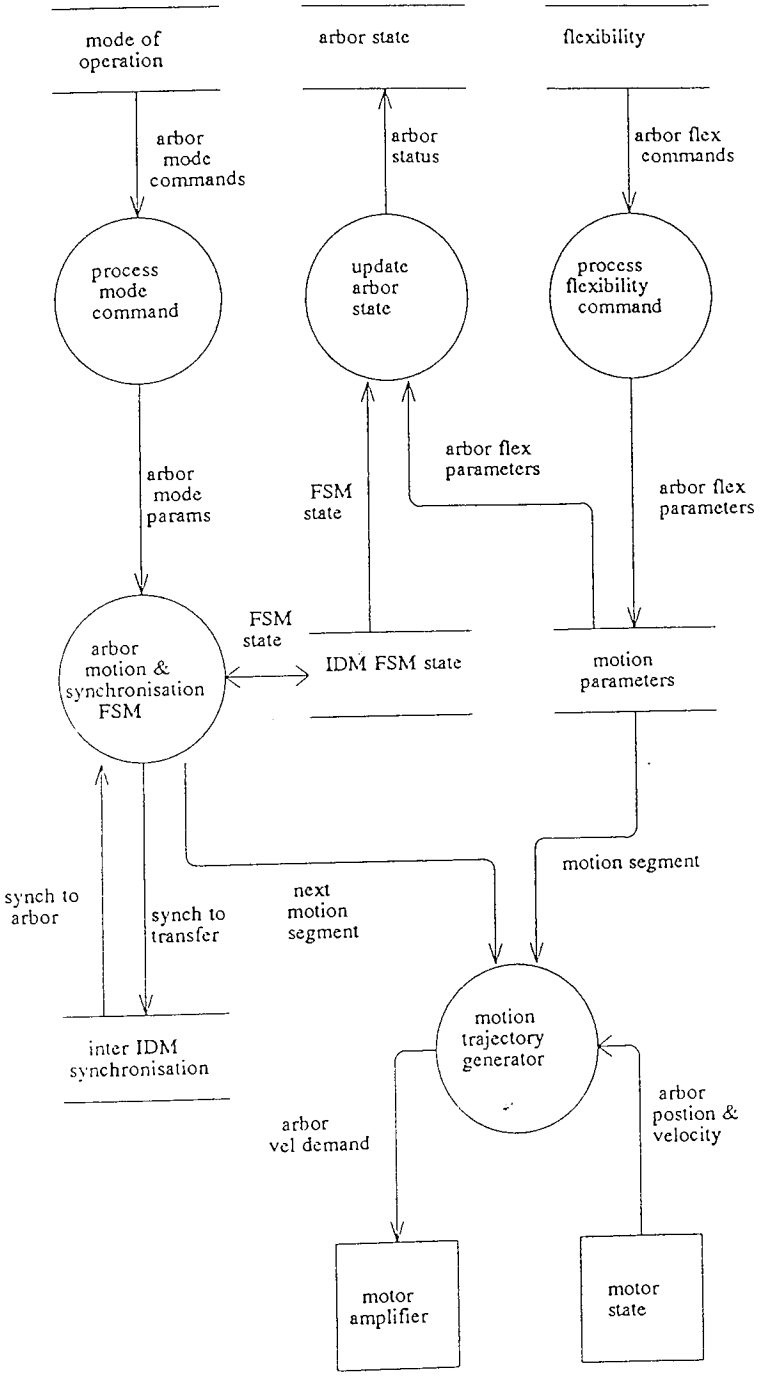


Figure 5.15 Manage arbor IDM

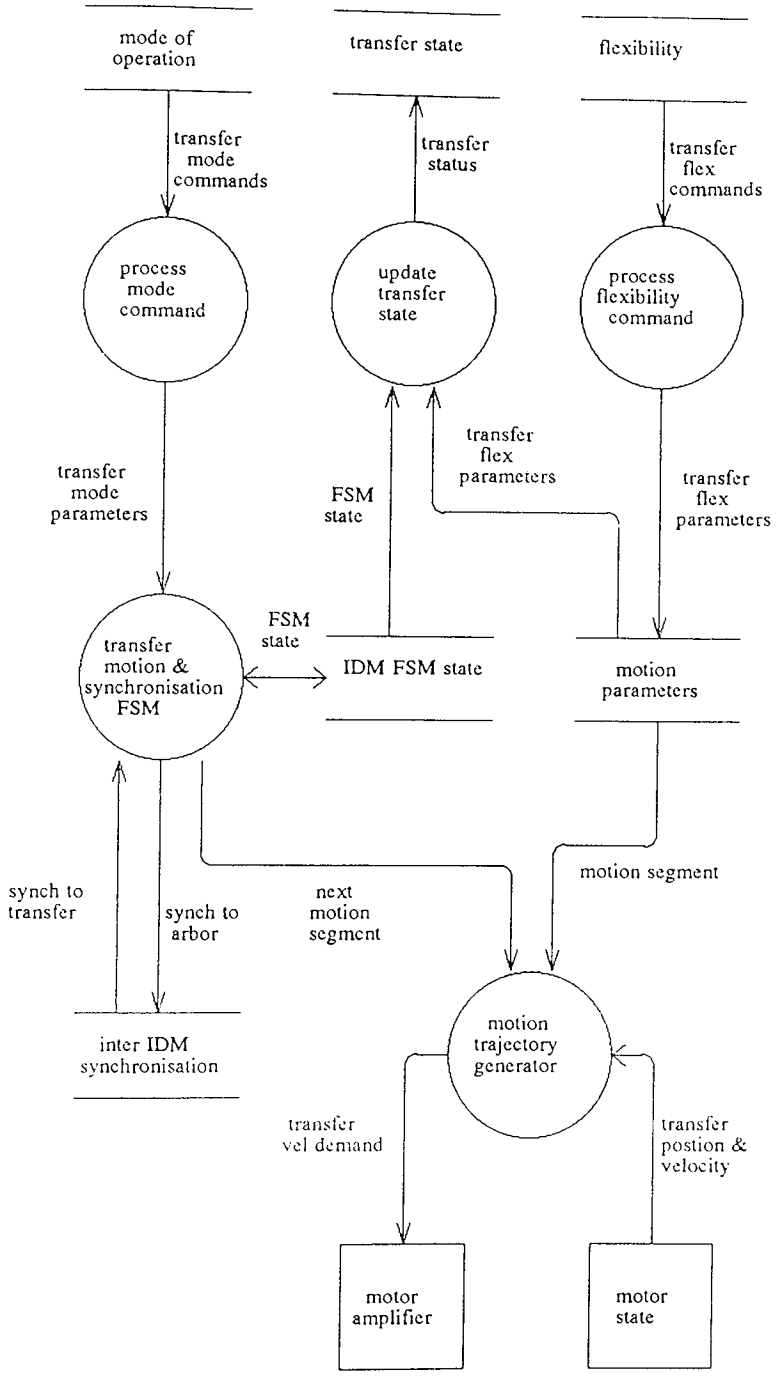


Figure 5.16 Manage transfer IDM

5.5.3 Implementation design

Section 5.5.2 described a design for the demonstrator controller. This design used the essential modelling technique of McMenamin [89] to construct a minimal system which incorporated all required controller functions assuming perfect technology. An implementation of this design must take into account the hardware and software on which the controller will run, constraints on functionality due to imperfect technology and custodial functions to ensure integrity and validity of system data.

The hardware implementation platform for the demonstrator will be an Inmos T800 Transputer card residing in an expansion slot of an IBM personal computer. The 80286 IBM PC providing HUI and disc storage to the Transputer board. The Transputer provides the main processing capability and is interfaced to the plant via ADC and DAC cards connected to physical Inmos links [73].

The software environment (Inmos' Transputer Development System or TDS) uses the concurrent programming language Occam [80] which provides the functionality to construct real-time, concurrent programs.

The implementation of the demonstrator controller design must take into account the specialities and limitations of the above processing system. The ramifications of real-time concurrent processing communicating by point-to-point synchronous communication must be understood due to the possibility of process deadlock after a communications failure between cooperating processes. This method of communication also provides an implementation without data stores between concurrent processes.

The implementation process described by McMenamin [89] takes the essential model and produces an incarnation suitable for the specified hardware and software. DFDs and Petri-nets will again be used to describe the controller implementation.

5.5.3.1 Behavioural model incarnation

The implementation DFD of figure 5.17 is based upon the control 2 axis IDM demonstrator diagram of figure 5.13. The implementation diagram takes into account technology constraints by providing an interface between control arbor / control transfer

IDMs and the motors and sensors by introducing the 'control DAC / ADC interface' transform.

The figure also shows the logical connections between transforms as direct data flows without intermediate data stores. This closely follows the Occam model of inter-process communication by annotating point-to-point unidirectional data and control flows between transforms. Thus the 'IDM interface' data store of figure 5.13 is replaced by direct data flows between IDM transforms and 'human user interface' transform for IDM status, mode and flexibility commands. 'Inter IDM synchronisation' is a component data store, found in figures 5.15 & 5.16, of the 'IDM interface' data store of figure 5.13. It is represented by discrete data flows between the two IDM control processes. Unfortunately this has the disadvantage of removing the visibility and regulation of a data-store.

The communication channels are labelled with the data they will carry, elaborating the design DFD. In particular the synchronisation channels between arbor and transfer controller are defined together with the source and sink of the data. The data transforms closely represent their intended function, for example the 'manage 2 axis module' transform of figure 5.13 is implemented as a 'human user interface' transform in figure 5.17 since the control functions are limited to disseminating commands, gathering and reporting status.

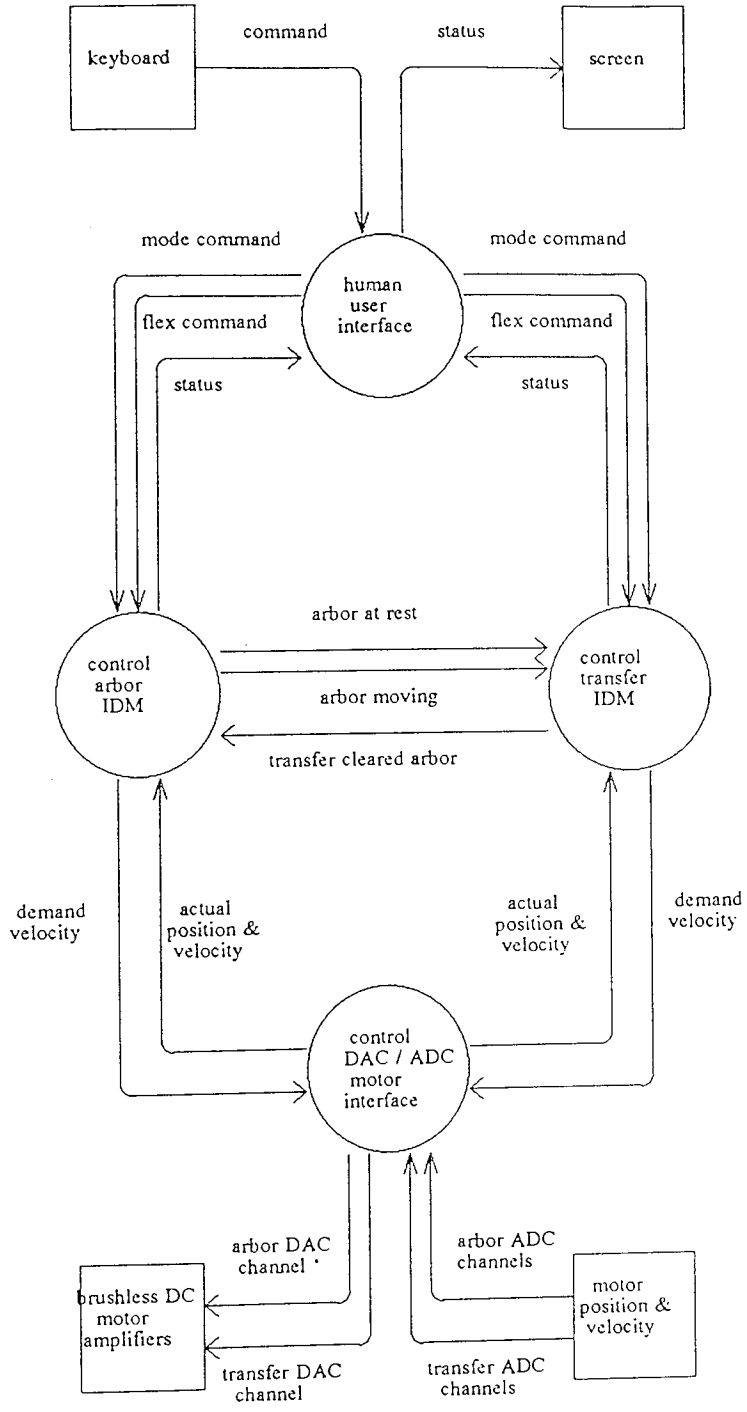


Figure 5.17 Demonstrator implementation DFD

The implementation DFD also shows the concurrent partitioning within the final controller since each transform will be a separate Occam process executing simultaneously. The next level of implementation modelling refines the functions of the data transforms of figure 5.17.

5.5.4 Human user interface

The demonstrator human user interface comprises two parts. The first accepts asynchronous commands from the user, for modes of operation and flexibility, and transmits them to the IDM controller processes. The second takes status data from the IDM controller processes and generates screen reports of the machine status. Channel communication is used extensively to connect the HUI to the User and the IDM controllers. The human user interface DFD of figure 5.18 supports the functions of the manage 2 axis module DFD of figure 5.14, the flexibility and mode of operation commands supported are detailed in table 5.1. The 'report state' transform provides the user with a list of available commands, the last command executed and the current state of the flexibility parameters of the system (transfer speed, arbor speed, pack size and transfer start distance).

The HUI task has the responsibility for sequencing the start-up and shut-down of the system. This is achieved by sending start/shutdown commands on HUI output channels.

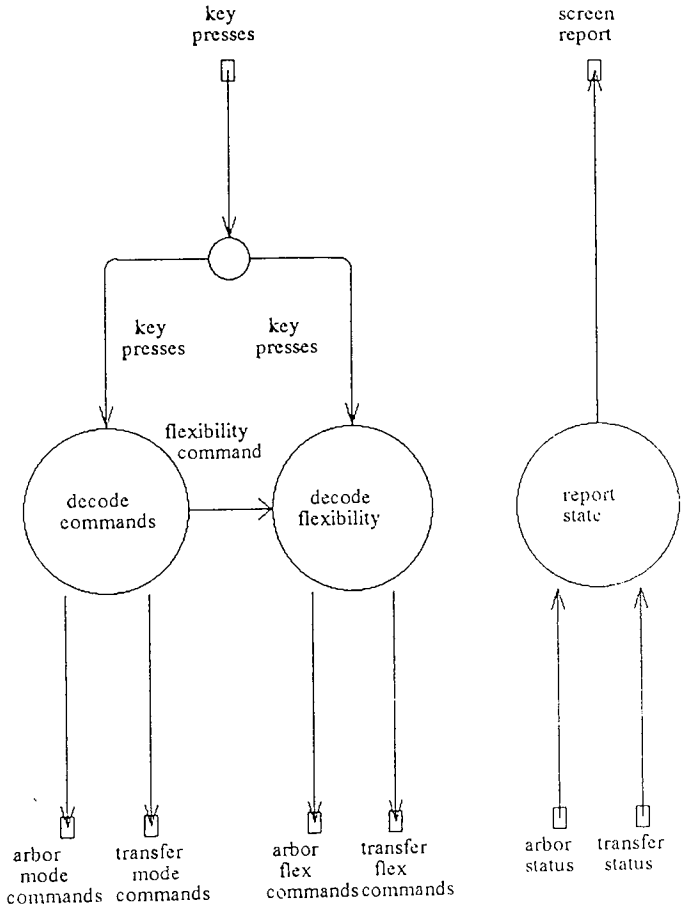


Figure 5.18 Demonstrator implementation HUI DFD

5.5.5 Control arbor

This DFD, shown in figure 5.19, represents the function of axis control and will be realised by a series of cooperating processes. The DFD has real-time control event flows to support sampled data operation for generating a velocity demand (IDM clock). The diagram containing the 'essential' components of the design DFD of figure 5.15 partitioned into three sections according to function.

The 'decode mode of operation' and 'decode flexibility command' transforms send and receive command and status to the HUI, performing the functions of 'process mode command', 'process flexibility command' and 'update arbor state' of figure 5.15. The data stores 'motion parameters' and 'IDM FSM state' correspond with datastores in figure 5.15, additional 'flexibility parameters' and 'mode of operation' data stores are included to refine the detail of the 'IDM state' datastore of figure 5.10.

The 'motion and synchronisation FSM' transform, based on the Petri-net of figure 4.3 schedules the execution of motion segments (defined in table 3.2 of the requirements specification) held in the 'motion parameters' datastore, in order to construct complex motion profiles. This transform also performs inter IDM synchronisation, as defined in the Petri-net of figure 4.3 to schedule and interlock the operation of the arbor IDM with the transfer IDM. The arbor 'motion and synchronisation FSM' transform sending discrete 'arbor at rest' and 'arbor moving' synchronisation communications and receiving 'transfer cleared arbor' communications in real-time.

The 'motion segment dispatcher', 'motion segment calculation', 'generate velocity demand' and 'motion segment complete' transforms are the implementation of the 'motion trajectory generator' transform of figure 5.15.

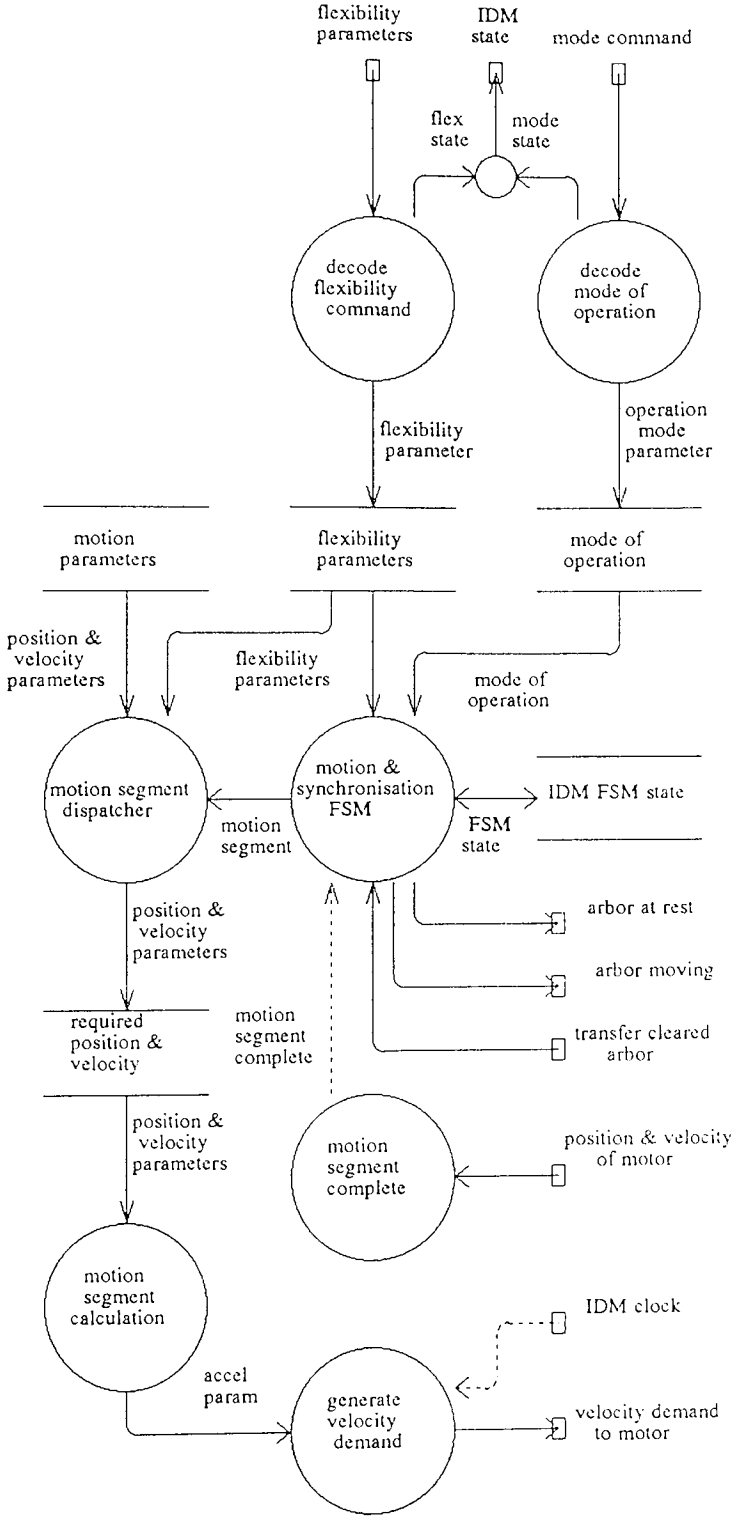


Figure 5.19 Demonstrator implementation : Control arbor

The 'motion segment dispatcher' processes motion segment identifiers provided by the 'motion & synchronisation FSM' transform in order to access motion segment data held in the 'motion parameters' data store, current values of 'arbor speed' and 'transfer start' flexibility are read from the 'flexibility parameters' data store and used to provide terminating velocity and position demands for the current motion segment. Thus dynamic flexibility is incorporated into the IDM motion profiles. The 'motion segment calculation' transform generates an acceleration parameter which is used by the 'generate velocity demand' to produce a real-time constant acceleration ramping velocity demand to the motor.

5.5.6 Control transfer

The control transfer DFD, shown in figure 5.20, represents the function of IDM control for the transfer slider and is similar to figure 5.19 since it uses the same modular structure. The data transforms perform the same functions in figures 5.19 and 5.20. The 'motion and synchronisation FSM' transform of figure 5.19 is based on the requirement specification Petri-net of figure 4.3. The motion segment specifications for the transfer slider are defined in table 3.3 of Chapter 3.

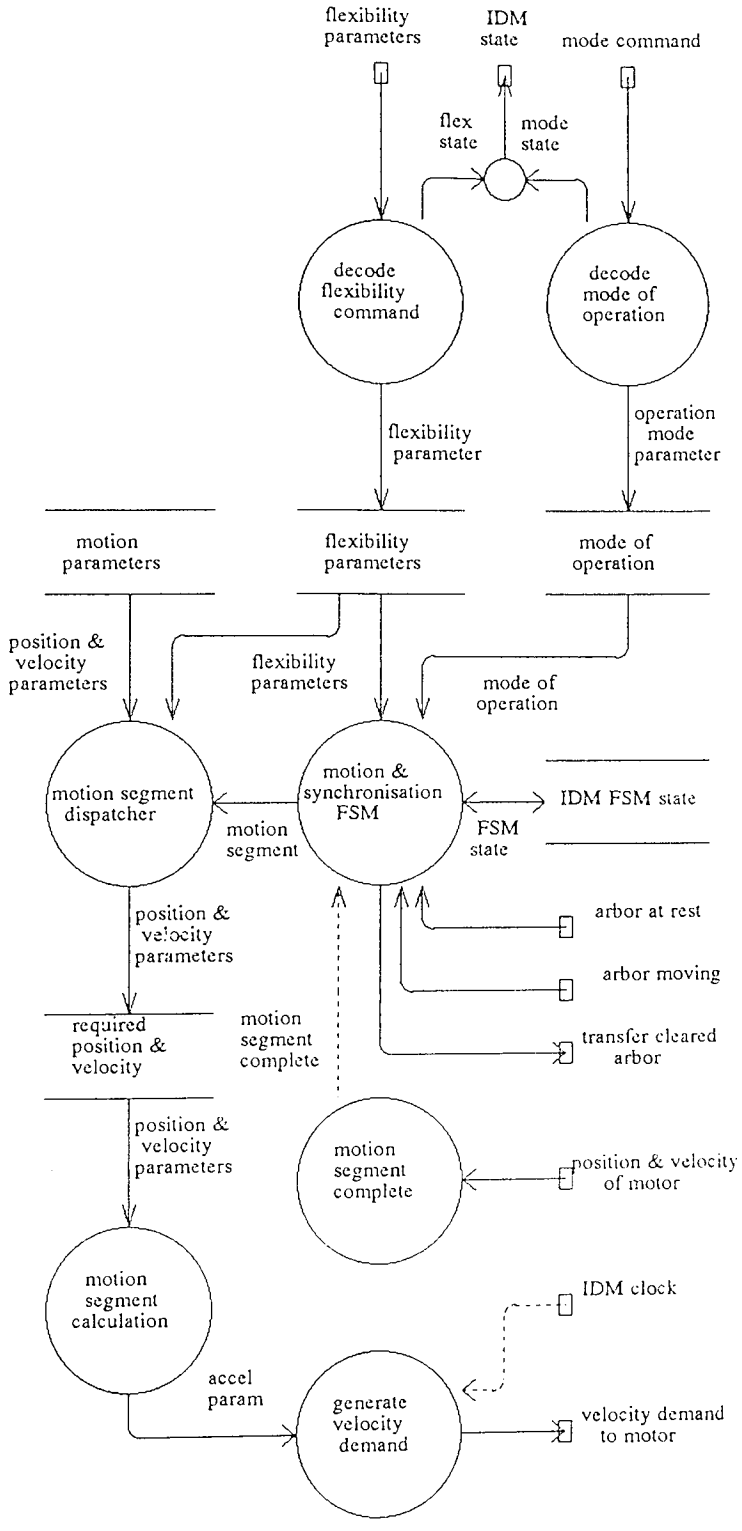


Figure 5.20 Demonstrator implementation : Control transfer

5.5.7 Control DAC / ADC motor interface

The control DAC / ADC motor interface DFD is a description of the interfacing necessary to connect the control IDM process transforms to the physical plant. The module is hardware dependant and for the Occam/Transputer implementation uses a four channel DAC card and an eight channel ADC card to send motor velocity demands and receive velocity feedback.

The diagram of figure 5.21 shows the controller output data flows being concentrated to the appropriate DAC card and the plant sensors being converted to appropriate controller input data. The DAC output conversions being prompted by data present on the 'demand velocity' data flows, whilst the ADC inputs run asynchronously at the maximum sample speed of the hardware transmitting the last input value when a read is made from the data flow.

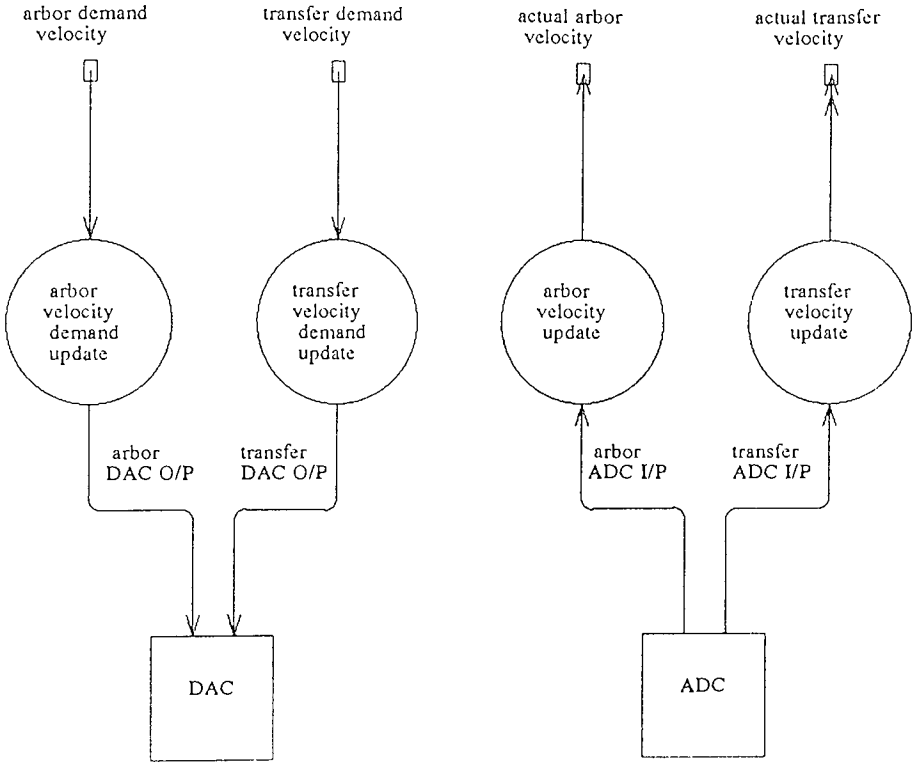


Figure 5.21 Implementation model : Control DAC / ADC motor interface

5.6 Prototypes developed in support of the implementation

In order to develop a correct and feasible system, areas of controller implementation which have uncertainty or ambiguity associated with them, such as dialogue and format of the user interface or exceptional processing after a fault, can be examined using a prototype. These prototypes are executable representations of the controller function under consideration. They demonstrate perceived functionality allowing judgements on suitability from interested parties such as machine user's. They also demonstrate feasibility of the controller to provide required real-time performance.

5.6.1 Operator command / status - HUI

These functions provide both the command and status processing of the machine.

To obtain the correct user command / machine status dialogue a prototype of the HUI can be developed which incorporates the functional commands and also details non-functional requirements such as screen formats and data reports.

The commands are often based on the flexibility inherent in the machine and the user requirements for product and process detailed in section 3.4 provides a basis for the development of the prototype HUI.

An example of a prototype HUI screen-layout for the demonstrator can be found in figure 5.22 which demonstrates the essence of user/machine dialogue by showing available commands and status reports. An iteration through versions of the prototype builds up the required command/status structure by reference to specifications and from judgements on suitability and format from users of the prototype.

Demonstration machine start up display

Two axis complex motion demand generation incorporating position and velocity flexibility
(C) C.M.Draper 1990, Aston University, Birmingham
Flexibility :
transfer speed 9 -> 100 cpm, 9cpm increments
arbor speed 20 -> 110 cpm, 9cpm incremnets
pack size 40 -> 60mm, 2.2mm increments
transfer start distance (synch)

Press a key to begin

Demonstration machine main menu (after command '2' entered)

MODE CONTROL
1 - single shot motion
2 - start cyclic motion
3 - stop cyclic motion
4 - shutdown program
5 - emergency stop

FLEXIBILITY CONTROL
6 - change flexibility parameter
7 - display flexibility parameter

Report from <<< ARBOR >>> command >>> START <<< executed
Report from <<< TRANSFER >>> command >>> START <<< executed

Demonstration machine display after command '7' entered

>>>> flexibility parameters <<<<
FP 1, arbor speed = 9
FP 2, transfer speed = 9
FP 3, pack size = 4
FP 4, tranfser start distance = 0

Figure 5.22 Screen displays from Occam prototype HUI

5.6.2 Prototype of machine controller

To test the real-time performance of the IDM controller design a prototype of machine operation based upon an Occam representation of the Petri-net interaction model of figure 4.3 was constructed. This model used a single FSM to schedule motion of two proprietary motion control cards. These cards had motion profiles stored which corresponded to the motion segments defined in tables 3.2 and 3.3 of the requirements specification. Particular values of flexibility parameters were used to create an envelope of IDM performance providing motion profiles demonstrating speed and position

flexibility. The motion profile generators were scheduled by the Occam FSM controller via a digital IO card. The aim of the prototype was to prove the capability of the motors to drive the complex motion profiles and to examine the suitability of Occam and the Transputer for real-time control in normal and abnormal operating conditions.

The source code of the Occam HUI/FSM & IO interface and the PROPOS-E motion control programs can be found in the appendices. Results of the prototype can be found in Chapter 6.

5.7 Conclusions

This Chapter has described an approach to the design of a computer controller for machines using independent drive mechanisms. A generic design, described using a DFD notation, based on a hierarchy of control within a modular software structure has been described. This model defines machine, manufacturing module and IDM levels within the controller and uses regulated interfaces for inter & intra level communication.

A design using the generic template has been described for the demonstration machine. This design being a subset of a full machine controller since the demonstrator is a single mechanical module from an existing machine.

An implementation of the demonstrator design tailored for the Transputer microprocessor and Occam programming language has been outlined. The coding and performance of which will be detailed in the next Chapter.

Finally prototypes used in support of the design process have been introduced. Prototype screen layouts with user commands and machine reports have been detailed. A prototype machine controller using Occam for inter IDM synchronisation and together with proprietary motion control has been introduced. This prototype seeks to prove the capability of the technology to perform the required control functions. Results of the prototype are detailed in Chapter 6.

----------//----------

Chapter 6

Controller implementation and results

6.0 Introduction

This Chapter describes the fully flexible implementation of the IDM demonstration machine computer controller. It details experimental results of the prototype and flexible demonstrator controllers. The results are analysed and verified against specifications to gauge the performance of the machine and it's computer controller. A discussion of the Occam controller implementation and results is provided which examines the suitability of the design techniques, hardware and software used in creating the controllers.

6.1 Occam implementation of the demonstrator IDM controller

This section gives an overview of the Occam implementation code structure. It then details the methods used in transforming the intermittent motion, discrete synchronisation module controller design defined by the DFDs and Petri-net FSMs in section 5.5 into an executable Occam code implementation.

In the full flexibility IDM controller a HUI process handles asynchronous operator commands for modes of operation and motion segment flexibility, passing commands to the two IDM controller processes via Occam channels. It also generates screen reports from status data returned from the IDM controller processes, the formats of which are described in figure 5.22.

The motion profiles for each IDM are derived from motion specification tables 3.2 and 3.3. The simple motion segments are stored in a motion database and are scheduled by a segment sequence FSM. The motion control function of the software is modular and reusable. Synchronisation for the IDM is incorporated into the segment scheduling FSM resulting in an IDM motion and synchronisation sequence FSM. This FSM is based on the validated Petri-net specifications of figure 4.3, to insure correct and safe operation in

the final implementation. A separate FSM is generated for each IDM and inserted as an Occam process into the generic motion-generation code structure.

The motion trajectory generator of each IDM controller outputs a velocity demand to it's associated servo-drive. To convert these software demands to output voltages a DAC process drives an Inmos link based DAC card. Similarly an ADC process takes voltage representations of velocity as input from an Inmos link based ADC card and returns these to the IDM controllers.

The system therefore comprises four main concurrent Occam processes operating in real-time, communicating with the plant via ADC/DAC interfaces and the user via keyboard and screen. Complete source code for the controller can be found in the appendices. The following sections details of how the Occam code is generated from the controller design.

6.1.1 Hierarchical partitioning of the implementation

Occam allows code to be modularised into concurrent threads of execution (parallel processes) and sequential threads of execution (sequential processes). These parallel and sequential threads can be formed into a hierarchy of execution. The implementation of the IDM machine controller uses a hierarchy based on the DFD layering of design introduced in Chapter 5. Each data process becomes a thread of execution, executing concurrently, communicating by message passing via point-to-point channels. The communication channels perform synchronisation and data transfer functions.

The DFD of figure 5.17 shows the first level of implementation partitioning for the demonstrator. The Occam code to support the implementation defines four procedures to represent the four DFD data processes. In figure 6.1 an Occam code fragment defines these procedures as concurrent processes using the PAR primitive and declares the communications channels between them. The parameters of the procedures are the channels which perform the functions of the named data flows of figure 5.17.

The code in figure 6.1 represents the main concurrent processes and the communications network between them. For the next level of decomposition each of the four main procedures is again represented by concurrent and sequential processes.

```
-- communication channel declarations
[2]CHAN OF ANY   to.HUI :
[2]CHAN OF INT   from.HUI.flex, from.HUI.command :
[2]CHAN OF BOOL  to.event :
CHAN OF BOOL     from.arbor.at.rest, from.arbor.moving,
                 to.transfer.cleared.arbor, stop.dac :
CHAN OF INT      arbor.to.dac, transfer.to.dac:

-- concurrent process construct
PAR
      HUI.process ( to.event,from.HUI.flex,
                    from.HUI.command,  to.HUI, stop.dac)
      arbor.drum( to.event[arbor], from.HUI.flex[arbor],
                  from.HUI.command[arbor], to.HUI[arbor],
                  from.arbor.at.rest,  from.arbor.moving,
                  to.transfer.cleared.arbor, arbor.to.dac)
      third.transfer( to.event[transfer],from.HUI.flex[transfer],
                    from.HUI.command[transfer], to.HUI[transfer],
                    to.transfer.cleared.arbor, from.arbor.at.rest,
                    from.arbor.moving, transfer.to.dac)
      dac           ( arbor.to.dac, transfer.to.dac,
                    to.dac, stop.dac )
```

Figure 6.1 Process partitioning & communications in the IDM demonstration controller implementation

Figure 6.2a shows an Occam code fragment which represents the 'control arbor IDM' data process. It implements the 'arbor.drum' procedure of figure 6.1 and is based upon the DFD design of figure 5.19. The code fragments of figures 6.2b and 6.2c are further decompositions of controller function. Figure 6.2b shows the 'motion.demand' procedure which performs the DFD data processes of 'decode flexibility command', 'decode mode of operation', 'motion & synchronisation FSM' and 'motion segment dispatcher' of figure 5.19 Figure 6.2c shows the 'velocity.demand.gen' Occam procedure which performs the DFD data processes of 'motion segment calculation', 'generate velocity demand' and

'motion segment complete' of figure 5.19. The Occam code of figures 6.1 and 6.2a,b,c demonstrates the hierarchical partitioning of DFDs into threads of execution using Occam SEQ & PAR constructs and introduces channels as a mechanism for passing data and synchronising concurrent Occam processes.

```
CHAN OF BOOL HUI, demand.satisfied :
CHAN OF INT to.accel.calc, velocity.demand.clock :
PAR
  Event.timer ( velocity.demand.clock, to.event, rate )
  Motion.demand ( from.HUI.flex,  from.HUI.command,
                  to.HUI, to.accel.calc, demand.satisfied )
  Velocity.demand.gen ( velocity.demand.clock, to.servo,
                        to.accel.calc, demand.satisfied )
```

Figure 6.2a Occam 'arbor.drum' procedure

```
PROC Motion.demand ( CHAN OF INT from.HUI.flex, from.HUI.command, to.HUI,
                     to.accel,  from.velocity )
  WHILE not.finished
  SEQ
   -- monitor for new commands / schedule motion
   ALT     -- get an IDM mode or flexibility  command
    from.HUI.command ? command
      -- check and implement command
    running & SKIP       -- no commands, carry on with motion control
     SEQ
      IF
        emergency
          SEQ
            -- manage communications for dedlock free operation
        TRUE
          SEQ
            -- motion and synchronisation sequence FSM
```

Figure 6.2b Occam 'motion.demand' procedure

```
PROC Velocity.demand.gen (CHAN velocity.demand.clock, to.servo,
                          to.accel.calc,demand.satisfied )
  SEQ
    WHILE velocity.demand.enabled
     SEQ
      -- monitor for new rate demand / shutdown or clock start
      ALT
       velocity.demand.clock ? vel.sample.rate
          SEQ  -- Occam real-time IDM control
              -- calculate new velocity demand
              IF    - inform 'motion.demand' when simple segment complete
                 (new.demand>terminal.velocity)
                    demand.satisfied ! TRUE
              -- output new vel demand
              to.servo ! new.demand   -- new demand is current velocity

       -- get new motion segment parameters from 'motion.demand'
       to.accel.calc ? position.increment,  terminal.velocity
       SEQ
        -- calculate acceleration rate
        -- read current IDM velocity
```

Figure 6.2c Occam 'velocity.demand.gen' procedure

6.1.2 Communication between concurrent processes

Inter process point-to-point channel communication is used to pass data between concurrent processes and to synchronise inter process operations. The synchronisation function of channels is demonstrated in the 'velocity.demand.gen' procedure of figure 6.2c. The real-time control loop code is activated by communication from the 'event.timer' procedure which sends data on the 'velocity.demand.clock' channel at the IDM sample rate. Asynchronous channel communication is demonstrated in the 'motion.demand' procedure of figure 6.2b. The ALT construct allows races between channel communications and logical constructs. This allows the asynchronous reception of channel communication by forcing the process to check if data is available on a

channel before continuing execution via an alternative path. In figure 6.2b the 'from.HUI.command' channel opens an alternative execution path if data is present on the channel, if no data is present then the motion control code is executed.

Within each Occam process care must be taken to avoid deadlock, a channel communication never being received, and livelock, a channel communication never being sent. Deadlock & livelock have the effect of halting a thread of execution in an Occam program. A significant amount of code is required in the full flexibility implementation to accommodate failure of inter IDM communications in a fault-tolerant manner.

6.1.3 Occam state machines for IDM interaction and motion profiling

The validated Petri-net model of IDM motion and interaction of figure 4.3 is used to create the motion and synchronisation sequence FSM for each IDM. A state variable is used to hold the dynamic state of the FSM and is analogous to the token in a Petri-net FSM diagram.

The states themselves are represented by sequential 'SEQ' processes selected by a conditional 'IF' statement using the state variable in the test condition. Within each state the possible inter-IDM synchronisations are supported by inter-IDM communication using channels, channel gets '?' await a scheduling communication and channel puts '!' initiate a scheduling communication.

A channel put, to initiate motion in a cooperating IDM, may be issued when an IDM controller FSM enters a state which represents the IDM at a defined interaction position. Similarly a channel get may be used to schedule the next motion segment of an IDM controller by forcing a wait until the communication is satisfied. Figure 6.3a & 6.3b show simplified Occam code for the state machines derived from the Petri-net of figure 4.3.

The IDM FSM controller changes state when the defined motion termination condition for the current state is achieved and any input communications are complete. The motion conditions which affect a change of state rely on a distance being covered (defined in the motion specifications). Thus the FSM controller logic does not use time for any synchronisation between IDMs. This ensures that the synchronisation is valid for the defined range of speed flexibility.

```
SEQ
  -- arbor  motion dispatcher with inbuilt synchronisation
  IF
   hard.emergency.stop
    -- manage emergency stop

   state = arbor.at.rest
      PAR
       SEQ
        from.arbor.at.rest ! TRUE
       SEQ
        from.transfer.clear.of.arbor ? prompt
        state := arbor.moving.1

   state = arbor.moving.1
    SEQ
     motion.seg ( 0 )
     state := arbor.moving.2

   state = arbor.moving.2
    SEQ
     motion.seg ( 1 )
     state := arbor.moving.3

   state = arbor.moving.3
    SEQ
     motion.seg ( 2 )
     state := arbor.moving.to.rest

   state = arbor.moving.to.rest
    PAR
      SEQ
       from.arbor.moving ! TRUE      -- enable start for transfer
      SEQ
        motion.seg ( 3 )
        state := arbor.at.rest
        -- deal with cycle commands
```

Figure 6.3a Arbor 'motion & synchronisation sequence FSM'
simplified Occam code fragment

```
SEQ
  -- transfer motion dispatcher with inbuilt synchronisation
  IF
   hard.emergency.stop
    -- deal with emergency

   state = transfer.at.rest
    SEQ
     from.arbor.moving ? prompt
     state := transfer.moving.to.enter.arbor

   state = transfer.moving.to.enter.arbor
    SEQ
     motion.seg ( 0 )
     from.arbor.at.rest ? prompt
     state := transfer.moving.in.arbor.1

   state = transfer.moving.in.arbor.1
    SEQ
     motion.seg ( 1 )
     state := transfer.moving.in.arbor.2

   state = transfer.moving.in.arbor.2
    SEQ
     motion.seg ( 2 )
     state := transfer.moving.in.arbor.3

   state = transfer.moving.in.arbor.3
    SEQ
     motion.seg ( 3 )
     state := transfer.moving.in.arbor.4

   state = transfer.moving.in.arbor.4
    SEQ
     motion.seg ( 4 )
     state := transfer.moving.outside.arbor.to.rest

   state = transfer.moving.outside.arbor.to.rest
     PAR
      from.transfer.moving.clear.to.rest ! TRUE
      SEQ
       motion.seg ( 5 )
       state := transfer.at.rest
       -- perform any cycle commands
```

Figure 6.3b Transfer 'motion and synchronisation sequence FSM' simplified Occam code fragment

6.1.4 Flexibility in motion and mode of operation

The channel communications for changing flexibility and receiving IDM mode commands are performed at the beginning of each state machine scan. Mode commands which modify the loop operations of the IDM FSM are processed at the end of the motion sequence cycle and allow single shot and repetitive operation. Flexibility commands change the dynamic values of defined flexibility parameters held in the IDM motion segment datastore.

When the FSM controller schedules the next motion segment these flexibility parameters are used to calculate dynamic position and velocity data which is then passed to the 'velocity.demand.gen' procedure. The flexibility parameters are used to calculate a particular value of terminating position and velocity between the defined limits in the motion segment data structure. Figure 6.4 contains a code fragment for the 'motion.seg' dynamic motion parameter calculation and communication procedure.

```
PROC motion.seg ( INT seg )
  SEQ           - calculation is   min+(flex*(max-min)/9)
    position := Motion.segment[seg][ms.pos.min] +
                (((Motion.segment[seg][ms.pos.max] -
                  Motion.segment[seg][ms.pos.min])/9) *
                  flex.param[Motion.segment[seg][ms.flex.pos]])

    velocity := Motion.segment[seg][ms.vel.min] +
                (((Motion.segment[seg][ms.vel.max] -
                  Motion.segment[seg][ms.vel.min])/9) *
                  flex.param[Motion.segment[seg][ms.flex.vel]])

    to.accel ! position       -- send dynamic position and velocity data
    to.accel ! velocity

    from.velocity ? vel.ok   -- await motion segment completion
```

Figure 6.4 'motion.seg' Occam code fragment

6.1.5 Motion profile generation and closed loop control

Figure 6.2c shows the real-time control procedure which performs closed loop control for the IDM. The procedure accepts position increment and terminal velocity parameters as asynchronous channel communications data from the 'motion.demand' procedure. Given the actual and required position and velocity a constant acceleration parameter is calculated, this corresponds to the 'motion segment' calculation of figure 5.19.

The 'event.timer' procedure generates a scheduling communication to the 'velocity.demand.gen' procedure at the defined IDM sample rate. In response to this communication a new velocity demand is created from the current demand and the calculated acceleration parameter. This velocity demand is then output to the DAC procedure via the 'to.servo' communications channel.

The DAC procedure generates a control voltage for the brushless DC servo-amplifier. Feedback of velocity is generated and if the defined termination velocity has been achieved the motion and synchronisation sequence FSM is informed by communication on the 'demand.satisfied' channel. Servo-control of the IDM drive is achieved by a proprietary PID controller built into the brushless DC drive system.

6.2 Results

The operational results of the prototype controller described in section 5.6.2 and the fully flexible controller described in section 6.1 are detailed in this section.

The dynamic operation of the controllers are presented as traces of the real-time motion profiles generated by the two IDM drives of the demonstration machine. These profiles demonstrate the complex motion trajectories created from simple motion segments and dynamic modification of position and velocity in segments of the motions. In addition the synchronisation between motion profiles is demonstrated to show the capability for maintaining normal synchronisation and synchronisation after an abnormal plant condition.

Two sets of demonstration machine results are given. The first results are for the prototype controller based on Occam synchronisation and proprietary motion control.

The second results are for the final implementation of a self-contained Occam/Transputer controller which embodies dynamic motion and synchronisation flexibility.

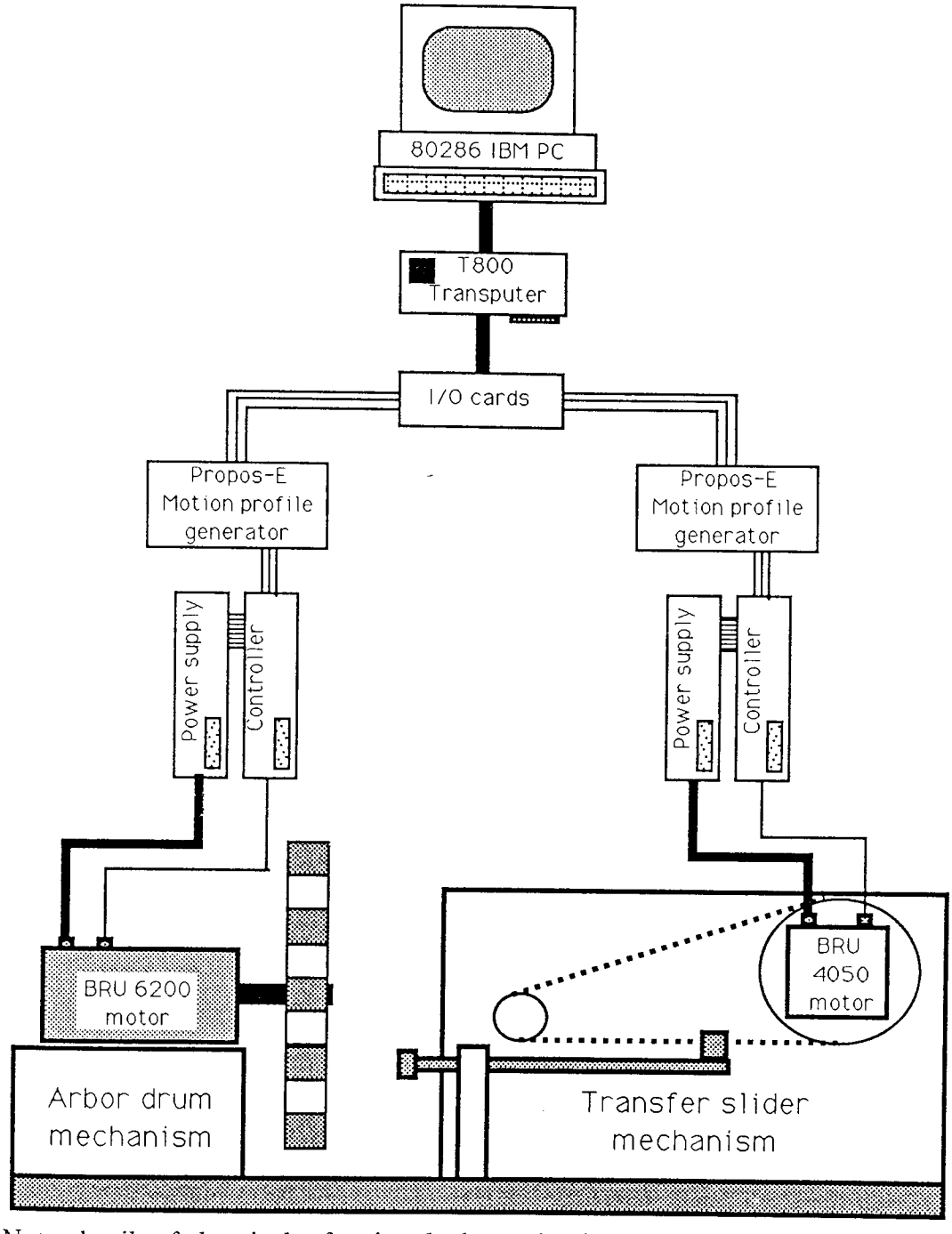
The design of both prototype and final implementation controllers use the Petri-net models of IDM synchronisation and motion derived in Chapter 4. These models use position as a scheduling parameter for changes of FSM state, as a result the models are speed independent and do not require definition of explicit timing for the cooperating IDMs actions. The motion of each IDM is scheduled by synchronisation logic decoupled from speed control logic. However the results detailed in this Chapter show timings of IDM operations, these relate to a particular setting of a speed flexibility parameter for each IDM motion controller.

6.2.1 Prototype demonstrator controller

The prototype two IDM demonstration machine computer controller comprises an IBM PC hosting a T414 Transputer card interfaced by a Transputer link based digital IO card to two proprietary motion controllers. Each motion controller provides a velocity demand and receives position data from a brushless DC motor servo controller.

The IBM PC provides keyboard, screen and disc storage facilities for the Transputer development system. The Transputer controller provides sequence and safety synchronisation commands for the two Electrocraft PROPOS-E motion controllers. The PROPOS-E boards execute assembler-like motion programs which respond to the Transputer discrete synchronisation commands. Flexibility in motion is achieved by reprogramming the motion controllers with new profiles. A schematic of the hardware for the prototype can be seen in figure 6.5

The dynamic performance of the prototype is shown in figures 6.6 to 6.8. These figures demonstrate motion of the arbor and transfer mechanisms, including position and velocity flexibility, and motion sequence of the transfer mechanism under normal and abnormal IDM synchronisation conditions.



Note. details of electrical safety interlocks on the rig have been omitted for clarity

Figure 6.5 Prototype hardware for 2 IDM demonstration machine

Each figure shows two real-time traces taken from a storage oscilloscope. The upper trace is the demand velocity while the lower is measured velocity from a tachometer. The figures also details the settings of flexibility parameters required to produce the motion profiles.

Figure 6.6 shows the arbor drum motion profile at maximum speed and with maximum pack-size (position) flexibility. This profile is the physical incarnation of the motion specification diagram of figure 3.9. The triangular velocity profile of figure 6.6 matches the specification profile in figure 3.9.

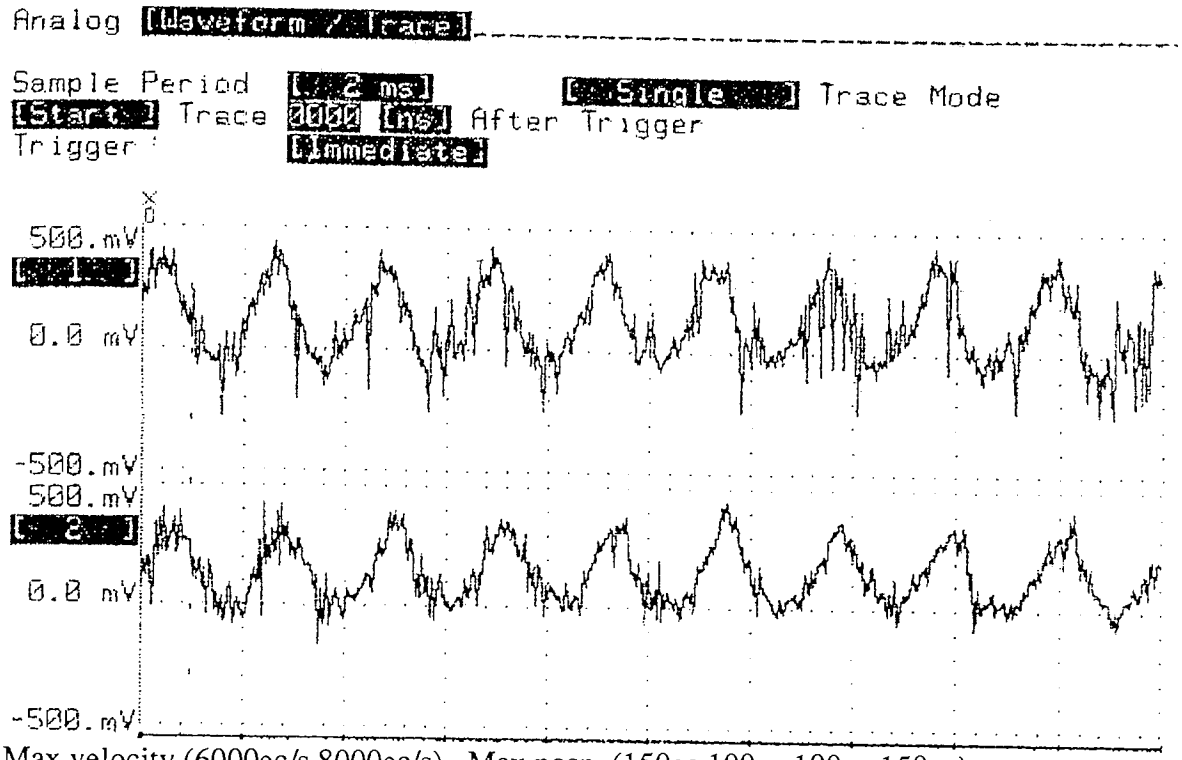
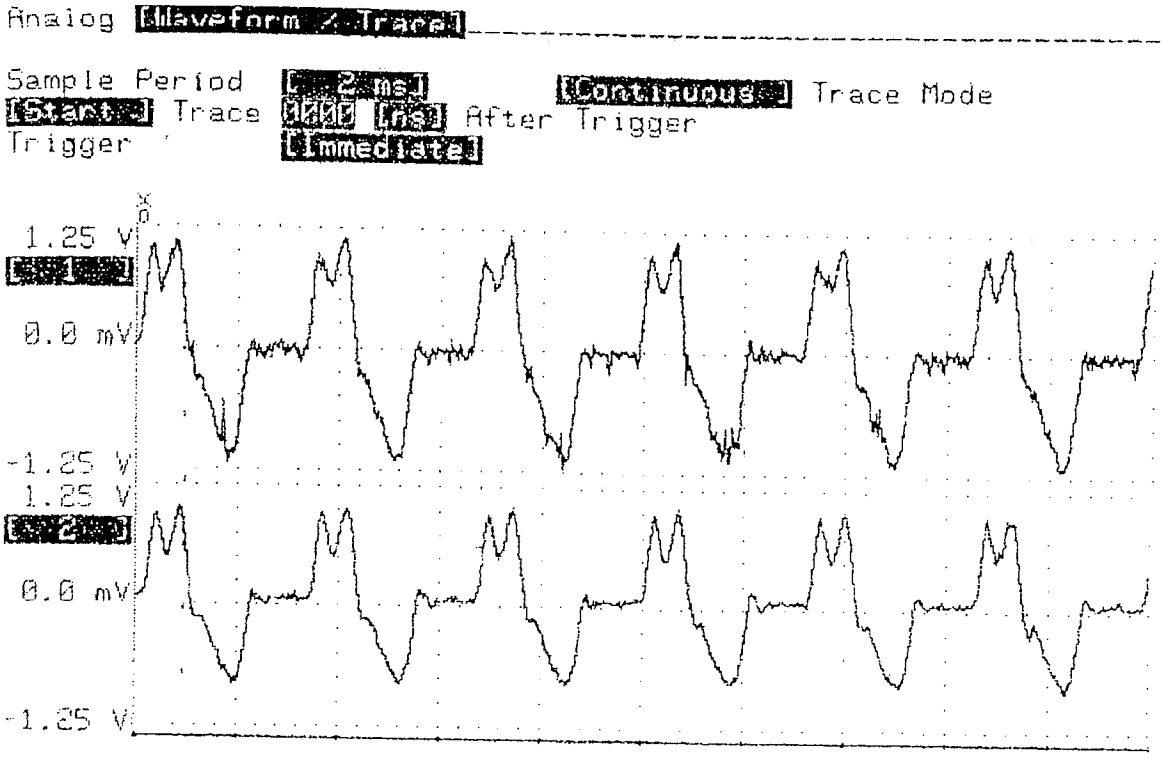


Figure 6.6 Arbor motion at maximum speed

The trace of figure 6.7 shows the maximum speed velocity motion profile for the transfer mechanism. This profile is the physical incarnation of the motion specification diagram of figure 3.11.

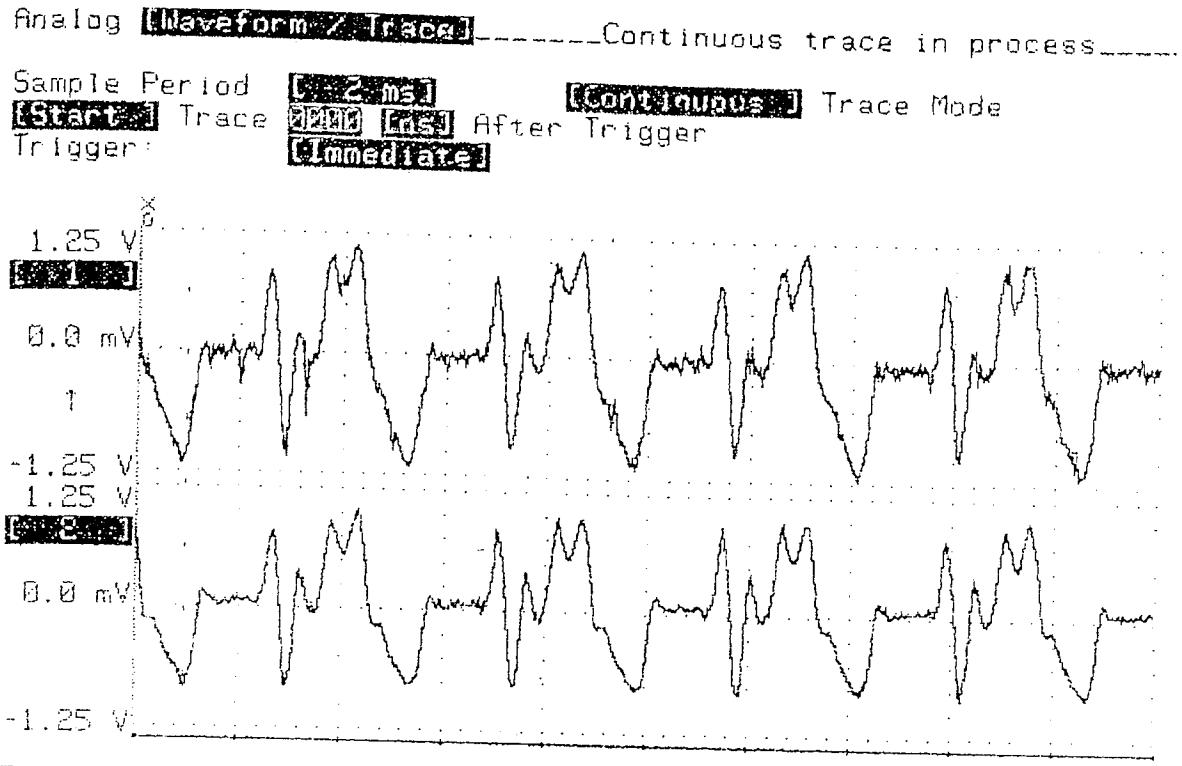


Max velocity (2400 ec/s) 190 ppm, Max position (350ec,700ec) normal profile

Figure 6.7 Transfer Max position & velocity motion

These diagrams are a close match with the velocity motion profiles detailed in the transfer motion specifications of figure 3.11. This demonstrates the capability of the prototype to meet the machine requirements and verifies the operation of the prototype against specifications. The prototype also demonstrates how a complex motion profile may be generated from a sequence of simple motion segments.

The prototype also demonstrates the ability of the motion sequence to respond to plant conditions directly. In figure 6.8 the transfer motion profile for an abort-enter arbor operation is shown. The retract of the transfer to the rest position can be clearly seen.



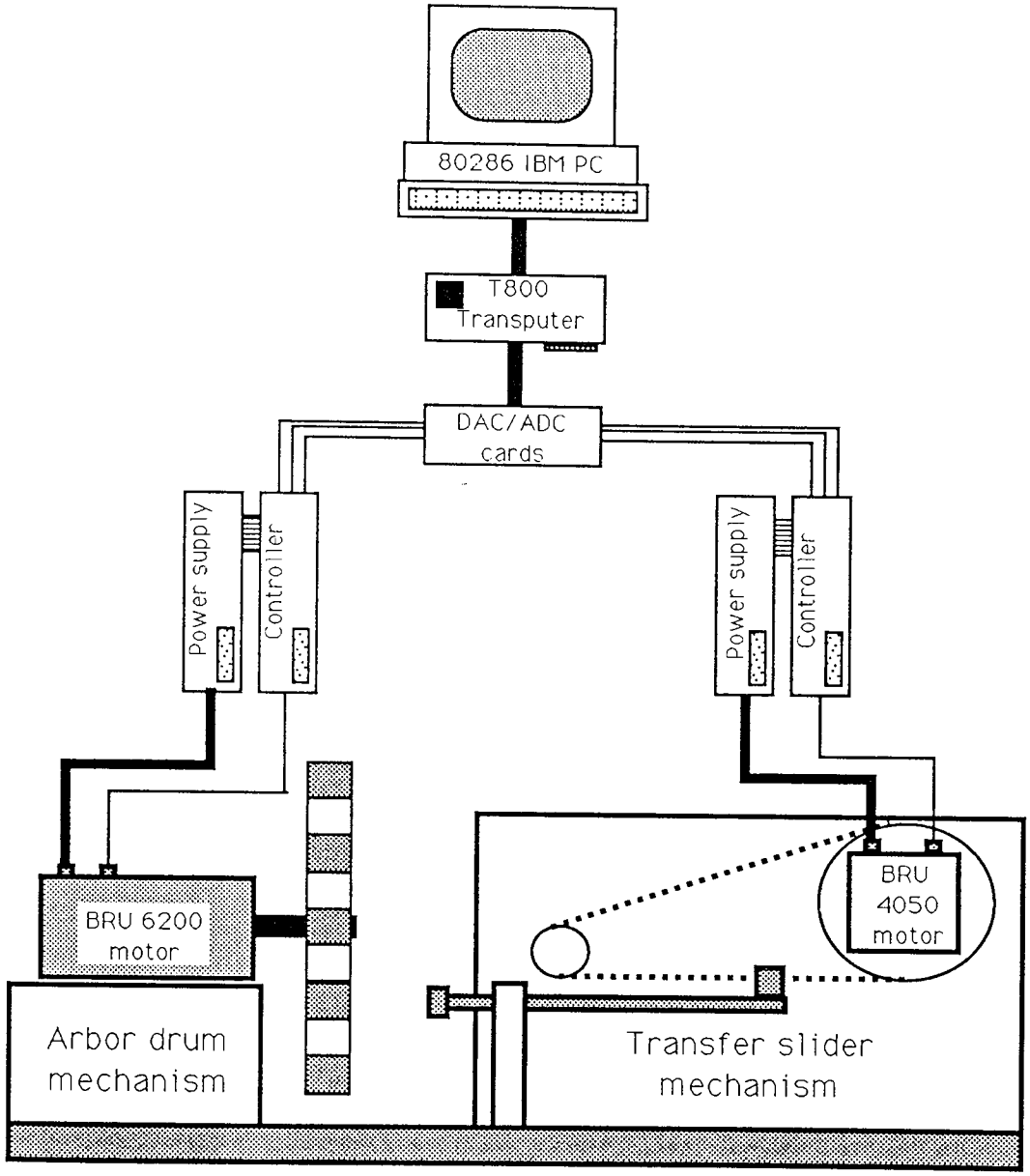
Fast retract transfer

Figure 6.8 Transfer abort enter arbor

These profiles demonstrate the capability of the actuators to drive the independent drive mechanisms through the specified motion profiles with the necessary flexibility in speed, position and synchronisation. The prototype proved the concept of flexible IDMs synchronised by software, however to fully exploit the IDMs a full implementation of motion and synchronisation flexibility in the controller is required.

6.2.2 Occam / Transputer fully flexible controller for the 2 IDM demonstration machine

The results of the demonstration machine Occam / Transputer final implementation two IDM controller which permits maximum use of the flexibility inherent in the IDMs is presented in this section. The controller uses a modular software design based on Occam processes executing concurrently and communicating by channels. The hardware for the final implementation controller can be seen in figure 6.9. A detailed description of the controller implementation can be found in section 6.1.



Note. details of electrical safety interlocks on the rig have been omitted for clarity

Figure 6.9 Implementation hardware for the 2 IDM demonstration machine

In the controller the software IDM control processes, one each for arbor and transfer, are formed from standard control modules which provide motion generation and synchronisation between intermittent motion, discrete-synchronisation IDMs. Flexibility in position, velocity and synchronisation of the basic motion profile is catered for by the controller.

The aim of the controller is to demonstrate the suitability of Occam and the Transputer for modular control of real-time processes; to demonstrate the flexibility in position, velocity and synchronisation of independently driven mechanisms; and to show how a correctly specified multiple IDM control system can accommodate abnormal plant states in a recoverable manner or if this is impossible in a fail-safe manner.

The results of the final implementation controller can be seen in the following figures. The transfer velocity motion profile is the top trace in the figures whilst the arbor drum is the bottom trace. Flexibility in motion of each IDM is achieved by changing motion parameters for speed and distance for an individual IDM. Flexibility in synchronisation is achieved by varying the position parameter which controls an inter IDM synchronisation event (represented by an inter FSM transition in the Petri-net models).

Figures 6.10 and 6.11 show the complete motion profiles of arbor and transfer respectively.

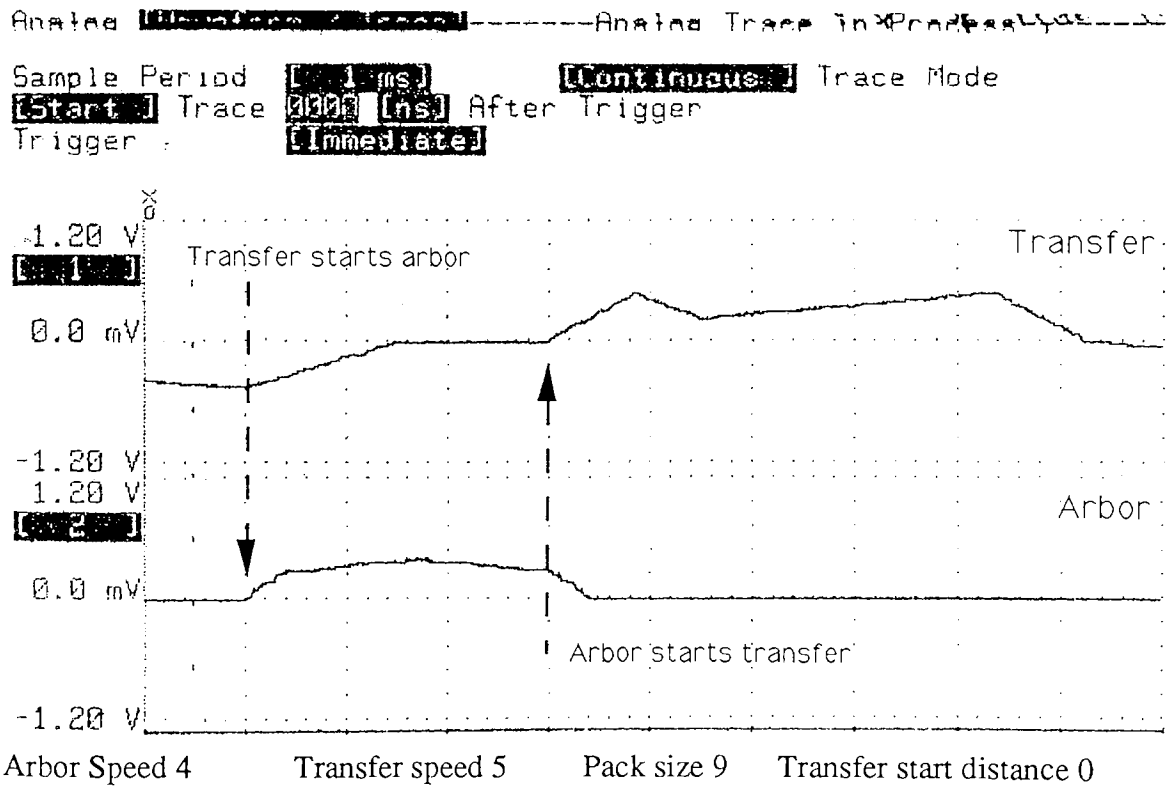


Figure 6.10 Transfer and arbor motions - transfer starts arbor when clear

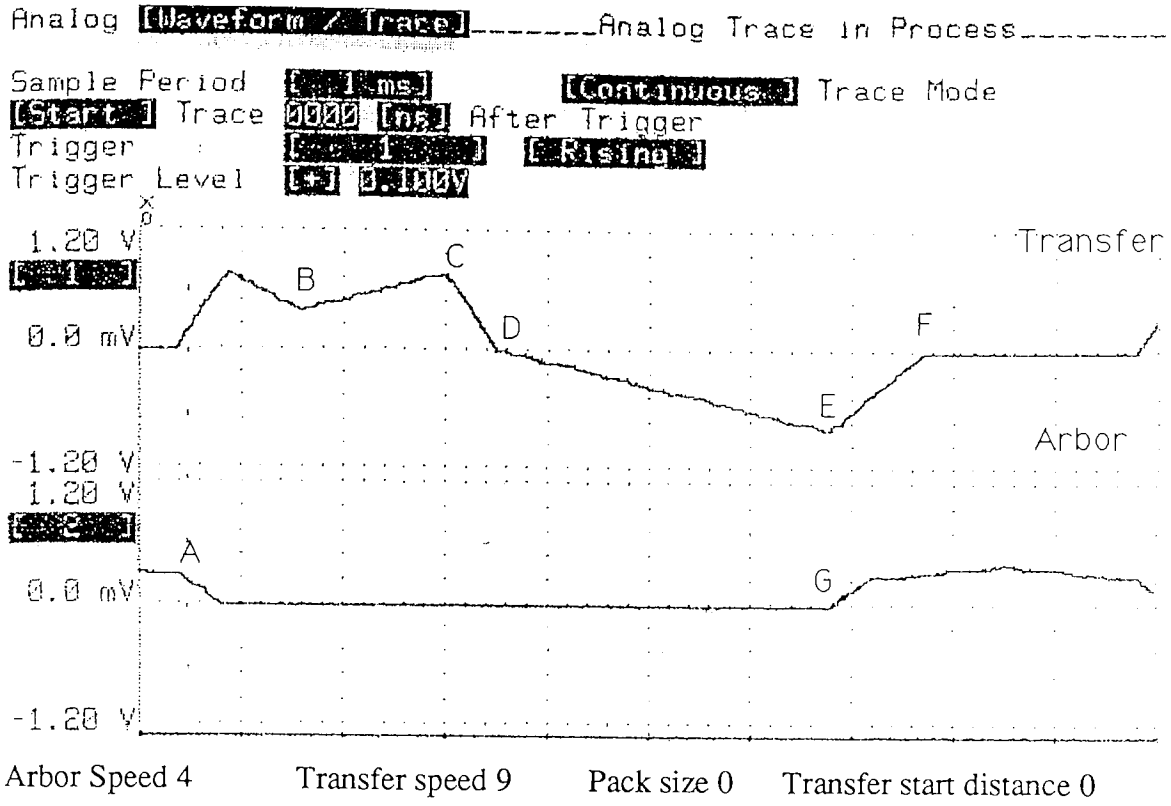


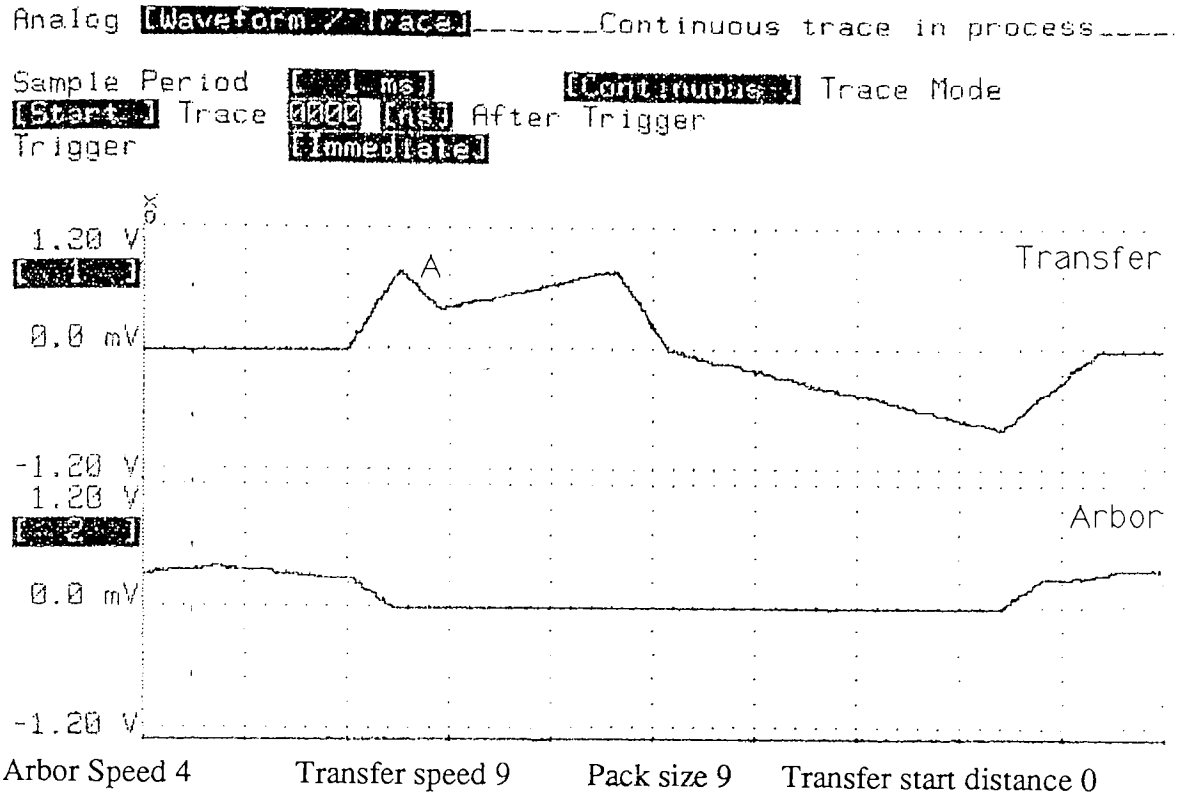
Figure 6.11 Transfer and arbor motions - transfer starts arbor when clear, transfer shows minimum pack size flex

The complex motion profiles are constructed from a sequence of motion segments. The arbor has four segments. The transfer has six segments for normal operation, three for 'abort enter arbor' and one for emergency stop.

Figure 6.11 also shows the synchronisation between axes for the demonstration machine. The synchronisation occurs as follows: The transfer starts when the arbor drum is decelerating to rest (position A), the transfer accelerates towards the arbor and then decelerates to contact the pack (position B). The transfer then accelerates to maximum speed (position C) before decelerating to rest fully inserted in the arbor (position D). The transfer then accelerates out of the arbor until it is fully clear when it initiates the start of the arbor drum motion (position E) before it decelerates to rest (position F). The arbor drum performs a simple incremental move on receiving the transfer-clear synchronisation (position G), again signalling the transfer to start when it commences deceleration to rest (position A).

Figure 6.11 shows the transfer executing a minimum pack size profile while the upper trace of figure 6.12 shows a maximum pack size transfer profile. The first transfer

deceleration segment of figure 6.12 (position A) shows this by having a steeper gradient than the corresponding segment in figure 6.12, indicating a reduction in distance travelled to allow for the larger pack.



Arbor Speed 4 Transfer speed 9 Pack size 9 Transfer start distance 0

Figure 6.12 Transfer profile maximum pack size flex

The traces in figures 6.13 show the minimum speed of operation for the demonstration machine of 7.5 pack cycles per minute. The maximum speed of 84 pack cycles per minute is shown in figure 6.14.

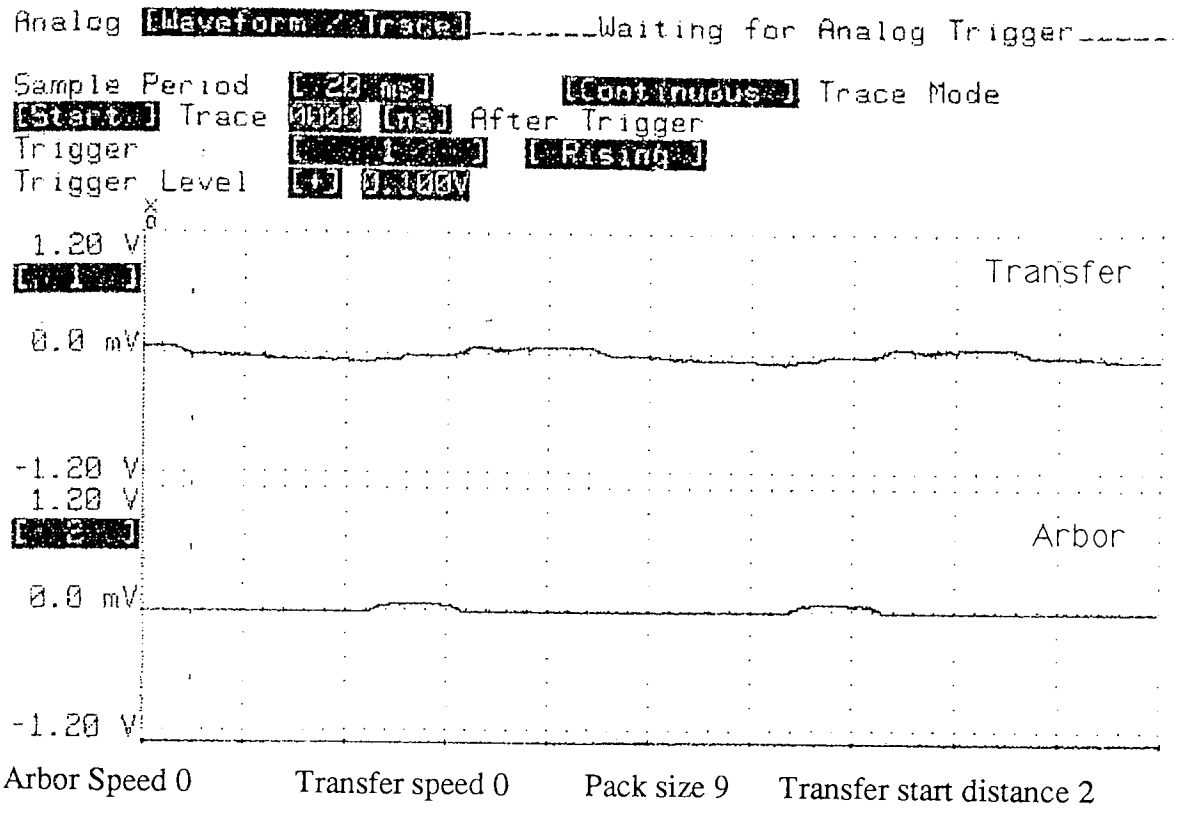
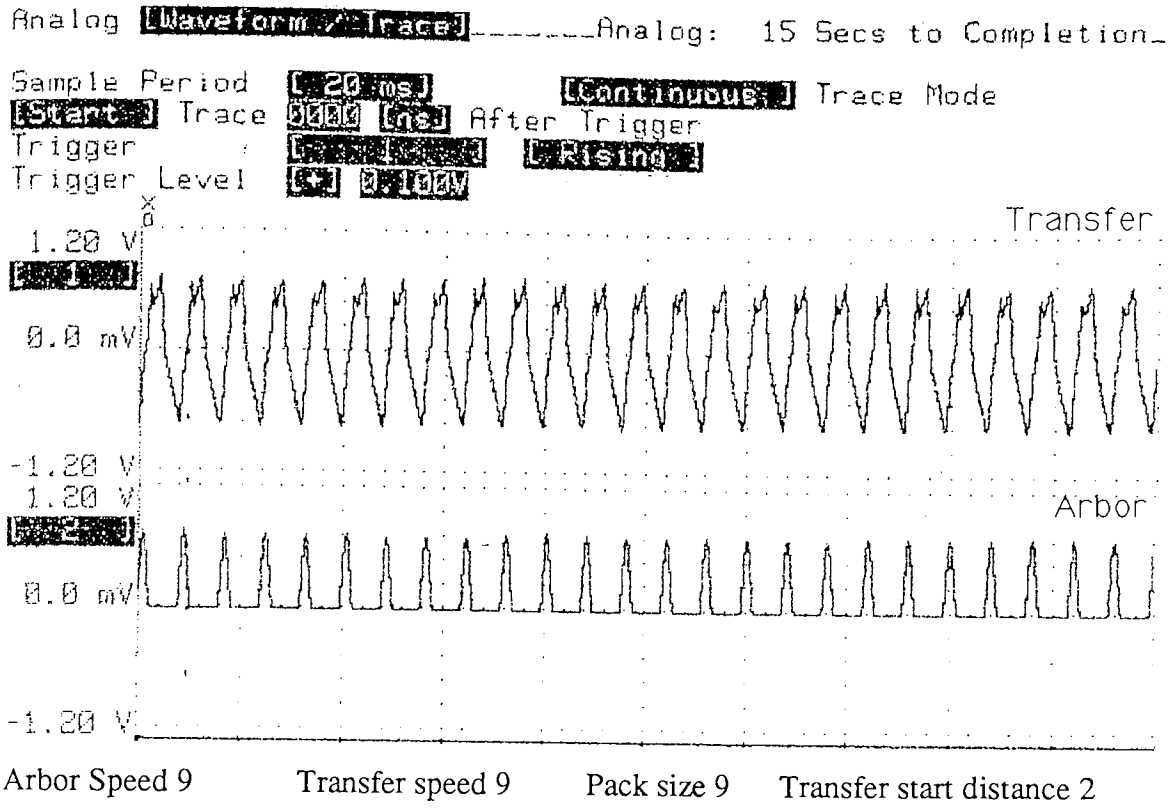


Figure 6.13 Transfer and arbor minimum speed profiles



Arbor Speed 9 Transfer speed 9 Pack size 9 Transfer start distance 2

Figure 6.14 Transfer and arbor maximum speed profiles

Figures 6.10 to 6.14 have shown the operation of the demonstrator under normal synchronisation conditions. However since the transfer start distance is flexible abnormal conditions can be induced. The transfer trace of figure 6.15 shows the arbor still moving when the transfer is preparing to enter the arbor (position A). Under these abnormal and unsafe conditions the transfer decelerates, aborting the arbor enter operation and returns to the rest position (position B).

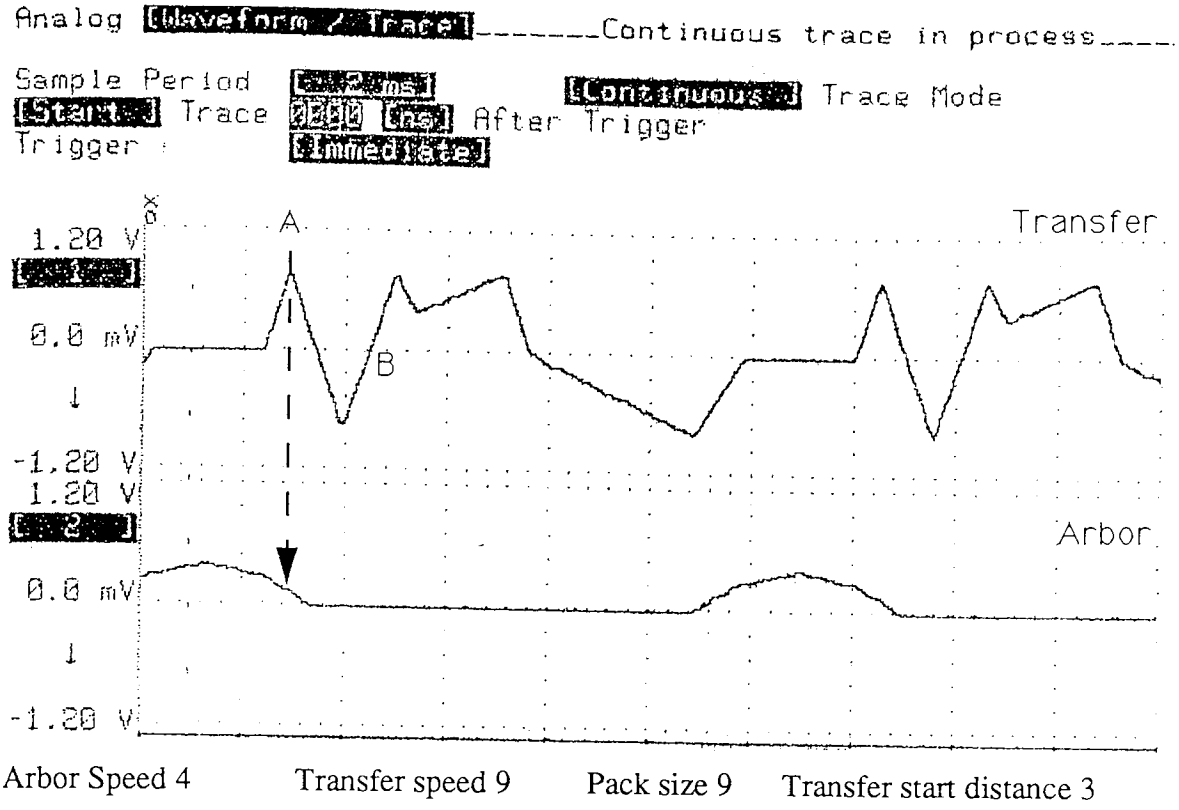


Figure 6.15 Transfer abort enter arbor

In figure 6.15 after the transfer has aborted successfully the arbor has finished moving and the transfer restarts it's motion cycle. However in figure 6.16 the arbor is still moving even after the transfer has reached it's home position, the transfer is therefore inhibited from moving until the arbor stops (position C).

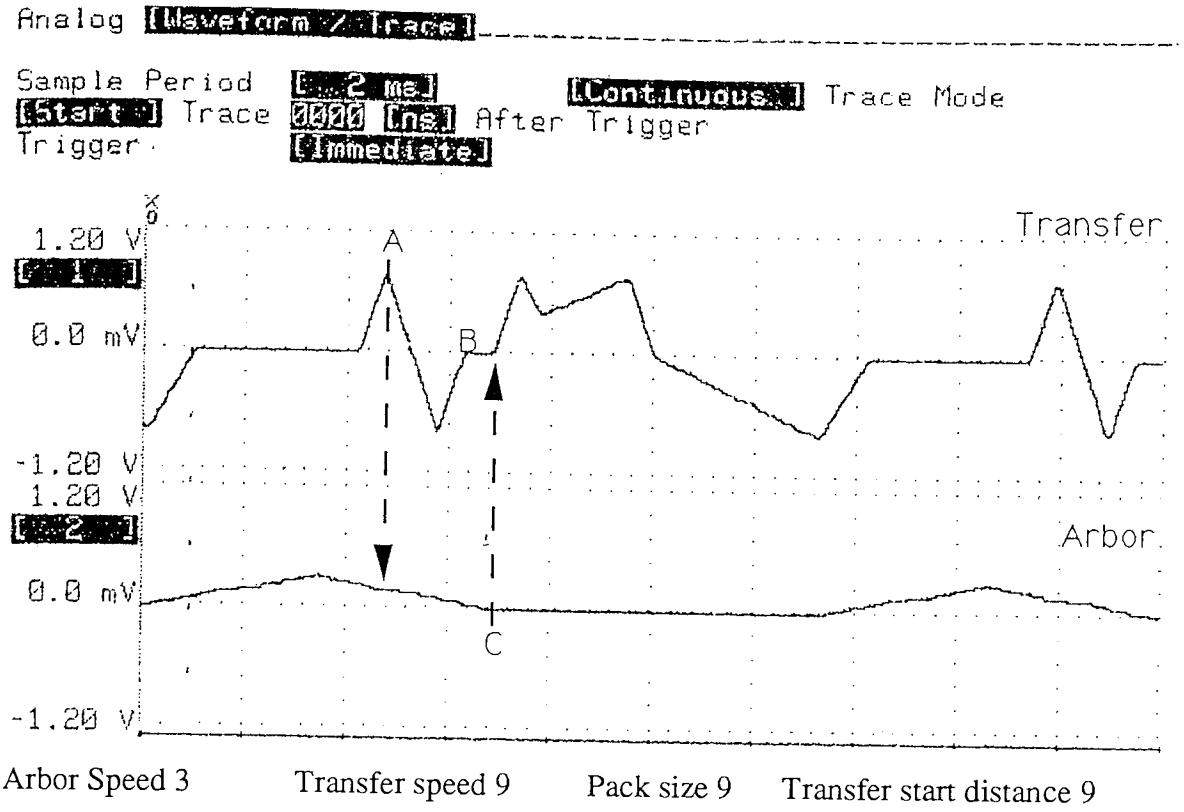


Figure 6.16 Transfer abort enter with delayed restart until arbor at rest

The results of the final implementation controller demonstrates the capability of IDM machines to provide dynamic flexibility in operation for velocity, position and synchronisation. The validated synchronisation logic provides fail-safe, recoverable operation of the two IDMs. The operation of the machine follows the operations predicted by the Petri-net of figure 4.3 and reachability tree of figure 4.6. This comparison verifies the controller implementation against the machine requirements specification.

6.2.3 Discussion of results

The fast-prototype demonstrator controller was constructed from proprietary motion controllers and a module synchronisation controller based on executable specifications derived from a Petri-net FSM model of machine requirements. The controller provides intermittent motion control with discrete-synchronisation for the demonstration machine. Motion control flexibility was controlled by local parameters for each IDM,

synchronisation was based on position of mechanisms allowing time to be removed from the synchronisation logic.

The prototype gave confidence in the suitability of the electromechanical servo-drives and associated mechanisms to meet the machine requirements. The construction of complex motion profiles from motion segments derived from specifications was easily accomplished, however flexibility in motion profiles for speed and position required reprogramming. Occam proved to be suitable for expressing Petri-net specification models as executable specifications, a close transformation between specification model and executable implementation was possible.

The final implementation demonstrator controller expanded on the features of the prototype by allowing full use of the specified IDM flexibility. In particular the dynamic alteration of position and velocity parameters of specified motion segments allowed velocity and position changes to be made to the IDMs complex motion profiles while the machine was running. These profiles could also be modified according to the state of the IDMs allowing fail-safe, recoverable operation of the machine.

The use of defined motion segments in the implementation allowed direct use of motion specification tables. The tables were employed by a standard motion-generation routine allowing the reuse of the same software structure across different IDMs, providing modularity in the Occam controller.

The use of IDM position as a synchronisation scheduling parameter allowed the removal of timing constraints from the synchronisation logic. The synchronisation logic could accommodate abnormal machine conditions without requiring knowledge of IDM velocity. Inappropriate operation of each individual IDM being detected and dealt with by invoking either fault-tolerant or fail safe synchronisation strategies. The simple nature of the demonstrator operation made discrete synchronisation and intermittent motion control of the IDMs sufficient, however more complex mechanisms requiring continuous synchronisation could not use this method.

Occam proved to be suitable for real-time control at high sample rates (greater than 1ms). The language required no operating system to support inter-process communication, however some of the visibility of the design was lost due to the point-to-point communication based on channels. The language does not support explicit scheduling of processes but can accommodate a simple prioritisation of tasks, these features allow prototypes to be created quickly with few overheads compared to a separate

language/operating system such as 'C' and OS-9. For a single processor executing many Occam processes the limited scheduling capabilities of the language may result in incorrect operation.

The Occam source code which implemented the controller proved to be large, the main body of the code was dedicated to the concurrent processes comprising the two IDM controllers. Much of the code was necessary to support fault tolerant synchronisation communications between IDMs, since a change in the sequence of inter IDM communications could lead to deadlock or livelock in the IDM control processes. However many of the IDM controller code blocks have a generic structure allowing reuse of code modules by in line block copies for each IDM. A commercial development using Occam would need to use code compression techniques such as library mechanisms to make the code more manageable.

The Transputer was capable of supporting the real-time Occam code, however the limited support of hardware vendors to allow connection to plant was a disadvantage. The Transputer development system programming support environment has powerful editing tools but code generation and project support tools are primitive. The system development shortcomings of TDS, coupled with the lack of peripheral hardware support and the absence of deterministic scheduling in the Occam language makes the TDS/Occam/Transputer unattractive in commercial ventures. However, the design methodology which lead to the Occam/Transputer implementation is valid for other hardware/software implementations.

6.3 Conclusions

This Chapter has described the fully flexible implementation of the IDM demonstration machine computer controller. The Occam implementation being a close representation of the DFD and Petri-net design models described in Chapter 5.

Experimental results of two controllers of the two IDM demonstrator machine were detailed. A prototype IDM controller demonstrated IDM flexibility and established the suitability of the drives and mechanisms selected for the demonstration machine. The final implementation controller demonstrated the full flexibility of the IDMs by displaying flexibility in position and velocity control of individual IDMs and flexibility in synchronisation between IDMs to support fault-recoverable and fail-safe IDM operation.

The results of the experimental machine were compared with the motion specifications of Chapter 3 and the validated synchronisation specifications of Chapter 4 to verify the capability of the controllers to meet specifications. The controllers and IDMs proved capable of achieving the required motions and synchronisations whilst also demonstrating the envelope of flexibility in both motion profiling and synchronisation.

Position based discrete synchronisation together with intermittent motion control for each IDM using local parameters for speed and distance flexibility proved capable of achieving the specified demonstrator operations. The Occam controller was created specifically to support intermittent motion IDMs linked by discrete synchronisation, a different implementation structure of cooperating processes would be required for continuous synchronisation applications.

Modularisation of the Occam implementation followed the partitioning of data processes in the high level DFD models. Lower level DFD models did not support such a clear correspondence between DFD data processes and Occam procedures. The Occam language did allow a close mapping of the Petri-net design models of discrete synchronisation to inter process channel communication in the implementation code. However point-to-point channel communication removed the visible data stores of the DFD design, encapsulating data in Occam procedures, this made diagnostics and commissioning of the code more difficult, additional diagnostics code being required. In addition exceptional synchronisation communication conditions required extensive code to avoid deadlock and livelock of cooperating IDM control processes. The code complexity increasing with the number of inter IDM synchronisations. The resulting Occam source program code was large, however modularity in the design allowed code reuse for each IDM.

A full discussion of the final implementation of the machine controller together with experimental results can be found in the main body of this Chapter. A critical examination of the IDM machine specification, design and implementation process is presented in Chapter 7.

----------//----------

Chapter 7

Conclusions

7.0 Introduction

This Chapter reiterates the aims and objectives of the research. It draws together the tools and methods used in the thesis and critically examines their suitability in the specification, design and implementation of computer controllers for multiple independent drive mechanism high-speed machines. Areas for further research are identified and final conclusions on the research are drawn.

7.1 Aims and objectives

The goal of the research was to derive generic methods for the specification, design and implementation of modular computer controllers for flexible, high-speed manufacturing machines constructed using multiple independent drive mechanisms. The thesis builds on existing research into independent drive mechanisms by examining the specification and design of IDM machines to create a generic computer controller development method for these machines.

7.2 High speed machines

In Chapter 1 the characteristics of high speed machines were introduced together with the conventional machine design method employing a central prime mover and complex mechanical linkages. The IDM machine design approach using multiple servo-controlled drives was shown to reduce the complexity of mechanical mechanisms whilst introducing flexibility in machine function. Computer software was identified as being fundamental to the control of the IDMs in the machine. It was shown that IDM machines require a reliable, fail-safe computer controller which could provide the required IDM flexibility.

The absence of specification methods for IDM machine designs was highlighted together with the ramifications of errors, omissions and ambiguities in these specifications for the computer controller. The following were identified as areas for research: methods of capturing machine requirements within a multidisciplinary environment; methods of requirements validation to support quality and safety; and methods for design and implementation of computer controllers tailored to the needs of a modular IDM machine.

7.3 Literature and technology review

Chapter 2 discussed the technology and development methods required to construct an IDM machine and provided a critical literature review. Other multiple drive applications such as robotics, CNC machines, PLC systems and existing IDM machine systems were discussed and it was shown that in order to achieve computer control of IDM machines off line programming techniques together with modular software and distributed hardware are necessary.

To support an IDM machine computer controller development a set of common tools and techniques was proposed based on easily assimilated DFD and Petri-net graphical notations together with associated text, databases and diagrams. These notations support: machine requirements capture and specification within a multi-disciplinary engineering project; the validation of functional specifications using constraints checking and Petri-net FSM models; a computer control system design using concurrent FSMs and DFDs; and an implementation based on close transforms of Petri-net FSM and DFD designs into real-time, concurrent software. In addition executable prototypes constructed from development models were proposed to identify design faults and implementation constraints during controller development.

The IDM development method derived was exercised by the creation of an experimental IDM machine controller. This demonstration machine comprised intermittent motion IDMs linked by discrete synchronisation. The machine utilised two IDMs and was used to illustrate the flexibility of IDM machine designs. The experimental results of the demonstration machine enabled comparison between observed-operation, machine specifications and Petri-net models of machine performance.

7.4 Requirements capture

A method for capturing IDM machine requirements was presented in Chapter 3. The proposed method used a sequence of capture phases where each phase gathered requirements from a different viewpoint or source. Each stage built on data gathered for the previous stage. After all capture stages were complete a hierarchical set of requirements was available for the IDM machine. Requirements detail increases down the hierarchy making the requirements useful to all since the requirements may be examined at the level of detail appropriate to the reader. The hierarchical structure also allowed parent-child relationships between requirements to be expressed, this would enable changes in high level requirements to be propagated down through the requirements data by a well defined path.

The six sequential phases necessary to gather pertinent machine requirements are: user requirements for product and process; product forming operations; mechanical modules to perform the product forming operations; synchronisation of mechanisms; motion of mechanisms; and independent drive requirements of mechanisms.

Methods for partitioning and labelling components of these steps were introduced. In the product operation step manufacturing operations were labelled as fabrication, transportation or safety/quality operations according to function. These product operations were then grouped into manufacturing modules using the concepts of intermediate-product, common-synchronisation and fabrication sequence partitioning. The logical relationships between mechanisms were defined by describing motion and synchronisation between mechanisms as hierarchical synchronous, parallel synchronous and parallel asynchronous relationships.

A method of generating complex motions from a sequence of simple motion segments, parameterised to allow flexibility in position and velocity was introduced.

The captured requirements require validation and structuring in order to provide a correct, complete, unambiguous and feasible machine specification.

7.5 Requirements validation and specification

Methods of validating captured machine requirements based on cross-referencing and constraints testing on the machine database and the creation and analysis of Petri-net models of machine requirements were presented in Chapter 4. A Petri-net interaction/motion model of machine requirements was created and analysed using a reachability tree. The execution of the Petri-net model was validated by a critical review of performance to identify errors in the modelled requirements. Validation was necessary to allow a complete, correct and feasible set of requirements to be produced in the form of a machine requirements specification document.

The machine requirements specification document forms a hierarchical decomposition of functional requirements for the IDM machine, the highest levels of the hierarchy describe machine operation and product to be manufactured, while the lowest levels describe the motion profiles and drive requirements of individual IDMs in the machine.

A significant problem associated with validation of requirements is the lack of automated procedures for requirements testing. This results in the system developer being responsible for the thoroughness of the validation process. Different representations of requirements may make checking for consistency between representations difficult, however the use of Petri-nets as a notation for combining and structuring requirements into coherent models has been demonstrated.

7.6 Design of an IDM computer controller

Chapter 5 detailed a generic model for the computer controller of an IDM machine, described using a DFD notation and based on a hierarchy of control within a distributed, modular software structure. The controller model was partitioned by data rate and complexity into three sections defining machine, manufacturing module and IDM levels within the controller. At each level finite state machine and process logic functions were defined. Inter & intra level communication was supported by visible, well-regulated interfaces.

An essential model design using the generic template was derived for the demonstration machine. An implementation of the demonstrator design tailored for the Transputer

microprocessor and Occam programming language was described. Prototype screen layouts with user commands and machine reports were detailed and a prototype machine controller using Occam for inter IDM synchronisation together with proprietary motion control introduced.

Inappropriate use of flexibility in IDM requirements was shown to be capable of introducing unsafe dynamic performance in the demonstrator IDMs during the requirements validation phase. Therefore a careful implementation of the Occam controller synchronisation, to match the Petri-net FSM specification model, was required in order that the controller could recover from unsafe plant states.

7.7 Flexible controller implementation and experimental results

An implementation coded in Occam which could accommodate the full flexibility of the IDMs specified was described in Chapter 6. The Occam/Transputer implementation allowed an executable form of the machine requirement specifications to be created. Once initial code modules had been constructed motion specifications could be inserted as application specific data into a generic motion-generation module. Individual interaction and motion scheduling modules for each IDM were required but these models could be constructed in a straightforward way from Petri-net specification models. The decomposition of low level DFD designs into Occam procedures proved more difficult. The complete source code of the full flexibility IDM controller can be found in the appendices.

The results of the prototype controller verified that the servo-drive hardware was capable of performing the dynamic motion profiles required by the machine specifications, it also demonstrated the feasibility of generating complex motions from a sequence of simple motion segments. The experimental results of the full flexibility IDM machine controller proved the ability of Occam and the Transputer to perform real-time control. It also demonstrated the full IDM flexibility in position, velocity and synchronisation while providing fail-safe, recoverable machine operation.

The Occam/Transputer implementation was only suitable for intermittent motion, discrete synchronisation based on position. This implementation could not accommodate continuous synchronisation requiring the expression of common timing across IDMs. The Occam source code was large, much of the code being required to accommodate

abnormal operation of IDMs and associated failures in inter-IDM controller communications.

7.8 Assessment of IDM machine development method

The method of using a sequence of requirements capture phases worked well when applied to small projects such as the demonstrator, when the machine requirements for machine function, hardware and software were well understood. Machine applications with more unknowns proved harder to manage since uncertainties in high level requirements for product and process made requirement refinement difficult since many pivotal requirements were absent or subject to change. This was due to the top-down, pre specified approach of the method.

The data flow diagram notation was applicable to many different system models including product flow, mechanical and electrical partitioning. However the notation lacked depth for specific applications requiring textual extensions to support the basic diagram. DFDs have been used as a common notation between development engineers to show modularity and connectivity in the machine, the graphical notation being readily accepted by engineers from all disciplines. CASE tools to support the DFD notation, such as Easycase, allowed hierarchies of DFDs to be created and related. In addition the CASE tools allowed improved diagram and document management.

The Petri-net notation proved to be easily understandable by electrical, mechanical and software engineers, however they are limited to representing control and sequence of IDM FSMs and cannot model data process detail. Petri-nets proved capable of modelling limited size state-machines, with larger models state-explosion and the increase in associated document size made model management a problem. Automated methods of creating, storing and executing Petri-net models would be necessary for use on large machine projects. Petri-nets were used successfully in abstracting specific control functions such as synchronisation safety. The smaller model was more easily created and the results of exercising the model were more amenable to analysis by inspection.

There was often a large amount of additional information required before executable Petri-net models could be developed. This was the case when modelling external connections to the control system Petri-net such as actuators, sensors and operator inputs.

This has been addressed by developing models of the machine plant and the operator, however these asynchronous models can be difficult to represent in Petri-nets.

Limited size Petri-nets proved valuable in predicting behaviour by executable models and as a template for controller design. Direct representation of these executable models by Occam was straightforward, the primary deficiency of Occam being the lack of support for guarded outputs. This made deadlock of concurrent processes a possibility if inter process communications failed.

The use of prototypes alongside the main machine development can resolve some of these difficulties by allowing 'bottom-up' examinations of pivotal requirements related to implementation issues in parallel with the main IDM development.

The definition of a sequence of capture stages and the use of documented deliverables proved valuable in assessing progress of the machine development, and in identifying areas of uncertainty in the machine requirements. The method requires a central repository for design information supported by specification documentation. This improved accessibility to design information since the contents of the specification documents covered all engineering disciplines, was well structured and indexed.

7.9 Assessment of the Occam/Transputer IDM machine controller implementation

The demonstrator was an example of a single type of motion and synchronisation, intermittent motion with discrete synchronisation. The Occam/Transputer implementation of maximum flexibility in motion and synchronisation together with high modularity was possible due to the inherent loose coupling of the machine control required. The demonstrator displayed the full scope of IDM flexibility including the problem of inappropriate use of flexibility causing erroneous IDM operation. The Occam implementation was a close representation of the generic software design for IDM machines described in Chapter 5.

Different modes of synchronisation with close coupled motion control such as hierarchical synchronisation based on software master motion profiles could not support the levels of flexibility exhibited by the demonstrator.

The modularity of the software controller was tailored for intermittent motion and discrete synchronisation. Each IDM having separate motion-generation and synchronisation processes. For a continuous synchronisation machine a centralised motion-generator accommodating all continuous synchronisation IDMs would be more suitable.

The Occam physical source code size was great considering the simple nature of the demonstrator much additional code was required to accommodate synchronisation failures between IDM control processes. Occam and the Transputer proved capable of real-time control but is limited by the code development environment and lack of Transputer interface hardware suitable for machine control.

7.10 Assessment of the IDM machine development method in commercial projects

Components of the IDM machine development method have been used in commercial machine projects. In particular DFD and Petri-net notations have been applied to requirements capture, specification and design of IDM machines. Machine controller implementations based on a hierarchical structure of multiple state-machines with associated process logic have been used. Molins are using DFD and Petri-net notations in new machine developments for capture of physical machine context using product flow, and description of modes of machine operation. Practical experiences of the IDM approach to machine design and the use of the IDM development tools follow.

A principle benefit of the IDM machine development method is the provision of intuitive notations which engineers can use for presenting models of requirements. These models provide a basis for communicating requirements allowing discussion and refinement across multiple engineering disciplines. The DFD and FSM notations fit within the existing framework of tools used in machine design and can easily be introduced into the design process.

The IDM development method's ordering of requirements capture stages is tailored for a small design team. Large design teams with the capability for more concurrent development can be hindered by bottlenecks in the capture process. However the IDM design method requires a greater regulation of the design process, which allows improved project management.

The definition of terms applying to IDM machines to describe motion and synchronisation have proved useful in design meetings. They have been used when classifying mechanisms and identifying common synchronisation and motion modes between machine components. After an initial selection of synchronisation modes for servo-drives a common method is often chosen to aid implementation, particularly where technological constraints on hardware are present. Parallel synchronous synchronisation based on a common timebase is used in many commercial machines where the mechanical modules are initially identified using fabrication sequence partitioning. When creating IDM modules common synchronisation partitioning is often used to group operations into IDM axes.

Most IDM machines require continuous synchronisation unlike the IDM demonstrator described in the thesis. This form of synchronisation requires links between synchronisation and motion generation functions and requires the IDM controllers to have a concept of time. In an implementation global system clocks remove synchronisation flexibility while also making modularisation and distribution of IDM motion control functions unattractive due to the requirement to propagate timing information. As a result commercial machines implement modularity by locating groups of IDM controllers on a single processing node, allowing propagation of timing between cooperating controllers on a fast local connection. This is a different form of software modularity to that originally envisaged. Figure 2.2 shows the modularity in a typical commercial machine.

The software control described in the thesis concentrates on computer motion control of servo-drives. However a significant amount of binary sensor input and actuator control is required in most commercial machines. The Petri-net state-machine notation is well suited to defining this form of control where a sequence of binary outputs depends on a sequence of binary inputs. This type of Petri-net closely resembles the discrete synchronisation Petri-net shown in figure 4.2. The application of Petri-nets to the specification and modelling of binary sensor/actuator synchronisation is being used in industry. An implementation of this form of control is ideally suited to Programmable Logic Controllers. Current commercial machine developments are incorporating networks of PLCs providing distributed sequence control together with a centralised processing node for high-speed complex motion profiling of multiple servo drive IDMs. The main processor node often utilises common timebase synchronisation, the PLCs using discrete event synchronisation, both types of synchronisation being overlaid onto a common machine-cycle.

Integration of tools and notations into the design team has met with difficulties. The proposed IDM machine design method is essentially a pre specified top down approach which attempts to decompose functionality into a hierarchy of increasing detailed levels. The lower levels building on the requirements specified in higher levels. The IDM machine development method imposes a structure and guides the system analyst to address design issues in the appropriate design phase. However with new machine developments there are many unknowns in the user product and process requirements. In this situation it is difficult to proceed with a refinement of the requirements since pertinent information may be absent, incomplete or subject to change. It has been found that prototypes of designs can aid the gathering of requirements by displaying functionality to allow a critical assessment of performance. However it is easy to allow the prototype to changes roles into a system implementation. This should be avoided since inherent problems with poor design and code structure, inadequate documentation and missing functionality due to unstructured validation and testing make the implementation unwieldy, difficult to maintain and error-prone.

Molins has a method of iterative prototyping based on application independent, modular, code primitives tailored for IDM machines written in the programming language 'C'. These primitives can be brought together in an appropriate harness to create fast prototypes. Most of the underlying code of such a prototype is reusable and extensively tested. Functionality can be added since the primitive functions are modular with well defined interfaces. Each primitive is customised according to the machine requirement specifications. For example state machine primitives may implement a Petri-net specification model by creating corresponding states and transitions in the primitive using the model as a template. This method allows the machine designer to concentrate on application specific matters. The prototypes can be validated and verified against specifications for the functions which they perform.

Careful regulation of the prototypes is necessary to avoid incomplete implementations, multiple prototypes become building blocks for the final system rather than a single prototype evolving. The final machine controller must still be created, validated and verified but it can use several validated prototype modules in it's construction.

Most very high-speed machines manufacture a single product, the flexibility of IDMs for product changes are not needed by such machines, the IDMs are used to provide functionality unobtainable with conventional technology. An IDM machine may also achieve higher production speeds by designing the manufacturing operations around product flow.

Current awareness in the engineering fraternity of the possible flexibility in IDM machines is limited. As a result complex mechanical assemblies may be designed unnecessarily, equally over complex servo-drive systems may be used where a simpler mechanical connection would be sufficient.

Design engineers working in an integrated engineering discipline environment may not be aware of the full implications of such an integrated approach. The necessity for design management particularly with reference to changes to the machine requirements specification is not fully appreciated, especially where a problem in one discipline may be solved by changes in another.

Modifications are inevitable in any machine development however it is most important to ensure that modifications are not performed in an ad-hoc manner and that the ramifications of changes are understood in the associated engineering disciplines. The development engineers need to work closely under careful project management which will identify and reject inappropriate modifications in order to minimise consequential re-engineering from changes to specifications. For smaller machine developments unautomated methods can support the requirements capture and specification documents but for larger projects automated documentation storage and version control management may be necessary.

7.11 Further work

The development of methods for designing computer control systems for IDM machines is an evolutionary process. The research has identified sources of requirements and introduced classifications of IDMs to aid requirement specification, refinement to these classifications will come from further use in industrial projects.

DFD and FSM notations have been used throughout the development phases to avoid transformations between notations when creating designs from requirements, however inconsistencies between data representations may still occur. Automated methods for maintaining consistency and/or detecting inconsistency between data representations would enable more thorough validation and verification of specifications and designs.

The validation of specification data has been supported in the thesis by Petri-net modelling and constraints checking. The use of formal methods to mathematically prove qualities of machine control logic would provide a greater degree of confidence in the specification and subsequent controller implementation.

Modular implementations have been created which use requirements specification data directly, however these implementations require hand coding of the data from specification to implementation. The generation of the implementation from the machine requirements may be automated to reduce the lead-time of the controller development process. This technique would also facilitate the generation of fast-prototypes of controllers to examine machine requirements.

7.12 Conclusions

The thesis has presented a structured development method for the creation of computer controllers for multiple independent drive mechanism machines. The method uses data flow diagrams and finite state machines for the capture, annotation and validation of machine requirements for IDM machines to produce a machine requirement specification. A generic software structure design for the creation of modular machine controllers for IDM machines has been described. An implementation based on the Occam programming language and the Transputer microprocessor was created using the IDM development method for a two IDM demonstration machine.

The results of this machine demonstrated the capability of software controlled multiple IDMs to provide manufacturing operations with flexibility in motion control and synchronisation under both normal and abnormal operating conditions. The demonstrator controller providing fault-tolerant and fail-safe operation under abnormal plant conditions.

An analysis of the IDM development method and the application of Occam and the Transputer to IDM machines has been presented. Experiences using components of the method and it's associated DFD and Petri-net tools in commercial machine projects have also been described.

The application of the tools and notations of the IDM development method to commercial machine developments demonstrates their suitability for this task. The spread of IDMs

into all forms of machine designs continues and the maturation of techniques for specifying requirements, designing and implementing computer controllers for these machines have proved their worth in industry by improving the quality of machine specification, computer controller design and implementation.

----------//----------

References

[1] Jarvis J.W., 1986, "Computer integrated manufacturing technology and status", IEE publications, ISBN 0-85296-337-8.

[2] Chen F.Y., 1982, "Mechanics and design of cam mechanisms", Pergamon Press, ISBN 0-08-028049-8.

[3] Hall R., 1978, "The making of Molins, the growth and transformation of a family business 1874-1977", Molins Ltd, Evelyn St, London.

[4] Fenny L., Draper C.M., Holding D.J., 1988, "Modular machine systems", Proc SERC conference on high-speed machinery, IMechE.

[5] Strandh S., 1979, "The history of the machine", Bracken Books, ISBN 1-85170-333-0.

[6] Fogarasy A.A, Smith M.R., Sadek K., Allonby N., Daadbin A., 1988, "Design and development of high speed packaging machines", Proc SERC conference on high-speed machinery, IMechE.

[7] Rosegger G., 1986, "The economics of production and innovation: an industrial perspective", Pergamon, ISBN 00803-3958-1.

[8] Russell S., Liff S., 1988, "Technology Monitor 4", Aston University Business School, Technology Policy Unit, ISBN 1-85449-000-1.

[9] Leighton N.J., Jones B., and Fletcher H., 1988, "Precision press feed system using digitally controlled drives", Proc SERC conf on High Speed Machinery, IMeche.

[10] Hidde A.R., Marx U., 1991, "Flexible automation - fast signal transmission in a production line", Mechatronics, Vol 1 No 1, pp 19-35.

[11] "TISTAR Integrated plant control", Texas Instruments Ltd, Manton Lane, Bedford.

[12] Boehm B., 1981, "Software engineering economics", Prentice Hall, ISBN 0-13-822122-7.

[13] Johnson T.L., 1987, "Distributed hierarchical control architectures for automation", OPA Amsterdam B.V.

[14] Seaward D.R., 1989, "Continuous phase synchronised drives (for a rod making machine)", Phd Thesis, Aston University.

[15] Fenny L., 1989, "Flexible high-speed machinery", Phd Thesis, Aston University.

[16] Stamp K.J., Rees Jones J., 1988, "High speed planar guidance coordination with programmable motion control", Proc SERC conf on High Speed Machinery, IMeche.

[17] 1989, "Intelligent motion control and industrial plant automation", Quin Systems Ltd, 35 Broad St, Wokingham.

[18] Williams T.J., 1983, "Developments in hierarchical computer control systems as affecting industrial manufacturing systems of the future", Computer Applications In Production Engineering, IFIP.

[19] Holding D.J., Fenney L., Foster K., 1987, "The use of Occam in the specification, simulation, synchronisation & control of modular machines", Simulation Council, Conf on Computer Simulation, Bangor.

[20] Leveson N.G., 1986, "Software safety: why, what and how", Computing Surveys, Vol 18, No 2, June.

[21] Hannah J., Stephens R.C., 1984, "Mechanics of machines: Elementary theory and examples", Edward Arnold, ISBN 0-7131-3471-2.

[22] Noble D.F., 1984, "Forces of production: a social history of industrial automation", Knopf INC, ISBN 0-394-51262-0.

[23] Ziemba R., 1989, "Use of a Programmable Logic Controller (PLC) for temperature, position, velocity and pressure control of injection moulding machinery", Industrial applications society general meeting conference record.

[24] Leskiewicz H.J., 1981, "Pneumatic and hydraulic components and instruments in automatic control", Pergamon, ISBN 0080-27317-3.

[25] Warwick K., 1986, "Industrial digital control systems", Peter Perigrinus, ISBN 0-86341-081-2.

[26] Leveson N.G., Stolzy J.L., 1987, "Safety analysis using Petri-nets", IEEE Trans On S/Ware Eng Vol SE-13, No3.

[27] Nazemetz J.W., Hammer W.E., Sadowski R.P., 1985, "Computer integrated manufacturing systems: selected readings", Industrial Engineering and Management Press, ISBN 0-89806-066-4.

[28] Andeen G.B., 1988, "Robot design handbook", Donnelley, ISBN 0-07-060777-X.

[29] Naghdy F., Strickland P., 1989, "Distributed manipulation environment- a Transputer based distributed robotic system", Int journal Computer Integrated Manufacturing, Vol2, No5, pgs 281-289.

[30] Zalzala A.M.S., Morris A.S., 1989, "A Fast Tracker For Intelligent Robot Manipulators", Proc Int Conf. Application of tranputers, SERC, Liverpool.

[31] Braganca C.A.J., Sholl P., 1985, "VAL-II a language for hierarchical control of a robotised automated factory", Robotica Vol 3.

[32] Grotzinger .S., 1988, "A Petri-net Characterisation of a high speed placement machine", IEEE PROC of 38th Electronic Component Conf, IEE CAT.NO 88CH2600-5, pgs 64-8.

[33] Paul R.P., Shimano B., Mayer G.E., 1981, "Kinematic control equations for simple manipulators", IEEE Trans Syst Man Syber, Vol SMC-11, pp449-55.

[34] Kreysig E., 1983, "Advanced engineering mathematics", Wiley, ISBN 0-471-99841-5.

[35] Vail P.S., 1988, "Computer Integrated manufacturing", PWS, Kent, ISBN 0-534-91465-9.

[36] Gibbs D., 1984, "An introduction to CNC machinery", Cassel computing, ISBN 0-304-31169-3.

[37] Gupton J.A.,1986, "Computer controlled industrial machines and processes", Prentice-Hall, ISBN 0-13-165267-2.

[38] Koren Y., 1983, "Computer control of manufacturing systems", McGraw-Hill, ISBN 0-07-035341-7.

[39] Puente E., MacConail P., 1988, "Computer integrated manufacturing", Proc 4th CIM Europe Conference, IFS, ISBN 0-948507-99-3.

[40] Webb J.W., 1988, "Programmable controllers principles and application", Merrill, ISBN 0-675-20452-6.

[41] Warnock I.G., 1988, "Programmable controllers operation and application", Prentice Hall, ISBN 0-13-730037-9.

[42] Fanard A.G., Lobelle M.C., Mulemangabo E.B., 1989, "G++ a graphical language intended to help the development of industrial process control applications", Proceedings of IEE 2nd Int Conference on Software Engineering For Real-Time Systems.

[43] 1989, "A new dimension in programmable control", Telemecanique, University of Warwick science park, Coventry, CV4 7EZ.

[44] Seaward D.R., Vernon G.W., 1991, "Using high performance drives technology - an original equipment manufacturer's point of view", Proc. Eurotech Direct 91 / Machine Systems, IMechE, ISBN 0-85298-776-5.

[45] Hyer N.C., 1987, "Capabilities of group technology", Prentice Hall.

[46] Naylor, G.H.F., 1985, "Dictionary of mechanical engineering. 3rd edition", Butterworths, ISBN 0-408-01505-5.

[47] Belove C., 1986, "Handbook of modern electronics and electrical engineering", Wiley, ISBN 0-471-09754-3.

[48] Considine, 1986, "Standard handbook of industrial automation", Chapman & Hall, ISBN 0-412-00831-9.

[49] Seaward D.R., Vernon G.W., 1991, "The drive for higher speeds", Engineering, Issue 10.

[50] Tal.J., 1989, "Motion control applications", available from Galil Motion Control Inc. CA. 94303.

[51] 1989, "The programmable transmission system", Quin Systems Ltd, 35 Broad St, Wokingham.

[52] "BRU 500 technical reference, brushless DC drives", Electrocraft Ltd, 4th Ave, Crewe.

[53] 1991, "Fairchild product index", Fairchild Industrial Products, 1501 Fairchild drive, Winston-Salem, North Carolina.

[54] 1991, "TTL databook", Texas instruments, Dalla, Texas, ISBN 3-88078-064-1.

[55] SD-SCICON, 1991, "Intercolor industrial monitors", SD-SCICON, Abney Park, Manchester, Cheshire, SK8 2PD.

[56] Lucas Deeco, 1991, "Touchscreen catalogue", Lucas Duralith Corp. 31047 Genstar Road, Hayward, California, USA.

[57] Quick components, 1991, "LCD touchpanel technology", Meridan Centre, King St, Oldham, OL8 1EZ.

[58] British Standards Institute, 1990, "Draft British standard: Functional safety of programmable electronic systems", 2 Park Street, London.

[59] British Standards Institute, 1990, "Draft British standard: Software for computers in the application of industrial safety-related systems", 2 Park Street, London.

[60] Basu A., 1987, "Parallel processing systems a nomenclature based on their characteristics", IEE Proc, Part I, Vol 134, No 31187, pp143-147.

[61] 1988, "DC Motors, speed and servo control. Vers 3", Electrocraft Ltd, 4th Ave, Crewe.

[62] Bennett S., 1988, "Real-time computer control: An introduction", Prentice Hall, ISBN 0-13-762485-9.

[63] Carling A., "Parallel processing the Transputer and Occam", Sigma, 1988, ISBN 1-85058-077-4.

[64] Basu A., 1987, "Design of an adaptable pipeline based on Transputers", Amputero, IEEE, CH2417-4/87/0000/0815.

[65] Conte G.C., Del Corso D., 1985, "Multi-microprocessor systems for real-time applications", Reidel, ISBN 90-277-2054-1.

[66] Dwyer J., Ioannou A.,1987, "MAP & TOP Advanced manufacturing communication", Kogan Page, ISBN 1-85091-355-2.

[67] Dahl O.J., Djikstra E.W., Hoare C.A.R., 1972, "Structured programming", Academic press, ISBN 0-12-200550-3.

[68] Hoare C.A.R., 1978, "Communicating sequential processes", Comm. ACM, Vol 21, No 8, 1978, pp666-677.

[69] Holding D.J., 1988, "Software fault tolerance", Proc Unicom Seminar on 'Failsafe control systems', London, 28-30 June, Kogan Page.

[70] Saridis G.N., 1988, "On the theory of intelligent machines: a survey", IEEE Proc of the 27th conference on decision and control.

[71] "Product handbook", Data translation Ltd, Mulberry Business Park, Wokingham, Berkshire.

[72] "VME bus products", Crellon Microsystems, 3 the business centre, Molly Millars Lane, Wokingham, Berkshire.

[73] 1988, "Transputer development system vers D", Inmos, Prentice Hall, ISBN 0-13-928995-X.

[74] Puusaari P., Sintonen L., Virvalo T., 1989, "In-machine communication - a general solution for interconnecting independent intelligent actuators", IEEE Int conf on communications, BOSTONICC/89.

[75] Miller W., 1987, "OS-9 Operating system manual", Microware Systems Corp. 1900 NW 114th St, Des Moines, Iowa, USA.

[76] 1988, "OS-9/68000 operating system - technical information", The SoftCentre, Vivaway Ltd, Software House, Burr St, Luton, Beds.

[77] Cooling J., 1991, "Meeting the deadlines: real-time executives", .EXE, Vol.5, Iss 8.

[78] 1988, "Versatile real-time executive", Ready systems, 449 Sherman Ave, PO Box 61029, Pal Alto, CA 94306-9991.

[79] Linkens D.A., Virk G.S., 1987, "Computer control", Institute Measurement & Control, SERC vac school, collected lectures, Sheffield, March.

[80] Hoare C.A.R., 1988, "OCCAM 2 reference manual", Prentice Hall, ISBN 0-13-629312-3.

[81] Pountain D., May D., 1987, "A tutorial introduction to Occam programming", Blackwell Scientific Publications Ltd, ISBN 0-632-01847-X.

[82] 1991, "Easycase Computer Aided Software Engineering Tool", Software Construction Company Ltd, The Maltings, Wokingham

[83] Coffin S., 1988, "UNIX the complete reference : System V release 3", McGraw Hill, ISBN 0078812992.

[84] Datem, 1988, "DCM804 Bitbus controller module user guide", Datem Ltd, 148 Colonnade Rd, Nepean, Ontario, Canada.

[85] Lyons, T., NIssen, J., 1986, "Selecting an ADA environment", Cambridge University Press.

[85] Bollinger J.G., Duffie N.A., 1988, "Computer control of machines and processes", Addison Wesley, ISBN 0-201-10645-0.

[86] Aktas, Z.A., 1987, "Structured analysis and design of information systems", Prentice Hall, ISBN 0-13-854571-5.

[87] STARTS, 1986, "SAFRA (an interconnected set of tools) a debrief report", NCC Publications, ISBN 0-85012-574-X.

[88] Birrell N.D., Ould M.A., 1985, "A practical handbook for software development", Cambridge University Press, ISBN 0-521-34792-0

[89] McMenamin S.M., Palmer J.F., 1984, "Essential systems analysis", Yourdon Press, ISBN 0-917072-30-8.

[90] Anderson T., 1985, "Software requirements specification and testing", Blackwell Scientific publications,ISBN 0-6320-1309-5.

[91] Goldsack S.J., 1989, "Specifying requirements: An introduction to the FOREST approach", Dept of computing, Imperial College, London.

[92] McLeod W.T., 1982, "The new collins concise dictonary of the English language", Collins, ISBN 0-00-433091-9.

[93] 1989, "Software management system", Intasoft Ltd, Tresco Houe, 153 Sweetbrier Lane, Exeter, UK.

[94] Looney M., 1986, "CORE: a method for expressing requirements", National computer centre, ISBN 0-850125472.

[95] 1987, "MASCOT Design Support Environment", Alvey Project ref PRJ/SE/44, Ferranti Computer Systems Ltd, Ty Coch Way, Cwmbran, Gwent, NP44 7XX.

[96] 1987, "Official handbook of MASCOT version 3.1 issue 1', Royal signals and research establishment.

[97] Parnas, 1987, "Processes and information". IEE Proc, Part I, Vol 134, No 31187, pp143-147.

[98] "CORAL 66 programming language", Texas Instruments, Dallas, Texas, ISBN 3-88078-064-1

[99] Findlay W., Watt D.A., 1981, "Pascal, an introduction to methodical programming", Pitman, ISBN 0-273-
01714-4

[100] Smith G., 1986, "STARTS - JSD (Jackson System Development) a debrief report", NCC Publications, ISBN 0-85012-549-9

[101] Jackson M., 1983, "System Development", Prentice Hall, ISBN 0-13-88-328-5.

[102] Jackson M., 1991, " Speedbuilder / pdf reference", JSD associates,.ISBN 0-13-88-330-2.

[103] DeMarco T., "Structured analysis and system specification", Yourdon Press

[104] Yourdon E., Constantine L., 1979, "Structured design: Fundamentals of a discipline of computer program and system design", Prentice-Hall.

[105] Ward P., Mellor S., 1986, "Structured development for real-time systems Vol 1-3", Yourdon Press, ISBN 0-917072-51-0.

[106] Myers, 1978, "Composite/Structured design", Van Nostrand Reinhold.

[107] 1989, "DBASE 3+ User manual", MIcrosoft Corp., 16011, NE 36th Way, Box 97017, Redmond WA, 98073-9717.

[108] Avenur A., 1989, "Finite state machines for real-time software engineering", IEE conf Software Engineering For Real Time Systems.

[109] Agerwala T., 1979, "Putting Petri-nets to work", IEEE, Computer, Dec, pgs 85-94.

[110] Carpenter G.F., 1987, "The use of OCCAM & Petri-nets in the simulation of logic structures for the control of loosely coupled distributed systems", Bangor.

[111] 1988, "Computer aided real time design, CARD, tools", Ready systems, 449 Sherman Ave, PO Box 61029, Pal Alto, CA 94306-9991.

[112] Peterson J.L, 1981, "Petri-net theory and the modelling of systems", Prentice Hall, ISBN 0-1366-1983-5.

[113] Chang C.K., Chang Y-F., Song C-C., Aoyama M., 1989, "Integral : Petri-net approch to distributed software development", Information and Software Technology, Vol 32 No 10 pgs 535-545.

[114] Murata T., Komoda N., Matsumoto, 1984, "Petri-based factory automation controller for flexible and maintainable control specifications", IECON '84 Proc IEEE INT Conf Industrial Electronics, Control & Instrumentation, Tokyo, pgs 362-366.

[115] Wang Y., 1988, "A distributed specification model and it's prototyping", IEEE trans on software engineering, Vol 14, No 8.

[116] Draper C.M., Holding D.J., 1989, "The specification and fast prototyping of a distributed real-time computer control system for a modular independently driven high-speed machine", Proc IEE 2nd Int. Conf. Software engineering for real-time systems, Cirencester.

[117] Draper C.M., Holding D.J., 1989, "Specification and verification of the real-time synchronisation software for a modular independently driven high-speed machine", IEE Colloquium on Control system software reliability for industrial applications.

[118] Willson R.G., Krogh B.H., 1990, "Petri-Net Tools For The Specification And Analysis Of Discrete Controllers", IEEE Trans On Software Eng Vol 16 No 1.

[119] 1989, "EXCEL spreadsheet user manual", MIcrosoft Corp., 16011, NE 36th Way, Box 97017, Redmond WA, 98073-9717.

[120] 1990, "CAMLINKS user manual", Limacon, Meadow Farm, Horton, Malpas, Cheshire, SY14 7EU.

[121] 1989, "SPADE - Southampton program analysis and development environment", PVL Ltd, 34 Bassett Crescent East, Southampton, SO2 3FL.

[122]1988, "MALPAS State of the art software verification and validation", RTP Ltd, Newnhams, West St, Farnham, Surrey, GU9 7EQ

[123] Gerrard C.P., Coleman D.C., Gallimore R.M., 1990, "Formal specification and design time testing", IEEE Trans on software eng. Vol 16 No 1.

[124] Merlin P.M., Farber D.J., 1976, "Recoverability of communication protocols - Implications of a theoretical study" IEEE trans on communications, Sept.

[125] Holliday M.A., Vernon M.K., 1987, "A generalised timed Petri-net model for performance analysis", IEEE Trans Software Eng Vol SE-13, No12.

[126] Sagoo J.S., Holding D.J., 1990, "The specification and design of hard real-time systems using timed and temporal Petri-nets", North Holland, Microprocessing and Microprogramming 30, pgs 389-396.

[127] Holding D.J., Hill M.R., Carpenter G.F., 1988, "The design of distributed, software fault tolerant. real-time systems incorporating decision mechanisms", Microprocessing and Microprogramming 24, 801-806.

Appendices

8.0 Prototype programs and results for evaluating independent drive performance

A test program to perform a triangle move inside the IDM displacement constraints was written. Displacement constraints are a mechanical design constraint. The value of velocity used was increased until the drive could not achieve continuous performance, when the motor halted due to overcurrent protection. The value of velocity that could be sustained reliably was then noted. This is the maximum velocity for the IDM in one IDM cycle. The maximum acceleration is found using the following formula from Newton's laws of motion under constant acceleration.

$$\text{acceleration} = \frac{(\text{Max velocity})^2}{2.\text{displacement}} \quad \text{(zero initial value)}$$

Arbor drum test program

Electrocraft Propos-E motion control program

```
001   A     RPE   0
002         P/V   250   6000
003         P/V   250   0
004   A     RPT
005         END
```

Arbor drum results

Actuator constraints	
Actuator cycle displacement	500ec
Maximum velocity inside cycle	6000ec/s
Maximum acceleration inside cycle	72,000 ec/s^2

Transfer mechanism test program

Electrocraft Propos-E motion control program

```
001   A     RPE   0
002         P/V   700      75,000*
003         P/V   700      0
004         P/V   -700     -8,000
005         P/V   -700     0
006   A     RPT
007         END
```

* limit of velocity from computer controller (Propos-E) not motor.

Transfer mechanism results

Actuator constraints	
Actuator cycle displacement	1,400ec
Maximum velocity inside cycle	75,000 ec/s
Maximum acceleration inside cycle	3,700,000 ec/s^2

8.1 Demonstrator machine prototype programs for Occam synchronisation and Propos-E motion control

Prototype Propos-E programs

Transfer mechanism

normal operation
position / velocity flexibility parameters given as 'max (min)'

```
001   A     RPT   0
002         P/V   175          24000 (2400)
003         P/V   300 (192)    12000 (1200)
004         P/V   700 (858)    24000 (2400)
005         P/V   175          0
006         P/V   -1085        -24000 (-2400)
007         P/V   -315         0
008         DWL   120
009   A     RPT
010         END
```

abort enter arbor

```
001   A     RPT   0
002         P/V   175          24000
003         P/V   175          0
004         P/V   -175         -24000
005         P/V   -175         0
006         DWL   50
007         P/V   175          24000
008         P/V   300          12000
009         P/V   700          24000
010         P/V   175          0
011         P/V   -1085        -24000
012         P/V   -315         0
013   A     RPE
014         END
```

Arbor mechanism

maximum speed, maximum position

```
001    A      RPT   0
002           P/V   150          6000
003           P/V   100          8000
004           P/V   100          8000
005           P/V   150          0
006    A      RPE
007           END
```

Prototype Occam program

The Occam code for the prototype arbor / transfer mechanism machine is presented below. The demonstrator uses two Propos-E motion controllers with a single Occam FSM.

```
--*****************************************************
--*
--*        Intermittent Motion / Intermittent Synchronisation
--*          Concurrent motion with interlock SINGLE FSM
--*          CRM 402H Digital I/O - T414 Transputer Based
--*         Parallel Controller  ---  C.M.Draper 4/12/90
--*
--*****************************************************

--{{{ description
-- This program provides distributed control logic for 2 Propos E motion
-- controllers. The software acts as a state machine, where states are passed
-- between concurrent FSMs. Channel communications are therefore NOT explicit
-- rendezvous, rather the FSM uses the data passed on the channels in the
-- execution of the appropriate logic.

-- a single link to the Digital IO card is used and an appropriate interface
-- is provided

--{{{ COMMENT finite state machine description
--:::A COMMENT FOLD
--{{{ finite state machine description
PLACES :
p1 : transfer at rest
p2 : transfer moving to enter arbor
p3 : transfer moving in arbor
p4 : transfer moving outside arbor to rest

p5 : arbor at rest
p6 : arbor moving

TRANSITIONS :
t1 : p1 + p5 -> p2 + p5
t2 : p1 + p6 -> p2 + p6
t3 : p2 + p5 -> p3 + p5
```

```
t4 : p2 + p6 -> p4 + p6
t5 : p3 -> p4
t6 : p4 -> p1
t7 : p4 + p5 -> p6 + p4
t8 : p6 -> p5
t9 : p2 -> p4      (watchdog timer for no channel comms - optional)

initial marking is p1 + p5
--}}}
--}}}
--}}}
--{{{  code
--{{{  COMMENT  protocols
--:::A COMMENT FOLD
--{{{
Connection to the motion generator boards has the following protocol

For the current two axis system

LINK 2
axis 1 uses bits 0,1 for output
axis 1 uses bits 0,1,2 for input

LINK 3
axis 2 uses bits 0,1 for output
axis 2 uses bits 0,1,2 for input
--}}}
--}}}
--{{{  implementation details
-- implementation detail
--{{{  libraries
#USE userio
#USE userhdr
#USE interf
--}}}

--{{{  external channel definitions and protocols
-- external protocols / channel defn, implementation details

-- motion generation board protocol -- implementatin detail


-- The CRM 402H Digital Board is connected as follows
-- axis 1 to Host transputer link 2.
-- axis 2 to Host transputer link 3.

PROTOCOL  to.digital.protocol IS BYTE :
PROTOCOL  from.digital.protocol IS BYTE :

CHAN OF to.digital.protocol    to.io.board   :
CHAN OF from.digital.protocol  from.io.board :

-- Transfer state variables      moving, moving.in.arbor, emergency
-- Arbor state variables         moving, slowing, emergency
PROTOCOL state.variable IS BOOL; BOOL; BOOL; BOOL; BOOL; BOOL :
```

```
PLACE to.io.board AT 2:
PLACE from.io.board AT 6:
--}}}

--{{{  declarations
VAL INT transfer IS 0 :
VAL INT arbor    IS 1 :

PROTOCOL status.channel IS INT :
PROTOCOL command.channel IS INT :

-- Digital Board Bit Masks Values
VAL BYTE transfer.start.msk        IS BYTE 01 : -- Bit 0
VAL BYTE transfer.clear.to.enter.msk IS BYTE 02 : -- Bit 1
VAL transfer.moving.msk            IS 01 :     -- Bit 0
VAL transfer.moving.clear.msk      IS 02 :     -- Bit 1
VAL transfer.emergency.msk         IS 64 :     -- Bit 6
VAL BYTE arbor.start.msk           IS BYTE 12 : -- Bits 2,3
VAL arbor.moving.msk               IS 04 :     -- Bits 2
VAL arbor.slowing.msk              IS 08 :     -- Bits 3
VAL arbor.emergency.msk            IS 128 :    -- Bit 7

VAL BYTE null.byte                 IS BYTE 00 :

-- NB. Digital Bit mask is same as expected returned value
-- since logic high for a particular bit pattern is same as bit mask
-- to read that bit pattern

--}}}
--}}}

--{{{  Man Machine Interface  /  task control
PROC HUI.task ( CHAN OF BOOL from.HUI,
          CHAN OF INT  from.fsm)
 --{{{  declarations
 -- HUI control codes
 VAL INT  start       IS 01 :
 VAL INT  stop        IS 02 :
 VAL INT  emergency   IS 03 :
 INT key :
 --}}}

 SEQ
  --{{{  COMMENT initial display
  --:::A COMMENT FOLD
  --{{{  initial display
  write.text.line(screen,"Transputer based synchronisation and safety interlock")
  write.text.line(screen,"logic for the arbor drum / third transfer mechanism")
  newline(screen)
  write.text.line(screen,"Motion generation by PROPOS - E controllers")
  write.text.line(screen,"2 speeds of operation for arbor to demonstrate interlock")
  newline(screen)
  newline(screen)
  write.text.line(screen,"press a key to start  / stop")
  newline(screen)
  newline(screen)
```

```
    keyboard ? key
    --}}}
    --}}}
    ALT
     keyboard ? key
      SEQ
       write.text.line ( screen, "Keyboard stop detected " )
       from.HUI ! FALSE  -- stop FSM
     from.fsm ? key
      --{{{
      SEQ
       write.text.line ( screen, "FSM Error stop detected " )
       IF
        key = 1
         write.text.line ( screen, "Transfer error " )
        key = 2
         write.text.line ( screen, "Arbor error " )
        TRUE
         SKIP
       from.HUI ! FALSE  -- stop FSM
     --}}}
   newline (screen)
   newline (screen)
   write.text.line(screen,"press a key to return to tds")
   keyboard ? key
:
--}}}
--{{{  FSM
PROC FSM ( CHAN OF BOOL from.HUI,
       CHAN OF INT to.HUI, to.io,
       CHAN OF state.variable from.io )

 --{{{  declarations
 BYTE command.byte :
 BOOL running, transfer.moving, transfer.moving.clear, transfer.emergency :
 BOOL arbor.moving, arbor.stopping, arbor.emergency :
 INT command :
 --}}}
 --{{{  code
 SEQ
  running := TRUE
  WHILE running
   SEQ
    -- await a communication from HUI, IO
    ALT
     from.HUI ? running
      to.io ! -999999
     from.io ? transfer.moving;
           transfer.moving.clear;
           transfer.emergency;
           arbor.moving;
           arbor.stopping;
           arbor.emergency
      SEQ
       SKIP
       --{{{  COMMENT was used for debugging
```

```
    --:::A COMMENT FOLD
    --{{{
    write.text.line ( to.screen, "transfer.moving = ")
    write.int (to.screen, INT transfer.moving, 4)
    write.text.line ( to.screen, "transfer.moving.clear = ")
    write.int (to.screen, INT transfer.moving.clear, 4)
    write.text.line ( to.screen, "transfer.emergency = ")
    write.int (to.screen, INT transfer.emergency, 4)
    write.text.line ( to.screen, "arbor.moving = ")
    write.int (to.screen, INT arbor.moving, 4)
    write.text.line ( to.screen, "arbor.stopping = ")
    write.int (to.screen, INT arbor.stopping, 4)
    write.text.line ( to.screen, "arbor.emergency = ")
    write.int (to.screen, INT arbor.emergency, 4)
    --}}}
    --}}}

SEQ
 IF
  transfer.emergency
   SEQ
    to.io ! -999999
    to.HUI ! 1
    running := FALSE
  arbor.emergency
   SEQ
    to.io ! -999999
    to.HUI ! 2
    running := FALSE
  TRUE AND running
   --{{{  continue
   SEQ
    command := 0
    IF
     (((NOT arbor.moving) OR (arbor.moving AND arbor.stopping))AND
     (NOT transfer.moving))
      -- enable transfer start
      command := command + (INT transfer.start.msk)
     TRUE
      -- disable transfer start
      SKIP
    IF
     (transfer.moving.clear AND (NOT arbor.moving))
      -- enable arbor start
      command := command + (INT arbor.start.msk)
     TRUE
      -- disable arbor start
      SKIP
    IF
     (((NOT arbor.moving) AND transfer.moving) AND
     (NOT transfer.moving.clear))
      -- enable transfer clear to proceed
      command := command + (INT transfer.clear.to.enter.msk )
     TRUE
      -- disable transfer clear to proceed
      SKIP
```

```
            -- transmit command byte
            to.io ! command
          --}}}

  --}}}
:
--}}}
--{{{  Digital IO Interface
PROC Digital.io ( CHAN OF INT                    axis.to.io,
                  CHAN OF state.variable         io.to.axis,
                  CHAN OF to.digital.protocol    to.io.board,
                  CHAN OF from.digital.protocol  from.io.board)

  --{{{  declarations
  BYTE io.read, command.byte :
  INT  io.state, command, old.state :
  BOOL transfer.moving, transfer.moving.in.arbor, transfer.emergency :
  BOOL arbor.moving, arbor.slowing, arbor.emergency :
  BOOL running :


  TIMER timer :
  INT timenow :
  VAL INT sample.delay IS 16 :  --    64us clock -> 1ms sample rate
  --}}}
  --{{{  code
  SEQ
    -- set up interval timer
    timer ? timenow
    --{{{  clear command byte and output it
    -- clear command byte
    command.byte := null.byte
    to.io.board ! command.byte
    -- initialise io state
    from.io.board ? io.read
    io.state := INT io.read
    --}}}
    running := TRUE
    WHILE running
      SEQ
        --{{{  read commands, timer gives sample rate

        ALT
          axis.to.io ? command
            --{{{  process command
            SEQ
              IF
                command = (-999999)
                  SEQ
                    running := FALSE
                    to.io.board ! null.byte
                TRUE
                  SEQ
                    command.byte := BYTE command
                    to.io.board ! command.byte
```

```
        --}}}

        timer ? AFTER timenow + sample.delay
          -- set up timer for sample
          timer ? timenow
      --}}}
      --{{{  read / decode io.state
      old.state := io.state
      from.io.board ? io.read
      io.state := INT io.read
      IF
        old.state = io.state
          SKIP
        old.state <> io.state     -- inputs have changed
          --{{{  process new input state
          SEQ
            -- lines are active low
            transfer.moving         := BOOL ((((~io.state)/\transfer.moving.msk)/
                                       transfer.moving.msk))
            transfer.moving.in.arbor := BOOL (((~io.state)/\transfer.moving.clear.msk)/
                                       transfer.moving.clear.msk)
            transfer.emergency       := BOOL (((io.state)/\transfer.emergency.msk)/
                                       transfer.emergency.msk)

            arbor.moving            := BOOL (((~io.state)/\arbor.moving.msk)/
                                       arbor.moving.msk)
            arbor.slowing           := BOOL (((~io.state)/\arbor.slowing.msk)/
                                       arbor.slowing.msk)
            arbor.emergency          := BOOL (((io.state)/\arbor.emergency.msk)/
                                       arbor.emergency.msk)

            -- transmit new state to axis controllers
            io.to.axis ! transfer.moving;
                       transfer.moving.in.arbor;
                       transfer.emergency;
                       arbor.moving;
                       arbor.slowing;
                       arbor.emergency
          --}}}
      --}}}

  --}}}
:
--}}}

SEQ
  --{{{  channel declarations
  CHAN OF INT to.mmi :
  CHAN OF BOOL from.mmi, HUI.to.io :
  CHAN OF INT axis.to.io :
  CHAN OF state.variable io.to.axis :
  CHAN OF BYTE to.board, from.board:
  CHAN OF ANY stopper :
  --}}}
  -- main program
```

```
  PAR
    HUI.task   ( from.mmi,
                 to.mmi)
    FSM          ( from.mmi,
                 to.mmi,
                 axis.to.io,
                 io.to.axis)
    Digital.io ( axis.to.io,
                 io.to.axis,
                 to.io.board,
                 from.io.board)

--}}}
```

8.2 Distributed 2 axis flexible motion/synchronisation controller for the IDM demonstration machine

The following Occam code is the source for the two axis demonstrator providing full flexibility of IDMs. There are four concurrent processes, 'arbor.drum', 'third.transfer', 'HUI.process' and 'dac'. These follow the designs detailed in the main body of the thesis.

'arbor drum' and 'third.transfer' processes use a common structure of an 'IDM motion-trajectory / synchronisation' process. The different functionality for each IDM motion/synchronisation is provided by unique motion segment definitions and FSMs for each process.

Each 'IDM motion-trajectory / synchronisation' process itself consists of a number of concurrent processes. These provide the 'motion-generation / loop-control' FSM synchronisation and motion segment scheduling process and the 'motion demand' real-time motion control process.

The 'motion demand' process comprises the 'motion seg' process which calculates required position/velocity, the 'event timer' real-time event sequencing process and the 'velocity demand gen' real-time velocity demand generator. Plant feedback is used to detect the completion of the current motion segment in order to start the next segment. Position information is used for this task and timing of plant operation is not used, except to detect controller faults.

Occam source code

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%    TWO AXIS FLEXIBLE MOTION/SYNCHRONISATION CONTROLLER
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

--{{{  COMMENT description
--:::A COMMENT FOLD
--{{{  description
The motion generation with flexibility for two axis combines individual axis
motion generation with the synchronisation between axes.
As a result there are effectively 2 FSMs per axis, 1 for synch, 1 for motion.
Motion FSM is programmable, synch FSM is sequence fixed with some flexibility.
Note that states between synch and motion FSMs may correspond and accordingly a
state variable may be employed (eg. 'running')

--}}}
--{{{  finite state machine description
PLACES :
p1 : transfer at rest
p2 : transfer moving to enter arbor
p3 : transfer moving in arbor
p4 : transfer moving outside arbor to rest

p5 : arbor at rest
p6 : arbor moving

TRANSITIONS :
t1 : p1 + p5 -> p2 + p6
t2 : p1 + p6 -> p2 + p6
t3 : p2 + p5 -> p3
t4 : p2 + p6 -> p4
t5 : p3 -> p4
t6 : p4 -> p1
t7 : p4 + p5 -> p6
t8 : p6 -> p5
t9 : p2 -> p4     (watchdog timer for no channel comms - optional)

initial marking is p1 + p5
--}}}
--}}}

--{{{  libraries / definitions
#USE userhdr
#USE userio
#USE interf


--{{{  definitions
-- external protocols / channel defn, implementation details

-- The ADC is connected to Host transputer link 3.
-- The DAC is          ...              link 2.

PROTOCOL dac.protocol IS BYTE; BYTE; BYTE:
```

```
CHAN OF dac.protocol to.dac :
PLACE to.dac AT 2:

VAL INT arbor IS 1 :
VAL INT transfer IS 0 :
VAL INT arbor.max.vel IS 5000       :
VAL INT transfer.max.vel IS 50000    :

VAL INT single.shot IS 1 :
VAL INT start      IS 2 :       -- for use in HUI - motion demand loop
VAL INT stop       IS 3 :       -- control and reports from demand
VAL INT shutdown   IS 4 :
VAL INT flex.read  IS 5 :       -- for use in reports to HUI from demand
VAL INT bad.read   IS 6 :
VAL INT emergency  IS 7 :

--}}}


--}}}

--{{{  arbor drum
--
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%     Complex motion arbor drum
--
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PROC arbor.drum ( CHAN OF BOOL to.event,
              CHAN OF INT  from.HUI.flex,
                        from.HUI.command,
              CHAN OF ANY  to.HUI,
              CHAN OF BOOL from.arbor.at.rest,
                        from.arbor.moving,
                        from.transfer.clear.of.arbor,
              CHAN OF INT  to.servo)
 --{{{  libraries
 #USE userio
 #USE userhdr
 VAL INT terminate IS -999999 :

 --{{{  COMMENT not used
 --:::A COMMENT FOLD
 --{{{
 -- reduces speed of sampling to suit the graphics display
 -- as time.scale increases, sample rate decreases
 VAL INT time.scale IS 1000 :
 --}}}
 --}}}
 --}}}
 --{{{  configuration
 -- external protocols / channel defn, implementation details

 -- The DAC is      ...          link 2.
```

```
PROTOCOL dac.protocol IS BYTE; BYTE; BYTE:

CHAN OF dac.protocol to.dac :

PLACE to.dac AT 2:
--}}}
--{{{  declarations
VAL INT no.segs IS 4          : -- Number of segments
[no.segs][8]INT Motion.segment   : -- Motion segment, Id no,
                        -- incremental position,
                        -- and absolute terminal velocity.
VAL INT ms.pos.max  IS 0       : -- maximum position for seg
VAL INT ms.pos.min  IS 1       : -- minimum position for seg
VAL INT ms.vel.max  IS 2       : -- maximum velocity for seg
VAL INT ms.vel.min  IS 3       : -- maximum velocity for seg
VAL INT ms.flex.pos IS 4       : -- flexibility parameter for pos
VAL INT ms.flex.vel IS 5       : -- flexibility parameter for vel
                        -- flex param [0] is always 0
VAL INT ms.next.seg.ok  IS 6    : -- segment sequence
VAL INT ms.next.seg.err IS 7    : --

                        -- 0 param is not used
VAL INT no.f.params IS 3       : -- number of flexibility parameters
[no.f.params]INT flex.param    : -- table of flexibility parameters
INT result, rate  :

--}}}
--{{{  motion generation / loop control     (includes synch)
PROC Motion.demand ( CHAN OF INT from.HUI.flex, from.HUI.command, to.HUI,
            to.accel,
            CHAN OF BOOL from.velocity)

 -- looks after motion demands (including flexibility)
 -- inputs for start/stop and flexibility from HUI
 -- inputs for motion sequence from velocity demand

 -- state variable for motion is 'running' -used by commands

 --{{{  declarations
 INT  seg, value, param.id, command, position, velocity :
 BOOL not.finished, vel.ok, cycle.stop :

 INT state  :  -- current place of state machine
 VAL INT arbor.at.rest       IS 0 :     -- possible places
 VAL INT arbor.moving.1      IS 1 :
 VAL INT arbor.moving.2      IS 2 :
 VAL INT arbor.moving.3      IS 3 :
 VAL INT arbor.moving.to.rest  IS 4 :
 BOOL hard.emergency.stop :
 BOOL running :
 BOOL cycle.stop :
 BOOL prompt :

 TIMER arbor.timer :
 INT timenow :
 VAL INT emergency.delay IS 2000 :
```

```
--{{{  Motion seg procedure -calculates parameters, transmit and await acknowledge
PROC motion.seg ( INT seg )
  SEQ        -- cycle 'seg' through defined sequence
    -- calculation is    min+(flex*(max-min)/9)
    position := Motion.segment[seg][ms.pos.min] +
                (((Motion.segment[seg][ms.pos.max] -
                Motion.segment[seg][ms.pos.min])/9) *
                flex.param[Motion.segment[seg][ms.flex.pos]])

    velocity := Motion.segment[seg][ms.vel.min] +
                (((Motion.segment[seg][ms.vel.max] -
                Motion.segment[seg][ms.vel.min])/9) *
                flex.param[Motion.segment[seg][ms.flex.vel]])
    to.accel ! position
    to.accel ! velocity

    -- insert a timer here to check for vel loop problems ?

    from.velocity ? vel.ok
    --{{{  COMMENT succesful motion / synch checks
    --:::A COMMENT FOLD
    --{{{  succesful motion / synch checks
    -- check for succesful operation of velocity demand generator
    -- and synchronisation fsm.
    IF
      (vel.ok = FALSE) OR ((abort = TRUE) AND (seg = 0))
        SEQ
          seg := Motion.segment[seg][ms.next.seg.err]
      vel.ok = TRUE
        SEQ
          seg := Motion.segment[seg][ms.next.seg.ok]
      TRUE
        SKIP
    --}}}
    --}}}

:
--}}}
--}}}

SEQ
  --{{{  initialisation
  running := FALSE
  cycle.stop := FALSE
  not.finished := TRUE
  state := arbor.at.rest
  hard.emergency.stop := FALSE
  --}}}

  WHILE not.finished
    SEQ
      --{{{  monitor for new commands / schedule motion dispatcher
      ALT
        -- get a loop control command
        from.HUI.command ? command
```

```
  --{{{  process command
  SEQ
    IF
      command = single.shot
        --{{{
        SEQ
          --to.HUI ! single.shot
          running := TRUE            -- set state variable
          cycle.stop := TRUE
          state := arbor.at.rest
        --}}}
      command = start
        --{{{
        SEQ
          to.HUI ! start
          -- use motion.state to start up
          running := TRUE
          state := arbor.at.rest
          cycle.stop := FALSE
        --}}}
      command = stop
        --{{{
        SEQ
          --    to.HUI ! stop             -- report stop on completion
          IF
            running = TRUE
              cycle.stop := TRUE    -- state variable 'running' is affected indirectly
            running = FALSE         -- stopped already
              SKIP

        --}}}
      command = shutdown
        --{{{
        SEQ
          -- close down motion demand (does not take into account 'running')
          to.HUI ! shutdown
          not.finished := FALSE
        --}}}
      command = emergency
        --{{{
        SEQ
          to.HUI ! emergency
          IF
            running = TRUE
              hard.emergency.stop := TRUE
            running = FALSE
              -- already stopped
              SKIP
        --}}}
      TRUE
        SEQ
          -- unknown command
          to.HUI ! bad.read
  --}}}
-- get a flexibility parameter and new value
from.HUI.flex ? param.id
```

```
  --{{{  process flexibility
  SEQ
    from.HUI.flex ? value
    -- load flex param ready for use
    flex.param[param.id] := value
    to.HUI ! flex.read
  --}}}
-- no commands, carry on with motion control (note loop control variable)
running & SKIP
  SEQ
    --{{{  motion dispatcher with inbuilt synchronisation
    IF
      hard.emergency.stop = TRUE
        --{{{  alternative processing for emergency stop
        SEQ
          IF
            -- remember state is 'NEXT STATE' - arbor.moving.1 is in fact stationary
            state = arbor.at.rest
              --{{{  enable emergency transitions for arbor.at.rest
              SEQ
                -- emergency code  - finish any input/output communications
                arbor.timer ? timenow
                PAR
                  from.arbor.moving ! TRUE
                  from.arbor.at.rest ! TRUE
                  ALT
                    from.transfer.clear.of.arbor ? prompt
                      SKIP
                    arbor.timer ? AFTER (timenow + emergency.delay)
                      SKIP
            --}}}
            state = arbor.moving.1
              --{{{  enable emergency transitions for arbor.moving.1
              SEQ
                -- emergency code  - finish any input/output communications
                arbor.timer ? timenow
                PAR
                  from.arbor.moving ! TRUE
                  from.arbor.at.rest ! TRUE
                  ALT
                    from.transfer.clear.of.arbor ? prompt
                      SKIP
                    arbor.timer ? AFTER (timenow + emergency.delay)
                      SKIP

            --}}}
            state = arbor.moving.2
              --{{{  enable emergency transitions for arbor.moving.2
              SEQ
                -- emergency code  - finish any input/output communications
                arbor.timer ? timenow
                PAR
                  from.arbor.at.rest ! TRUE
                  from.arbor.moving ! TRUE
                  ALT
                    from.transfer.clear.of.arbor ? prompt
```

```
            SKIP
          arbor.timer ? AFTER (timenow + emergency.delay)
            SKIP
        -- emergency motion
        SEQ
          seg := 3
          motion.seg ( seg )
    --}}}
  state = arbor.moving.3
    --{{{  enable emergency transitions for arbor.moving.3
    SEQ
      -- emergency code  - finish any input/output communications
      arbor.timer ? timenow
      PAR
        from.arbor.at.rest ! TRUE
        from.arbor.moving ! TRUE
        ALT
          from.transfer.clear.of.arbor ? prompt
            SKIP
          arbor.timer ? AFTER (timenow + emergency.delay)
            SKIP
        -- emergency motion
        SEQ
          seg := 3
          motion.seg ( seg )

    --}}}
  state = arbor.moving.to.rest
    --{{{  enable emergency transitions for arbor.moving.to.rest
    SEQ
      -- emergency code  - finish any input/output communications
      arbor.timer ? timenow
      PAR
        from.arbor.at.rest ! TRUE
        from.arbor.moving ! TRUE
        ALT
          from.transfer.clear.of.arbor ? prompt
            SKIP
          arbor.timer ? AFTER (timenow + emergency.delay)
            SKIP
        -- emergency motion
        SEQ
          seg := 3
          motion.seg ( seg )
    --}}}
  TRUE
    --{{{  problems  - invalid state
    SEQ
      STOP
    --}}}

  state := arbor.at.rest
  running := FALSE
  hard.emergency.stop := FALSE
--}}}
```

```
state = arbor.at.rest
 --{{{  enable transitions for arbor.at.rest
 SEQ
  PAR
   SEQ
    from.arbor.at.rest ! TRUE
   SEQ
    from.transfer.clear.of.arbor ? prompt
    state := arbor.moving.1

 --}}}
state = arbor.moving.1
 --{{{  enable transitions for arbor.moving.1
 SEQ
  seg := 0
  motion.seg ( seg )
  state := arbor.moving.2

 --}}}
state = arbor.moving.2
 --{{{  enable transitions for arbor.moving.2
 SEQ
  seg := 1
  motion.seg ( seg )
  state := arbor.moving.3

 --}}}
state = arbor.moving.3
 --{{{  enable transitions for arbor.moving.3
 SEQ
  seg := 2
  motion.seg ( seg )
  state := arbor.moving.to.rest

 --}}}
state = arbor.moving.to.rest
 --{{{  enable transitions for arbor.moving.to.rest
 PAR
  SEQ
   from.arbor.moving ! TRUE      -- enable start for transfer
  SEQ
   seg := 3
   motion.seg ( seg )
   state := arbor.at.rest
   --{{{  cycle commands
   -- deal with cycle stop here
   IF
    cycle.stop = TRUE
     SEQ
      running := FALSE
      to.HUI ! command    -- report on completion (single shot/stop)
    TRUE
     SKIP
  --}}}
 --}}}
TRUE
```

```
            --{{{  problems - invalid state
            SEQ
              STOP
            --}}}
          --}}}
      --}}}


:
--}}}
--{{{  Timing event activity
PROC Event.timer (CHAN OF INT velocity.demand.clock,
                  CHAN OF BOOL HUI, INT rate)
  --{{{  description
  -- This process provides event sequencing to all of the  real-time activities
  -- of the servo system
  -- These include slow rate scanning of velocity demand ramps
  -- Faster scanning of servo/plant loops
  -- Faster scanning for plant state decode

  -- actual parameters, velocity demand generation sample rate 1s (1000ms)
  --                servo/plant loop rate              0.1s (100ms)
  --                plant state decode rate            0.1s (100ms)


  --}}}
  --{{{  code
  --{{{  Declarations

  VAL INT mtd IS 64           : -- 1 tick every 64us machine.tick.divisor
  --VAL INT vel.interval  IS 96  : -- vel gen sample rate 100000/mtd
  --VAL INT velocity.rate  IS 4   : -- sample rate in ms from above
  INT vel.interval, velocity.rate :
  VAL INT plant.interval IS 1562 :  -- plant sample rate 100000/mtd
  VAL INT plant.rate    IS 100  :  -- sample rate in ms from above
  VAL INT servo.interval IS 1562 :  -- vel gen sample rate 100000/mtd
  VAL INT servo.rate    IS 100  :  -- sample rate in ms from above
  INT plant.elapsed.time       : -- next event
  INT vel.elapsed.time         : -- next event
  INT servo.elapsed.time       : -- next event
  INT time                  : -- local scratchpad variable
  BOOL commence               : -- start / stop from HUI
  BOOL timer.enabled          : -- activity variable

  TIMER plant.decode.timer, velocity.demand.timer, servo.timer :



--}}}

SEQ
  --{{{  intialisation
  -- await initialisation
  HUI ? commence
  IF
```

```
    commence
      timer.enabled := TRUE
    NOT (commence)
      timer.enabled := FALSE

  --{{{  COMMENT  other activities
  --:::A COMMENT FOLD
  --{{{   other activities
  -- initialise plant state decode timer
  plant.decode.timer ? time
  -- set up next event for plant state
  plant.elapsed.time := time + plant.interval
  servo.timer ? time
  servo.elapsed.time := time + servo.interval
  -- set up next event for servo demand
  --}}}
  --}}}
  vel.interval := rate*16
  velocity.rate := rate

  -- initialise velocity demand generator timer
  velocity.demand.timer ? time
  -- set up next event for velocity demand
  vel.elapsed.time := time + vel.interval
  --}}}
  WHILE timer.enabled
    --{{{  monitor for stop and events
    -- monitor for stop and events
    ALT
      HUI ? commence
        --{{{
        SEQ
          IF
            commence
              timer.enabled := TRUE
            NOT (commence)
              SEQ
                --{{{  COMMENT  useless
                --:::A COMMENT FOLD
                --{{{
                -- shutdown servo
                servo.clock ! terminate
                -- shutdown plant state decode - start pipeline shutdown
                plant.decode.clock ! terminate
                -- shutdown velocity demand gen
                --}}}
                --}}}
                velocity.demand.clock ! terminate

                timer.enabled := FALSE
      --}}}
      velocity.demand.timer ? AFTER vel.elapsed.time
        --{{{
        SEQ
          -- set up next event for velocity demand
          velocity.demand.timer ? time
```

```
          vel.elapsed.time := time + vel.interval
          -- send event / time to vel demand gen
          velocity.demand.clock ! velocity.rate

        --}}}
       --{{{  COMMENT other activities
       --:::A COMMENT FOLD
       --{{{  other activities
       plant.decode.timer  ? AFTER plant.elapsed.time
        --{{{
        SEQ
          -- set up next event for plant state
          plant.decode.timer ? time
          plant.elapsed.time := time + plant.interval
          -- send event / time to plant decode
          plant.decode.clock ! plant.rate
        --}}}
       servo.timer ? AFTER servo.elapsed.time
        --{{{
        SEQ
          -- set up next event for servo sample
          servo.timer ? time
          servo.elapsed.time := time + servo.interval
          -- send event / time to servo
          servo.clock ! servo.rate
        --}}}
      --}}}
      --}}}
     --}}}

  --}}}
:
--}}}
--{{{  Velocity demand gen activity
PROC Velocity.demand.gen (CHAN OF INT velocity.demand.clock, to.servo,
                          to.accel.calc,
                    CHAN OF BOOL demand.satisfied )
  --{{{  description
  -- read in an accleration rate from the motion trajectory generator
  -- apply the rate to the velocity demand for the servo
  -- transmit new velocity demands at appropriate intervals

  --}}}
  --{{{  code

  --{{{  declarations
  INT accel.conversion       :     -- convert accel rate to velocity
                            -- generator sample rate
  BOOL velocity.demand.enabled :     -- activity variable
  INT acceleration.rate      :     -- rate parameter to vel gen
  INT new.demand             :     -- velocity demand to servo
  INT vel.sample.rate        :     -- sample rate from event timer
  INT position.increment     :     -- used in acceleration calc
  INT terminal.velocity      :     --  "  "      "       "
  INT current.velocity       :     --  "  "      "       "
  INT divisor                :     --  "  "      "       "
```

```
INT mantissa            :    --  "  "      "      "
BOOL unfinished.flag    :    -- an acknowledgement of motion competion
                             -- is required from vel to HUI/motion disp
--}}}

SEQ
 --{{{  await initialisation
 -- await initialisation
 velocity.demand.clock ? vel.sample.rate
 new.demand  := 0
 acceleration.rate := 0
 -- calculate update rate
 IF
  vel.sample.rate > 0
   -- to convert ec/s/s to ec/sample.rate.squared
   -- nominal sample rate is 1 s
   -- 1000.0 divisor converts from ms to s
   -- note accel.conversion is a divisor (for int arithmetic)
   accel.conversion := (1000/vel.sample.rate)
  TRUE
   accel.conversion := 0
 velocity.demand.enabled := TRUE
--}}}
WHILE velocity.demand.enabled
 SEQ
  -- monitor for new rate demand/shutdown or clock start
  ALT
   velocity.demand.clock ? vel.sample.rate
    --{{{  calc / output new velocity demand or shutdown
    SEQ
     IF
      vel.sample.rate = terminate
       SEQ
        velocity.demand.enabled := FALSE
      TRUE
       SEQ
        -- calc new vel demand
        new.demand := new.demand + acceleration.rate
        --s=(v2-u2)/2a
        --{{{  COMMENT calculate new position  -- NOT used yet
        --:::A COMMENT FOLD
        --{{{  calculate new position  -- NOT used yet
        old.demand := new.demand       -- insert before demand changes
        new.position := new.position +
                    (((new.demand*new.demand)-(old.demand*old.demand))/
                    (2*acceleration.rate))
        --}}}
        --}}}
        --{{{  limit checks
        -- limiting checks here if you please
        IF
         new.demand > arbor.max.vel
          new.demand := arbor.max.vel
         new.demand < (-arbor.max.vel)
          new.demand := -arbor.max.vel
         (new.demand <= terminal.velocity) AND
```

```
        ((acceleration.rate < 0) AND unfinished.flag)
          --{{{
          SEQ
            new.demand := terminal.velocity  -- remove rounding
            demand.satisfied ! TRUE
            acceleration.rate := 0
            unfinished.flag := FALSE
          --}}}
        (new.demand >= terminal.velocity) AND
        ((acceleration.rate > 0) AND unfinished.flag)
          --{{{
          SEQ
            new.demand := terminal.velocity  -- remove rounding
            demand.satisfied ! TRUE
            acceleration.rate := 0
            unfinished.flag := FALSE
          --}}}
        TRUE
          SKIP
      -- output new vel demand
      --}}}
      to.servo ! new.demand   -- new demand is current velocity
--}}}

to.accel.calc ? position.increment
  SEQ
    to.accel.calc ? terminal.velocity
    unfinished.flag := TRUE
    --{{{  perform accel calc
    -- calculate acceleration rate
    -- using V2=U2+2*A*S
    divisor := 2*position.increment

    --{{{  GET CURRENT.VELOCITY FROM SOMEWHERE
    current.velocity := new.demand          -- for open loop only

    --}}}

    -- problem with negative terminal velocities / positive initial velocities
    -- and vice versa, if this occurs flag error
    -- check gradient of acceleration for substitution in final parameter
    IF
      (terminal.velocity > 0) AND (current.velocity < 0)
        STOP
      (terminal.velocity < 0) AND (current.velocity > 0)
        STOP
      TRUE
        SKIP

    mantissa:=((terminal.velocity*terminal.velocity) -
          (current.velocity*current.velocity))
    acceleration.rate := ( mantissa / divisor )

    -- result using ec/s and ec for v,s is in ec/s/s

    --{{{  new rate    converted for dac
```

```
          SEQ
            -- convert acceleration to fit sample rate
            -- convert from ec/s/s to ec/sample rate squared
            acceleration.rate := acceleration.rate/accel.conversion
          --}}}
          --{{{  zero acceleration detection
          IF
            (acceleration.rate = 0)
              STOP          -- won't work see below
            TRUE
              SKIP
          -- acceleration of 0 causes problems since the demand complete test cannot be
          -- carried out succesfully since final vel=initial vel
          -- we need some form of position detection (particularly since position is
          -- extremely important)
          --}}}
          --}}}
  --}}}
:
--}}}
SEQ
  --{{{  initialisation
  --{{{  set up sample speed
  -- set up sample speed
  --1< sample speed <10
  rate := 7               -- preset sample rate
  --}}}

  -- set up motion segment array
  --{{{  segment 0
  --  to max +ive velocity
  Motion.segment[0][ms.pos.max]     := 150   -- max incremental position (ec)
  Motion.segment[0][ms.pos.min]     := 30    -- min incremental position (ec)
  Motion.segment[0][ms.vel.max]     := 3000  -- max terminal velocity (ec/s)
  Motion.segment[0][ms.vel.min]     := 300   -- min terminal velocity (ec/s)
  Motion.segment[0][ms.flex.pos]    := 2     -- position flexibility variable
  Motion.segment[0][ms.flex.vel]    := 1     -- velocity flexibility variable
  Motion.segment[0][ms.next.seg.ok] := 1     -- seq sequence   - next if ok
  Motion.segment[0][ms.next.seg.err] := 6    --                next if error

              -- NB flex variable of 0 = indicates no flexibility
              --                    min parameter will be executed
  --}}}
  --{{{  segment 1
  --  slow to enter arbor
  Motion.segment[1][ms.pos.max]     := 100   -- max incremental position (ec)
  Motion.segment[1][ms.pos.min]     := 220   -- min incremental position (ec)
  -- note position increases with decreasing flex variable
  Motion.segment[1][ms.vel.max]     := 4000  -- max terminal velocity (ec/s)
  Motion.segment[1][ms.vel.min]     := 400   -- min terminal velocity (ec/s)
  Motion.segment[1][ms.flex.pos]    := 2     -- position flexibility variable
  Motion.segment[1][ms.flex.vel]    := 1     -- velocity flexibility variable
  Motion.segment[1][ms.next.seg.ok] := 2     -- seq sequence   - next if ok
  Motion.segment[1][ms.next.seg.err] := 2    --                next if error

              -- NB flex variable of 0 = indicates no flexibility
```

```
            --                  min parameter will be executed
--}}}
--{{{  segment 2
--  slow to enter arbor
Motion.segment[2][ms.pos.max]     := 100   -- max incremental position (ec)
Motion.segment[2][ms.pos.min]     := 220   -- min incremental position (ec)
Motion.segment[2][ms.vel.max]     := 3000    -- max terminal velocity (ec/s)
Motion.segment[2][ms.vel.min]     := 300     -- min terminal velocity (ec/s)
Motion.segment[2][ms.flex.pos]    := 2     -- position flexibility variable
Motion.segment[2][ms.flex.vel]    := 1     -- velocity flexibility variable
Motion.segment[2][ms.next.seg.ok] := 3     -- seq sequence   - next if ok
Motion.segment[2][ms.next.seg.err] := 3    --                next if error

            -- NB flex variable of 0 = indicates no flexibility
            --                  min parameter will be executed
--}}}
--{{{  segment 3
--  slow to enter arbor
Motion.segment[3][ms.pos.max]     := 150   -- max incremental position (ec)
Motion.segment[3][ms.pos.min]     := 30    -- min incremental position (ec)
Motion.segment[3][ms.vel.max]     := 0     -- max terminal velocity (ec/s)
Motion.segment[3][ms.vel.min]     := 0     -- min terminal velocity (ec/s)
Motion.segment[3][ms.flex.pos]    := 2     -- position flexibility variable
Motion.segment[3][ms.flex.vel]    := 0     -- velocity flexibility variable
Motion.segment[3][ms.next.seg.ok] := 0     -- seq sequence   - next if ok
Motion.segment[3][ms.next.seg.err] := 0    --                next if error

            -- NB flex variable of 0 = indicates no flexibility
            --                  min parameter will be executed
--}}}

-- set up flexibility parameters default values
--{{{  init flex.params
flex.param[0] := 0      -- not used
flex.param[1] := 0      -- valid range 0-9     speed
flex.param[2] := 0      -- valid range 0-9     synch position with transfer
                                -- transfer start

--}}}

--}}}
--{{{  main program
CHAN OF BOOL HUI, demand.satisfied :
CHAN OF INT   to.accel.calc, velocity.demand.clock :
PAR
 Velocity.demand.gen ( velocity.demand.clock,
                to.servo,
                to.accel.calc,
                demand.satisfied )
 Event.timer       ( velocity.demand.clock,
                to.event, rate )
 Motion.demand     ( from.HUI.flex,
                from.HUI.command,
                to.HUI,
                to.accel.calc,
                demand.satisfied )
```

```
  --}}}
:
--}}}
--{{{  third transfer
--
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
--%                    Complex motion transfer
--
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
PROC third.transfer ( CHAN OF BOOL to.event,
                      CHAN OF INT  from.HUI.flex,
                                   from.HUI.command,
                      CHAN OF ANY  to.HUI,
                      CHAN OF BOOL from.transfer.moving.clear.to.rest,
                                   from.arbor.at.rest,
                                   from.arbor.moving,
                      CHAN OF INT  to.servo)
  --{{{  libraries
  #USE userio
  #USE userhdr
  VAL INT terminate IS -999999 :

  --{{{  COMMENT not used
  --:::A COMMENT FOLD
  --{{{
  -- reduces speed of sampling to suit the graphics display
  -- as time.scale increases, sample rate decreases
  VAL INT time.scale IS 1000 :
  --}}}
  --}}}
  --}}}
  --{{{  configuration
  -- external protocols / channel defn, implementation details

  -- The DAC is        ...              link 2.

  PROTOCOL dac.protocol IS BYTE; BYTE; BYTE:

  CHAN OF dac.protocol to.dac :

  PLACE to.dac AT 2:
  --}}}
  --{{{  declarations
  VAL INT no.segs IS 9            : -- Number of segments
  [no.segs][8]INT Motion.segment   : -- Motion segment, Id no,
                                     -- incremental position,
                                     -- and absolute terminal velocity.
  VAL INT ms.pos.max  IS 0        : -- maximum position for seg
  VAL INT ms.pos.min  IS 1        : -- minimum position for seg
  VAL INT ms.vel.max  IS 2        : -- maximum velocity for seg
  VAL INT ms.vel.min  IS 3        : -- maximum velocity for seg
  VAL INT ms.flex.pos IS 4        : -- flexibility parameter for pos
  VAL INT ms.flex.vel IS 5        : -- flexibility parameter for vel
                                     -- flex param [0] is always 0
```

```
VAL INT ms.next.seg.ok  IS 6     : -- segment sequence
VAL INT ms.next.seg.err IS 7     : --

                        -- 0 param is not used
VAL INT no.f.params IS 4         : -- number of flexibility parameters
[no.f.params]INT flex.param      : -- table of flexibility parameters
INT result, rate  :

--}}}
--{{{  motion generation / loop control
PROC Motion.demand ( CHAN OF INT from.HUI.flex, from.HUI.command, to.HUI,
                     to.accel,
                     CHAN OF BOOL from.velocity )

  -- looks after motion demands (including flexibility)
  -- inputs for start/stop and flexibility from HUI
  -- inputs for motion sequence from velocity demand

  -- state variable for motion is 'running' -used by commands

  --{{{  declarations
  INT  seg, value, param.id, command, position, velocity :
  BOOL not.finished, vel.ok, cycle.stop :

  INT state  :  -- current place of state machine
  VAL INT transfer.at.rest                  IS 0 : -- possible places
  VAL INT transfer.moving.to.enter.arbor    IS 1 :
  VAL INT transfer.moving.in.arbor.1        IS 2 :
  VAL INT transfer.moving.in.arbor.2        IS 3 :
  VAL INT transfer.moving.in.arbor.3        IS 4 :
  VAL INT transfer.moving.in.arbor.4        IS 5 :
  VAL INT transfer.moving.in.arbor.5        IS 6 :
  VAL INT transfer.moving.outside.arbor.to.rest IS 7 :
  VAL INT transfer.moving.emergency.to.rest  IS 8 :
  BOOL running :
  BOOL prompt :
  BOOL hard.emergency.stop :
  BOOL initialise :

  TIMER transfer.timer :
  INT timenow :
  VAL INT watchdog.time IS 2000 :
  VAL INT emergency.delay IS 2000 :
  INT any :

  --{{{  Motion seg procedure -calculates parameters, transmit and await acknowledge
  PROC motion.seg ( INT seg )
    SEQ        -- cycle 'seg' through defined sequence
      -- calculation is   min+(flex*(max-min)/9)
      position := Motion.segment[seg][ms.pos.min] +
                  (((Motion.segment[seg][ms.pos.max] -
                  Motion.segment[seg][ms.pos.min])/9) *
                  flex.param[Motion.segment[seg][ms.flex.pos]])

      velocity := Motion.segment[seg][ms.vel.min] +
                  (((Motion.segment[seg][ms.vel.max] -
```

```
            Motion.segment[seg][ms.vel.min])/9) *
            flex.param[Motion.segment[seg][ms.flex.vel]])
    to.accel ! position
    to.accel ! velocity

    -- insert a timer here to check for vel loop problems ?

    from.velocity ? vel.ok
    --{{{  COMMENT succesful motion / synch checks
    --:::A COMMENT FOLD
    --{{{  succesful motion / synch checks
    -- check for succesful operation of velocity demand generator
    -- and synchronisation fsm.
    IF
      (vel.ok = FALSE) OR ((abort = TRUE) AND (seg = 0))
        SEQ
          seg := Motion.segment[seg][ms.next.seg.err]
      vel.ok = TRUE
        SEQ
          seg := Motion.segment[seg][ms.next.seg.ok]
      TRUE
        SKIP
    --}}}
    --}}}

:
--}}}
--}}}

SEQ
  --{{{  initialisation
  running := FALSE
  cycle.stop := FALSE
  not.finished := TRUE
  state := transfer.at.rest
  initialise := TRUE
  hard.emergency.stop := FALSE
  --}}}

  WHILE not.finished
    SEQ
      --{{{  monitor for new commands  / schedule motion
      ALT
        -- get a loop control command
        from.HUI.command ? command
          --{{{  process command
          SEQ
            IF
              command = single.shot
                --{{{
                SEQ
                  --to.HUI ! single.shot
                  running := TRUE            -- set state variable
                  cycle.stop := TRUE
                  state := transfer.at.rest
                  initialise := TRUE
```

```
      --}}}
    command = start
      --{{{
      SEQ
        to.HUI ! start
        cycle.stop := FALSE
        running := TRUE
        state := transfer.at.rest
        initialise := TRUE
      --}}}
    command = stop
      --{{{
      SEQ
        --to.HUI ! stop              -- report stop on completion
        IF
          running = TRUE
            cycle.stop := TRUE      -- state variable 'running' is affected indirectly
          running = FALSE           -- no problems stopped already
            SKIP
      --}}}
    command = shutdown
      --{{{
      SEQ
        -- close down motion demand (does not take into account 'running')
        to.HUI ! shutdown
        not.finished := FALSE
        -- will leave ungracefully
      --}}}
    command = emergency
      --{{{
      SEQ
        to.HUI ! emergency
        IF
          running
            hard.emergency.stop := TRUE
          NOT running
            SKIP -- already stopped
      --}}}
    TRUE
      SEQ
        -- unknown command
        to.HUI ! bad.read
  --}}}
-- get a flexibility parameter and new value
from.HUI.flex ? param.id
  --{{{  process flexibility
  SEQ
    from.HUI.flex ? value
    -- load flex param ready for use
    flex.param[param.id] := value
    to.HUI ! flex.read
  --}}}
-- no commands, carry on with motion control (note loop control variable)
running & SKIP
  SEQ
    --{{{  motion dispatcher with inbuilt synchronisation
```

```
IF
 hard.emergency.stop = TRUE
  --{{{  deal with emergency
  SEQ
   IF
    ((state = transfer.at.rest) OR (state = transfer.moving.to.enter.arbor))
     --{{{  enable appropriate emergency transitions
     -- finish communications
     PAR
      from.transfer.moving.clear.to.rest ! TRUE      -- outputs
      --{{{  inputs
      SEQ
       transfer.timer ? timenow
       ALT
        from.arbor.at.rest ? prompt
         SKIP
        transfer.timer ? AFTER timenow + emergency.delay
         SKIP
       ALT
        from.arbor.moving ? prompt
         SKIP
        transfer.timer ? AFTER timenow + emergency.delay
         SKIP
     --}}}
    --}}}
    state = transfer.moving.in.arbor.1
     --{{{  enable appropriate emergency transitions
     -- finish communications / motion to stop
     PAR
      from.transfer.moving.clear.to.rest ! TRUE      -- outputs
      --{{{  inputs
      SEQ
       transfer.timer ? timenow
       ALT
        from.arbor.at.rest ? prompt
         SKIP
        transfer.timer ? AFTER timenow + emergency.delay
         SKIP
       ALT
        from.arbor.moving ? prompt
         SKIP
        transfer.timer ? AFTER timenow + emergency.delay
         SKIP
     --}}}
     SEQ
      seg := 6
      motion.seg ( seg )
    --}}}
    state = transfer.moving.in.arbor.2
     --{{{  enable appropriate emergency transitions
     -- finish communications / motion to stop
     PAR
      from.transfer.moving.clear.to.rest ! TRUE      -- outputs
      --{{{  inputs
      SEQ
       transfer.timer ? timenow
```

```
      ALT
       from.arbor.at.rest ? prompt
         SKIP
       transfer.timer ? AFTER timenow + emergency.delay
         SKIP
      ALT
       from.arbor.moving ? prompt
         SKIP
       transfer.timer ? AFTER timenow + emergency.delay
         SKIP
    --}}}
    SEQ
     seg := 6
     motion.seg ( seg )
  --}}}
state = transfer.moving.in.arbor.3
  --{{{  enable appropriate emergency transitions
  -- finish communications / motion to stop
  PAR
    from.transfer.moving.clear.to.rest ! TRUE      -- outputs
    --{{{  inputs
    SEQ
     transfer.timer ? timenow
     ALT
      from.arbor.at.rest ? prompt
        SKIP
      transfer.timer ? AFTER timenow + emergency.delay
        SKIP
     ALT
      from.arbor.moving ? prompt
        SKIP
      transfer.timer ? AFTER timenow + emergency.delay
        SKIP
    --}}}
    SEQ
     seg := 6
     motion.seg ( seg )
  --}}}
state = transfer.moving.in.arbor.4
  --{{{  enable appropriate emergency transitions
  -- finish communications / motion to stop
  PAR
    from.transfer.moving.clear.to.rest ! TRUE      -- outputs
    --{{{  inputs
    SEQ
     transfer.timer ? timenow
     ALT
      from.arbor.at.rest ? prompt
        SKIP
      transfer.timer ? AFTER timenow + emergency.delay
        SKIP
     ALT
      from.arbor.moving ? prompt
        SKIP
      transfer.timer ? AFTER timenow + emergency.delay
        SKIP
```

```
    --}}}
    SEQ
     seg := 6
     motion.seg ( seg )
  --}}}
 state = transfer.moving.outside.arbor.to.rest
  --{{{  enable appropriate emergency transitions
  -- finish communications / motion to stop
  PAR
   from.transfer.moving.clear.to.rest ! TRUE      -- outputs
   --{{{  inputs
   SEQ
    transfer.timer ? timenow
    ALT
     from.arbor.at.rest ? prompt
      SKIP
     transfer.timer ? AFTER timenow + emergency.delay
      SKIP
    ALT
     from.arbor.moving ? prompt
      SKIP
     transfer.timer ? AFTER timenow + emergency.delay
      SKIP
   --}}}
   SEQ
    seg := 6
    motion.seg ( seg )
  --}}}
 state = transfer.moving.emergency.to.rest
  --{{{  enable appropriate emergency transitions
  -- finish communications / motion to stop
  PAR
   from.transfer.moving.clear.to.rest ! TRUE      -- outputs
   --{{{  inputs
   SEQ
    transfer.timer ? timenow
    ALT
     from.arbor.at.rest ? prompt
      SKIP
     transfer.timer ? AFTER timenow + emergency.delay
      SKIP
    ALT
     from.arbor.moving ? prompt
      SKIP
     transfer.timer ? AFTER timenow + emergency.delay
      SKIP
   --}}}
   SEQ
    seg := 6
    motion.seg ( seg )
  --}}}
 TRUE
  --{{{  problems  - invalid state
  SEQ
   STOP
  --}}}
```

```
    state := transfer.at.rest
    running := FALSE
    hard.emergency.stop := FALSE
  --}}}
((state = transfer.at.rest) AND initialise)
  --{{{  enable appropriate transitions
  SEQ
    from.arbor.at.rest ? prompt
    state := transfer.moving.to.enter.arbor
  --}}}
state = transfer.at.rest
  --{{{  enable appropriate transitions
  SEQ
    from.arbor.moving ? prompt
    state := transfer.moving.to.enter.arbor
  --}}}
((state = transfer.moving.to.enter.arbor) AND initialise)
  --{{{  enable appropriate transitions
  SEQ
    -- in initialise state (& after emergency) arbor is at rest
    -- so no further tests are required
    seg := 0
    motion.seg ( seg )
    initialise := FALSE          -- normal operation
    state := transfer.moving.in.arbor.1
  --}}}
state = transfer.moving.to.enter.arbor
  --{{{  enable appropriate transitions
  SEQ
    --{{{  initialise watchdog
    transfer.timer ? timenow
    --}}}
    seg := 0
    motion.seg ( seg )
    ALT
      from.arbor.at.rest ? prompt
        state := transfer.moving.in.arbor.1
      transfer.timer ? AFTER (timenow + watchdog.time)
        state := transfer.moving.emergency.to.rest
  --}}}
state = transfer.moving.in.arbor.1
  --{{{  enable appropriate transitions
  SEQ
    seg := 1
    motion.seg ( seg )
    state := transfer.moving.in.arbor.2
  --}}}
state = transfer.moving.in.arbor.2
  --{{{  enable appropriate transitions
  SEQ
    seg := 2
    motion.seg ( seg )
    state := transfer.moving.in.arbor.3
  --}}}
state = transfer.moving.in.arbor.3
```

```
  --{{{  enable appropriate transitions
  SEQ
    seg := 3
    motion.seg ( seg )
    state := transfer.moving.in.arbor.4
  --}}}
state = transfer.moving.in.arbor.4
  --{{{  enable appropriate transitions
  SEQ
    seg := 4
    motion.seg ( seg )
    state := transfer.moving.outside.arbor.to.rest
  --}}}
state = transfer.moving.outside.arbor.to.rest
  --{{{  enable appropriate transitions
  SEQ
    PAR
      from.transfer.moving.clear.to.rest ! TRUE
      SEQ
        seg := 5
        motion.seg ( seg )
        state := transfer.at.rest
        --{{{  cycle commands
        -- handle any cycle commands
        IF
          cycle.stop = TRUE
            SEQ
              running := FALSE
              -- absorb the transfer commence move message
              from.arbor.moving ? prompt
              initialise := TRUE            -- start up from arbor.at.rest
              to.HUI ! command    -- report on completion (single shot/stop)
          TRUE
            SKIP
        --}}}
  --}}}
state = transfer.moving.emergency.to.rest
  --{{{  enable appropriate transitions
  SEQ
    seg := 6
    motion.seg ( seg )
    --{{{  return to home position
    seg := 7
    motion.seg ( seg )
    seg := 8
    motion.seg ( seg )
    --}}}


    state := transfer.at.rest
    initialise := TRUE     -- restart on an arbor at rest signal only
    --{{{  cycle commands
    -- handle any cycle commands
    IF
      cycle.stop = TRUE
        SEQ
```

```
                running := FALSE
                -- absorb the transfer commence move message
                from.arbor.moving ? prompt
                initialise := TRUE          -- start up from arbor.at.rest
              TRUE
               SKIP
             --}}}
           --}}}
         TRUE
          --{{{  problems - invalid state
          SEQ
           STOP
          --}}}
        --}}}
    --}}}


:
--}}}
--{{{  Timing event activity
PROC Event.timer (CHAN OF INT velocity.demand.clock,
            CHAN OF BOOL HUI, INT rate )
 --{{{  description
 -- This process provides event sequencing to all of the  real-time activities
 -- of the servo system
 -- These include slow rate scanning of velocity demand ramps
 -- Faster scanning of servo/plant loops
 -- Faster scanning for plant state decode

 -- actual parameters, velocity demand generation sample rate 1s (1000ms)
 --              servo/plant loop rate              0.1s (100ms)
 --              plant state decode rate            0.1s (100ms)


 --}}}
 --{{{  code
 --{{{  Declarations

 VAL INT mtd IS 64          : -- 1 tick every 64us machine.tick.divisor
 --VAL INT vel.interval  IS 96  : -- vel gen sample rate 100000/mtd
 --VAL INT velocity.rate  IS 4   : -- sample rate in ms from above
 INT vel.interval, velocity.rate :
 VAL INT plant.interval IS 1562 :  -- plant sample rate 100000/mtd
 VAL INT plant.rate    IS 100  :  -- sample rate in ms from above
 VAL INT servo.interval IS 1562 :  -- vel gen sample rate 100000/mtd
 VAL INT servo.rate    IS 100  :  -- sample rate in ms from above
 INT plant.elapsed.time       : -- next event
 INT vel.elapsed.time         : -- next event
 INT servo.elapsed.time       : -- next event
 INT time                  : -- local scratchpad variable
 BOOL commence              : -- start / stop from HUI
 BOOL timer.enabled         : -- activity variable

 TIMER plant.decode.timer, velocity.demand.timer, servo.timer :
```

```
--}}}

SEQ
 --{{{  intialisation
 -- await initialisation
 HUI ? commence
 IF
  commence
   timer.enabled := TRUE
  NOT (commence)
   timer.enabled := FALSE

 --{{{  COMMENT  other activities
 --:::A COMMENT FOLD
 --{{{   other activities
 -- initialise plant state decode timer
 plant.decode.timer ? time
 -- set up next event for plant state
 plant.elapsed.time := time + plant.interval
 servo.timer ? time
 servo.elapsed.time := time + servo.interval
 -- set up next event for servo demand
 --}}}
 --}}}
 vel.interval := rate*16
 velocity.rate := rate

 -- initialise velocity demand generator timer
 velocity.demand.timer ? time
 -- set up next event for velocity demand
 vel.elapsed.time := time + vel.interval
 --}}}
 WHILE timer.enabled
  --{{{  monitor for stop and events
  -- monitor for stop and events
  ALT
   HUI ? commence
    --{{{
    SEQ
     IF
      commence
       timer.enabled := TRUE
      NOT (commence)
       SEQ
        --{{{  COMMENT  useless
        --:::A COMMENT FOLD
        --{{{
        -- shutdown servo
        servo.clock ! terminate
        -- shutdown plant state decode - start pipeline shutdown
        plant.decode.clock ! terminate
        -- shutdown velocity demand gen
        --}}}
        --}}}
```

```
            velocity.demand.clock ! terminate

            timer.enabled := FALSE
        --}}}
      velocity.demand.timer ? AFTER vel.elapsed.time
        --{{{
        SEQ
          -- set up next event for velocity demand
          velocity.demand.timer ? time
          vel.elapsed.time := time + vel.interval
          -- send event / time to vel demand gen
          velocity.demand.clock ! velocity.rate

        --}}}
      --{{{  COMMENT other activities
      --:::A COMMENT FOLD
      --{{{  other activities
      plant.decode.timer  ? AFTER plant.elapsed.time
        --{{{
        SEQ
          -- set up next event for plant state
          plant.decode.timer ? time
          plant.elapsed.time := time + plant.interval
          -- send event / time to plant decode
          plant.decode.clock ! plant.rate
        --}}}
      servo.timer ? AFTER servo.elapsed.time
        --{{{
        SEQ
          -- set up next event for servo sample
          servo.timer ? time
          servo.elapsed.time := time + servo.interval
          -- send event / time to servo
          servo.clock ! servo.rate
        --}}}
      --}}}
      --}}}
    --}}}

  --}}}
:
--}}}
--{{{  Velocity demand gen activity
PROC Velocity.demand.gen (CHAN OF INT velocity.demand.clock, to.servo,
                          to.accel.calc,
                    CHAN OF BOOL demand.satisfied )
  --{{{  description
  -- read in an accleration rate from the motion trajectory generator
  -- apply the rate to the velocity demand for the servo
  -- transmit new velocity demands at appropriate intervals

  --}}}
  --{{{  code

  --{{{  declarations
  INT accel.conversion       :     -- convert accel rate to velocity
```

```
                        -- generator sample rate
BOOL velocity.demand.enabled :     -- activity variable
INT acceleration.rate       :     -- rate parameter to vel gen
INT new.demand              :     -- velocity demand to servo
INT vel.sample.rate         :     -- sample rate from event timer
INT position.increment      :     -- used in acceleration calc
INT terminal.velocity       :     --  "  "       "       "
INT current.velocity        :     --  "  "       "       "
INT divisor                 :     --  "  "       "       "
INT mantissa                :     --  "  "       "       "
BOOL unfinished.flag        :     -- an acknowledgement of motion competion
                        -- is required from vel to HUI/motion disp
--}}}

SEQ
  --{{{  await initialisation
  -- await initialisation
  velocity.demand.clock ? vel.sample.rate
  new.demand  := 0
  acceleration.rate := 0
  -- calculate update rate
  IF
    vel.sample.rate > 0
      -- to convert ec/s/s to ec/sample.rate.squared
      -- nominal sample rate is 1 s
      -- 1000.0 divisor converts from ms to s
      -- note accel.conversion is a divisor (for int arithmetic)
      accel.conversion := (1000/vel.sample.rate)
    TRUE
      accel.conversion := 0
  velocity.demand.enabled := TRUE
  --}}}
  WHILE velocity.demand.enabled
    SEQ
      -- monitor for new rate demand/shutdown or clock start
      ALT
        velocity.demand.clock ? vel.sample.rate
          --{{{  calc / output new velocity demand or shutdown
          SEQ
            IF
              vel.sample.rate = terminate
                SEQ
                  velocity.demand.enabled := FALSE
              TRUE
                SEQ
                  -- calc new vel demand
                  new.demand := new.demand + acceleration.rate
                  --{{{  limit checks
                  -- limiting checks here if you please
                  IF
                    new.demand > transfer.max.vel
                      new.demand := transfer.max.vel
                    new.demand < (-transfer.max.vel)
                      new.demand := -transfer.max.vel
                    (new.demand <= terminal.velocity) AND
                    ((acceleration.rate < 0) AND unfinished.flag)
```

```
            --{{{
            SEQ
              new.demand := terminal.velocity  -- remove rounding
              demand.satisfied ! TRUE
              acceleration.rate := 0
              unfinished.flag := FALSE
            --}}}
          (new.demand >= terminal.velocity) AND
          ((acceleration.rate > 0) AND unfinished.flag)
            --{{{
            SEQ
              new.demand := terminal.velocity  -- remove rounding
              demand.satisfied ! TRUE
              acceleration.rate := 0
              unfinished.flag := FALSE
            --}}}
          TRUE
            SKIP
        -- output new vel demand
        --}}}
        to.servo ! new.demand   -- new demand is current velocity
    --}}}

  to.accel.calc ? position.increment
    SEQ
      to.accel.calc ? terminal.velocity
      unfinished.flag := TRUE
      --{{{  perform accel calc
      -- calculate acceleration rate
      -- using V2=U2+2*A*S
      divisor := 2*position.increment

      --{{{  GET CURRENT.VELOCITY FROM SOMEWHERE
      current.velocity := new.demand          -- for open loop only

      --}}}

      -- problem with negative terminal velocities / positive initial velocities
      -- and vice versa, if this occurs flag error
      -- check gradient of acceleration for substitution in final parameter
      IF
        (terminal.velocity > 0) AND (current.velocity < 0)
          STOP
        (terminal.velocity < 0) AND (current.velocity > 0)
          STOP
        TRUE
          SKIP

      mantissa:=((terminal.velocity*terminal.velocity) -
            (current.velocity*current.velocity))
      acceleration.rate := ( mantissa / divisor )
      --{{{  COMMENT  zero demand detection --       >>>>>>> DEBUGGING
<<<<<<<<
      --:::A COMMENT FOLD
      --{{{
      IF
```

```
          (acceleration.rate = 0) AND (new.demand = 0)
            STOP          -- stationary
          TRUE
            SKIP
        --}}}
        --}}}
        -- result using ec/s and ec for v,s is in ec/s/s

        --{{{  new rate
        SEQ
          -- convert acceleration to fit sample rate
          -- convert from ec/s/s to ec/sample rate squared
          acceleration.rate := acceleration.rate/accel.conversion
        --}}}
        --}}}
 --}}}
:
--}}}
SEQ
 --{{{  initialisation
 --{{{  set up sample speed
 -- set up sample speed
 --1< sample speed <10
 rate := 5              -- preset sample rate

 --}}}

 -- set up motion segment array
 --{{{  segment 0
 --  to max +ive velocity
 Motion.segment[0][ms.pos.max]     := 1000   -- max incremental position (ec)
 Motion.segment[0][ms.pos.min]     := 1000   -- min incremental position (ec)
 Motion.segment[0][ms.vel.max]     := 40000  -- max terminal velocity (ec/s)
 Motion.segment[0][ms.vel.min]     := 4000   -- min terminal velocity (ec/s)
 Motion.segment[0][ms.flex.pos]    := 0      -- position flexibility variable
 Motion.segment[0][ms.flex.vel]    := 1      -- velocity flexibility variable
 Motion.segment[0][ms.next.seg.ok] := 1      -- seq sequence   - next if ok
 Motion.segment[0][ms.next.seg.err] := 6     --                next if error

          -- NB flex variable of 0 = indicates no flexibility
          --                 min parameter will be executed
 --}}}
 --{{{  segment 1
 --  slow to contact packet
 Motion.segment[1][ms.pos.max]     := 2000   -- max incremental position (ec)
 Motion.segment[1][ms.pos.min]     := 1100   -- min incremental position (ec)
 Motion.segment[1][ms.vel.max]     := 20000  -- max terminal velocity (ec/s)
 Motion.segment[1][ms.vel.min]     := 2000   -- min terminal velocity (ec/s)
 Motion.segment[1][ms.flex.pos]    := 2      -- position flexibility variable
 Motion.segment[1][ms.flex.vel]    := 1      -- velocity flexibility variable
 Motion.segment[1][ms.next.seg.ok] := 2      -- seq sequence   - next if ok
 Motion.segment[1][ms.next.seg.err] := 2     --                next if error

          -- NB flex variable of 0 = indicates no flexibility
          --                 min parameter will be executed
 --}}}
```

```
--{{{  segment 2
-- enter arbor
--  accelerate to max position
Motion.segment[2][ms.pos.max]     := 4000   -- max incremental position (ec)
Motion.segment[2][ms.pos.min]     := 4900   -- min incremental position (ec)
-- note position  swap to allow -ive pos change from increasing flex param

Motion.segment[2][ms.vel.max]     := 40000  -- max terminal velocity (ec/s)
Motion.segment[2][ms.vel.min]     := 4000   -- min terminal velocity (ec/s)
Motion.segment[2][ms.flex.pos]    := 2      -- position flexibility variable
Motion.segment[2][ms.flex.vel]    := 1      -- velocity flexibility variable
Motion.segment[2][ms.next.seg.ok] := 3      -- seq sequence   - next if ok
Motion.segment[2][ms.next.seg.err] := 3     --                next if error

            -- NB flex variable of 0 = indicates no flexibility
            --                  min parameter will be executed

--}}}
--{{{  segment 3
--  to max +ive position
Motion.segment[3][ms.pos.max]     := 1000   -- max incremental position (ec)
Motion.segment[3][ms.pos.min]     := 1000   -- min incremental position (ec)
Motion.segment[3][ms.vel.max]     := 0      -- max terminal velocity (ec/s)
Motion.segment[3][ms.vel.min]     := 0      -- min terminal velocity (ec/s)
Motion.segment[3][ms.flex.pos]    := 0      -- position flexibility variable
Motion.segment[3][ms.flex.vel]    := 0      -- velocity flexibility variable
Motion.segment[3][ms.next.seg.ok] := 4      -- seq sequence   - next if ok
Motion.segment[3][ms.next.seg.err] := 4     --                next if error

            -- NB flex variable of 0 = indicates no flexibility
            --                  min parameter will be executed

--}}}
--{{{  segment 4
--  to rest (home position)
Motion.segment[4][ms.pos.max]     := -6200  -- max incremental position (ec)
Motion.segment[4][ms.pos.min]     := -6200  -- min incremental position (ec)
Motion.segment[4][ms.vel.max]     := -40000 -- max terminal velocity (ec/s)
Motion.segment[4][ms.vel.min]     := -4000  -- min terminal velocity (ec/s)
Motion.segment[4][ms.flex.pos]    := 2      -- position flexibility variable
Motion.segment[4][ms.flex.vel]    := 1      -- velocity flexibility variable
Motion.segment[4][ms.next.seg.ok] := 5      -- seq sequence   - next if ok
Motion.segment[4][ms.next.seg.err] := 5     --                next if error

            -- NB flex variable of 0 = indicates no flexibility
            --                  min parameter will be executed

--}}}
--{{{  segment 5
-- clear of arbor
--  stationary at rest (home position)
Motion.segment[5][ms.pos.max]     := -1800  -- min incremental position (ec)
Motion.segment[5][ms.pos.min]     := -1800  -- min incremental position (ec)
Motion.segment[5][ms.vel.max]     := 0      -- max terminal velocity (ec/s)
Motion.segment[5][ms.vel.min]     := 0      -- min terminal velocity (ec/s)
Motion.segment[5][ms.flex.pos]    := 0      -- position flexibility variable
```

```
Motion.segment[5][ms.flex.vel]    := 0      -- velocity flexibility variable
Motion.segment[5][ms.next.seg.ok] := 0      -- seq sequence   - next if ok
Motion.segment[5][ms.next.seg.err] := 0     --                next if error

            -- NB flex variable of 0 = indicates no flexibility
            --                  min parameter will be executed

--}}}
--{{{  segment 6   -- abort enter
Motion.segment[6][ms.pos.max]     := 1000   -- max incremental position (ec)
Motion.segment[6][ms.pos.min]     := 1000   -- min incremental position (ec)
Motion.segment[6][ms.vel.max]     := 0      -- max terminal velocity (ec/s)
Motion.segment[6][ms.vel.min]     := 0      -- min terminal velocity (ec/s)
Motion.segment[6][ms.flex.pos]    := 0      -- position flexibility variable
Motion.segment[6][ms.flex.vel]    := 1      -- velocity flexibility variable
Motion.segment[6][ms.next.seg.ok] := 0      -- seq sequence   - next if ok
Motion.segment[6][ms.next.seg.err] := 0     --                next if error

            -- NB flex variable of 0 = indicates no flexibility
            --                  min parameter will be executed

--}}}
--{{{  segment 7   -- return to home
Motion.segment[7][ms.pos.max]     := -1000  -- max incremental position (ec)
Motion.segment[7][ms.pos.min]     := -1000  -- min incremental position (ec)
Motion.segment[7][ms.vel.max]     := -40000     -- max terminal velocity (ec/s)
Motion.segment[7][ms.vel.min]     := -4000      -- min terminal velocity (ec/s)
Motion.segment[7][ms.flex.pos]    := 0      -- position flexibility variable
Motion.segment[7][ms.flex.vel]    := 1      -- velocity flexibility variable
Motion.segment[7][ms.next.seg.ok] := 0      -- seq sequence   - next if ok
Motion.segment[7][ms.next.seg.err] := 0     --                next if error

            -- NB flex variable of 0 = indicates no flexibility
            --                  min parameter will be executed

--}}}
--{{{  segment 8   -- position at rest
Motion.segment[8][ms.pos.max]     := -1000  -- max incremental position (ec)
Motion.segment[8][ms.pos.min]     := -1000  -- min incremental position (ec)
Motion.segment[8][ms.vel.max]     := 0      -- max terminal velocity (ec/s)
Motion.segment[8][ms.vel.min]     := 0      -- min terminal velocity (ec/s)
Motion.segment[8][ms.flex.pos]    := 0      -- position flexibility variable
Motion.segment[8][ms.flex.vel]    := 1      -- velocity flexibility variable
Motion.segment[8][ms.next.seg.ok] := 0      -- seq sequence   - next if ok
Motion.segment[8][ms.next.seg.err] := 0     --                next if error

            -- NB flex variable of 0 = indicates no flexibility
            --                  min parameter will be executed

--}}}

-- set up flexibility parameters default values
--{{{  init flex.params
flex.param[0] := 0      -- not used
flex.param[1] := 0      -- valid range 0-9       speed
flex.param[2] := 0      -- valid range 0-9       position - pack size
```

```
  --}}}

  --}}}
  --{{{  main program
  CHAN OF BOOL HUI, demand.satisfied :
  CHAN OF INT to.accel.calc, velocity.demand.clock :
  PAR
    Velocity.demand.gen ( velocity.demand.clock,
                          to.servo,
                          to.accel.calc,
                          demand.satisfied )
    Event.timer       ( velocity.demand.clock,
                          to.event, rate )
    Motion.demand     ( from.HUI.flex,
                          from.HUI.command,
                          to.HUI,
                          to.accel.calc,
                          demand.satisfied )
  --}}}
:
--}}}
--{{{  HUI
PROC HUI.process ( [2]CHAN OF BOOL to.event,
                   [2]CHAN OF INT from.HUI.flex,
                   from.HUI.command,
                   [2]CHAN OF ANY to.HUI,
                   CHAN OF BOOL stop.dac )
 -- starts stops activity using a pipeline routed through event timer
 --{{{  declarations
 BOOL HUI.report, HUI.command, running, any :  -- activity variable, loop control
 INT keypress, loop          : -- holds character, for display
 INT position, velocity, result :  -- for motion generation
 INT param.id, value, report, replies :
 VAL INT s IS  115:
 VAL INT key.0 IS 48 :
 VAL INT key.1 IS 49 :
 VAL INT key.2 IS 50 :
 VAL INT key.3 IS 51 :
 VAL INT key.4 IS 52 :
 VAL INT key.5 IS 53 :
 VAL INT key.6 IS 54 :
 VAL INT key.7 IS 55 :
 VAL INT key.8 IS 56 :
 VAL INT key.9 IS 57 :

 TIMER clock1 :
 INT timenow :
 VAL INT delay IS 1000 :

 [5]INT copy.flex.param  : -- holds a copy of flex params for display only
 CHAN OF BOOL report.run :
 --}}}
 SEQ
  --{{{  initialisation
  --{{{  display start up page
```

```
IF
 keypress = key.1
  --{{{  single shot motion
  SEQ
    -- commence run
    from.HUI.command[arbor] ! single.shot    -- single shot motion demand
    from.HUI.command[transfer] ! single.shot    -- single shot motion demand
    replies := 2
  --}}}
 keypress = key.2
  --{{{  start motion
  SEQ
    -- commence run
    from.HUI.command[arbor] ! start    -- start up motion demand
    from.HUI.command[transfer] ! start    -- start up motion demand
    replies := 2
  --}}}
 keypress = key.3
  --{{{  stop motion
  SEQ
    -- stop run
    from.HUI.command[transfer] ! stop    -- stop motion demand
    from.HUI.command[arbor] ! stop       -- stop motion demand
    replies := 2
  --}}}
 keypress = key.4
  --{{{  shutdown.program
  SEQ
    --{{{  COMMENT optional stop motion
    --:::A COMMENT FOLD
    --{{{  optional stop motion
    -- stop motion        -- optional
    from.HUI.command[arbor] ! emergency
    from.HUI.command[transfer] ! emergency
    to.HUI ? report
    --{{{  timer
    -- await emergency stop to execute
    clock1 ? timenow
    clock1 ? AFTER (timenow+delay)
    --}}}
    --}}}
    --}}}

    -- close down motion demand
    from.HUI.command[arbor] ! shutdown
    from.HUI.command[transfer] ! shutdown
    -- close down procedures
    to.event[arbor] ! FALSE       -- stop timer process
    to.event[transfer] ! FALSE    -- stop timer process
    HUI.command := FALSE
    replies := 2
    stop.dac ! FALSE       -- stop dac process
  --}}}
keypress = key.5
  --{{{  emergency stop
  SEQ
```

```
          -- stop motion
          from.HUI.command[arbor] ! emergency
          from.HUI.command[transfer] ! emergency
          replies := 2

write.text.line(screen,"%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%")
          write.text.line(screen,"%%   EMERGENCY STOP  -                    %%")
          write.text.line(screen,"%%                                    %%")
          write.text.line(screen,"%%   ACTUATORS NEED POSITION
INITIALISATION    %%")

write.text.line(screen,"%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%")
        --}}}
       keypress = key.6
         --{{{  change flexibility
         SEQ
          newline ( screen )
          write.full.string (screen, "enter parameter    1-4               ")
          --{{{  receive / decode keypress
          keyboard ? keypress
          param.id := -999999
          IF
            keypress = key.1
              --{{{
              SEQ
                write.text.line ( screen, "parameter 1 selected")
                param.id := 1
              --}}}
            keypress = key.2
              --{{{
              SEQ
                write.text.line ( screen, "parameter 2 selected")
                param.id := 2
              --}}}
            keypress = key.3
              --{{{
              SEQ
                write.text.line ( screen, "parameter 3 selected")
                param.id := 3
              --}}}
            keypress = key.4
              --{{{
              SEQ
                write.text.line ( screen, "parameter 4 selected")
                param.id := 4
              --}}}
            TRUE
              --{{{  invalid command
              SEQ
                write.text.line ( screen , ">>>>>>>>>>> INVALID COMMAND
<<<<<<<<<<<")
              --}}}
          --}}}
          IF
```

```
param.id < 1
  -- error
  SEQ
    replies := 0
param.id < 5
  SEQ
    write.full.string (screen, "enter operational value 0-9          ")
    --{{{  receive / decode keypress
    keyboard ? keypress
    value := -999999
    IF
      keypress = key.0
        --{{{
        SEQ
          write.text.line ( screen, "value 0 selected")
          value := 0
        --}}}
      keypress = key.1
        --{{{
        SEQ
          write.text.line ( screen, "value 1 selected")
          value := 1
        --}}}
      keypress = key.2
        --{{{
        SEQ
          write.text.line ( screen, "value 2 selected")
          value := 2
        --}}}
      keypress = key.3
        --{{{
        SEQ
          write.text.line ( screen, "value 3 selected")
          value := 3
        --}}}
      keypress = key.4
        --{{{
        SEQ
          write.text.line ( screen, "value 4 selected")
          value := 4
        --}}}
      keypress = key.5
        --{{{
        SEQ
          write.text.line ( screen, "value 5 selected")
          value := 5
        --}}}
      keypress = key.6
        --{{{
        SEQ
          write.text.line ( screen, "value 6 selected")
          value := 6
        --}}}
      keypress = key.7
        --{{{
        SEQ
```

```
            write.text.line ( screen, "value 7 selected")
            value := 7
          --}}}
        keypress = key.8
          --{{{
          SEQ
            write.text.line ( screen, "value 8 selected")
            value := 8
          --}}}
        keypress = key.9
          --{{{
          SEQ
            write.text.line ( screen, "value 9 selected")
            value := 9
          --}}}
        TRUE
          --{{{  invalid command
          SEQ
            write.text.line ( screen , ">>>>>>>>>>> INVALID KEY
<<<<<<<<<<<")
          --}}}

      newline (screen)
    --}}}
    --{{{  transmit new parameters
    -- note flexibility will have to be directed to the appropriate axis
    IF
      value < 0
        -- error
        SEQ
          replies := 0
      value < 10
        SEQ
          -- update
          copy.flex.param[param.id] := value   -- update local copy
          IF
            param.id = 1          --  speed arbor
              --{{{
              SEQ
                from.HUI.flex[arbor] ! 1
                from.HUI.flex[arbor] ! value
              --}}}
            param.id = 2          --  speed transfer
              --{{{
              SEQ
                from.HUI.flex[transfer] ! 1
                from.HUI.flex[transfer] ! value
              --}}}
            param.id = 3          --  pack size
              --{{{
              SEQ
                from.HUI.flex[transfer] ! 2
                from.HUI.flex[transfer] ! value
              --}}}
            param.id = 4          --  transfer start
              --{{{
```

```
                SEQ
                  from.HUI.flex[arbor] ! 2
                  from.HUI.flex[arbor] ! value
                --}}}
               TRUE
                SKIP
              replies := 1
            TRUE
             SKIP
           --}}}
         TRUE
          SKIP
      --}}}
     keypress = key.7
      --{{{  display flexibility
      SEQ
       newline(screen)
       newline(screen)
       write.text.line ( screen, ">>>>   flexibility parameters   <<<<")
       newline(screen)
       write.full.string ( screen, "FP 1,     arbor speed          = " )
       write.int(screen, copy.flex.param[1], 2)
       newline(screen)
       write.full.string ( screen, "FP 2,     transfer speed       = " )
       write.int(screen, copy.flex.param[2], 2)
       newline(screen)
       write.full.string ( screen, "FP 3,     pack size            = " )
       write.int(screen, copy.flex.param[3], 2)
       newline(screen)
       write.full.string ( screen, "FP 4, transfer start distance   = " )
       write.int(screen, copy.flex.param[4], 2)
       newline(screen)
       replies := 0
      --}}}
     TRUE
      --{{{  invalid command
      SEQ
       write.text.line ( screen , ">>>>>>>>>>>> INVALID COMMAND
<<<<<<<<<<<<")
       replies := 0
      --}}}

   --{{{  command acknowledge
   IF
    replies > 0
     --{{{  await / decode report
     SEQ loop = 0 FOR replies
      SEQ
       ALT
        to.HUI[arbor] ? report
         write.full.string (screen, "Report from <<< ARBOR >>>  " )
        to.HUI[transfer] ? report
         write.full.string (screen, "Report from <<< TRANSFER >>>  " )
       IF
        report = single.shot
         write.text.line (screen, "command >>> SINGLE SHOT <<< executed" )
```

```
            report = start
              write.text.line (screen, "command >>> START <<< executed" )
            report = stop
              write.text.line (screen, "command >>> STOP <<< executed" )
            report = shutdown
              write.text.line (screen, "command >>> SHUTDOWN <<< executed" )
            report = emergency
              write.text.line (screen, "command >>> EMERGENCY STOP <<< executed"
)
            report = flex.read
              write.text.line (screen, "command >>> FLEXIBILITY <<< executed" )
            report = bad.read
              write.text.line (screen, "%%%%%%%%   BAD COMMAND
%%%%%%%%" )
            TRUE
              write.text.line (screen, "%%%%%%%%   BAD REPORT
%%%%%%%%" )

        --}}}

      TRUE
        SKIP
    --}}}

    --}}}
  --}}}
:
--}}}
--{{{  Real time DAC o/p
PROC dac ( CHAN OF INT arbor.to.dac, transfer.to.dac,
        CHAN OF dac.protocol to.dac,
        CHAN OF BOOL from.HUI )

  --{{{  description
  -- Takes the velocity demands fed to the plant model
  -- and transmits it to the dac board for a r-time output.
  -- Output is scaled so that +-10v = max.vel
  -- and appears on dac channel 0 for arbor and channel 1 for transfer
  --}}}
  --{{{  code
  --{{{  external protocols / channel defn, implementation details

  PROC init.dac()
    -- This is a software initialisation routine for the DAC. It is used to
    -- synchronise the byte order.
    -- Subsequent data sent to the DAC should be : lo.byte;hi.byte;channel.byte.
    SEQ
      to.dac ! BYTE #FF;BYTE #FF;BYTE #FF
      to.dac ! BYTE #FF;BYTE #FF;BYTE #06
  :

  --}}}
  --{{{  other declarations
  VAL INT terminate IS -999999 :
  BOOL dac.op :
  INT velocity :
```

```
INT64 scaled.vel :
BYTE hi.byte, lo.byte, channel1, channel2 :
--}}}

SEQ
 --{{{  await initialisation
 dac.op   := TRUE
 hi.byte  := #00 (BYTE)
 lo.byte  := #00 (BYTE)
 channel1 := #00 (BYTE)
 channel2 := #01 (BYTE)
 -- init.dac()
 -- to.dac ! lo.byte;hi.byte;channel1
 -- to.dac ! lo.byte;hi.byte;channel2
 --}}}
 WHILE dac.op
  SEQ
   -- get new velocity from either source
   ALT
    arbor.to.dac  ? velocity
     --{{{  scale and output
     SEQ
      --{{{  check velocity is within limits
      IF
       velocity > arbor.max.vel
        velocity := arbor.max.vel
       velocity < (-arbor.max.vel)
        velocity := -arbor.max.vel
       TRUE
        SKIP
      --}}}
      --{{{  convert & output value to dac
      -- BYTE value #7FFF = +10v
      -- BYTE value #3FFF =  0v
      -- BYTE value #0000 = -10v

      -- set up 0v and scale

      --{{{  scale velocity for HP ANALYSER
      -- /10 scaling for HP logic analyser
      velocity := velocity / 10
      --}}}

      scaled.vel:= INT64(((velocity+(arbor.max.vel))*4095)/(2*(arbor.max.vel)))
      hi.byte :=BYTE( scaled.vel / 256(INT64) )
      lo.byte :=BYTE( scaled.vel REM 256(INT64) )
      --to.dac ! lo.byte;hi.byte;channel1
      --}}}
     --}}}
    transfer.to.dac ? velocity
     --{{{  scale and output
     SEQ
      --{{{  check velocity is within limits
      IF
       velocity > transfer.max.vel
        velocity := transfer.max.vel
```

```
            velocity < (-transfer.max.vel)
              velocity := -transfer.max.vel
            TRUE
              SKIP
          --}}}
          --{{{  convert & output value to dac
          -- BYTE value #7FFF = +10v
          -- BYTE value #3FFF =  0v
          -- BYTE value #0000 = -10v

          -- set up 0v and scale
          --{{{  scale velocity for HP ANALYSER
          -- /10 scaling for HP logic analyser
          velocity := velocity / 10
          --}}}

          scaled.vel:= INT64(((velocity+(transfer.max.vel))*4095)/(2*(transfer.max.vel)))
          hi.byte :=BYTE( scaled.vel / 256(INT64) )
          lo.byte :=BYTE( scaled.vel REM 256(INT64) )
          --to.dac ! lo.byte;hi.byte;channel2
          --}}}
        --}}}
      from.HUI ? dac.op    -- termination procedure
        SKIP
 --}}}

:
--}}}

--{{{  main program
-- outer level represents network, no shared variables
--{{{  channel declarations
[2]CHAN OF ANY  to.HUI :
[2]CHAN OF INT  from.HUI.flex,
           from.HUI.command :
[2]CHAN OF BOOL to.event :
CHAN OF BOOL    from.arbor.at.rest,
           from.arbor.moving,
           to.transfer.cleared.arbor,
           stop.dac :
CHAN OF INT     arbor.to.dac,
           transfer.to.dac:

--}}}
PAR
 arbor.drum          ( to.event[arbor],
                   from.HUI.flex[arbor],
                   from.HUI.command[arbor],
                   to.HUI[arbor],
                   from.arbor.at.rest,
                   from.arbor.moving,
                   to.transfer.cleared.arbor,
                   arbor.to.dac)
 third.transfer      ( to.event[transfer],
                   from.HUI.flex[transfer],
                   from.HUI.command[transfer],
```

```
                to.HUI[transfer],
                to.transfer.cleared.arbor,
                from.arbor.at.rest,
                from.arbor.moving,
                transfer.to.dac)
HUI.process     ( to.event,
                from.HUI.flex,
                from.HUI.command,
                to.HUI,
                stop.dac)
dac             ( arbor.to.dac,
                transfer.to.dac,
                to.dac,
                stop.dac )
--}}}
```

8.3 Paper : Modular machine systems

MODULAR MACHINE SYSTEMS

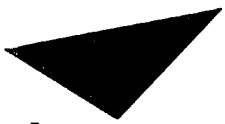
L. Fenney *, MSc, **C.M. Draper** *, BSc, **K. Foster** **, MA, PhD, CEng, FIMechE,
D.J. Holding *, BSc, PhD, CEng, FIEE, MinstMC, MBCS
* Aston University, ** Birmingham University.

Aston University

8.4 Paper : The specification and fast-prototyping of a distributed real-time computer control system for a modular independently driven high-speed machine

C.M.Draper, D.J.Holding

Aston University, UK



Aston University

8.5 Paper : Specification and verification of the real-time synchronisation software for a modularindependently driven high-speed machine

BY C.M.DRAPER, D.J.HOLDING ASTON UNIVERSITY

Aston University