ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

A Petri Net-Occam Based Methodology for the Development of Dependable Distributed Control Software

Peter Andrew Gray BSc MTech MSc

A thesis submitted in partial fulfilment of the requirements of Sheffield Hallam University for the degree of Doctor of Philosophy

December 1995

Abstract

Analysis of flexible manufacturing cells (FMCs) shows their requirement for flexible, correct, reliable, safe and distributed control. A comparison of the state of the art in software engineering for parallel systems, and an examination of safety related systems, reveal a need for formal and rigorous techniques at all stages in the software life cycle. However, parallel software, safety related software and formal techniques are complex. It is better to avoid faults rather than eliminate or tolerate them, and although less flexible, avoidance is often simpler to implement.

There is a need for a tool which overcomes many of these complexities, and this thesis discusses and defines such a tool in the form of a methodology. The novelty of the work is in the combination of the core goals to manage these issues, and how the strategies guide the user to a solution which will not deadlock and which is comprehensible.

Place-transition Petri nets are an ideal representation for designing and modelling the interaction of concurrent (and distributed) processes. Occam is a high level real time parallel language designed to execute on one or a network of transputers. Transputers are processing, memory and communication building blocks, and, together with occam, are shown to be suitable for controlling and communicating the control as the DCS in FMCs.

The methodology developed in this thesis adopts the mathematically based tools of Petri nets, occam and transputers, and, by exploiting their structural similarities, incorporates them in a steps and tasks to improve the development of correct, reliable and hence safe occam code. The four steps: identify concurrent and sequential operations, produce Petri net graphs for all controllers, combine controller Petri net graphs and translate Petri net graphs into occam; are structured around three core goals: Petri net/occam equivalence, comprehensibility and pro-activity; which are manifest in four strategies: output-work-backwards, concurrent and sequential actions, structuralise and modularise, and deadlock avoidance.

The methodology assists in all stages of the software development life cycle, and is applicable to small DCSs such as an FMC. The methodology begins by assisting in the creation of DCS requirements from the manufacturing requirements of the FMC, and guides the user to the production of dependable occam code. Petri nets allow the requirements to be specified as they are created, and the methodology's imposed restrictions enable the final Petri net design to be translated directly into occam. Thus the mathematics behind the formal tools is hidden from the user, which should be attractive to industry.

The methodology is successfully applied to the example FMC, and occam code to simulate the FMC is produced. Due to the novelty of the research, many suggestions for further work are given.

Contents

Figures

Tables

Plates

Dedication

My father and maternal grandmother died while I was studying for this degree. I wish to dedicate this thesis to their memory, and to my mother.

Acknowledgements

I would like to express my sincere gratitude to my supervision team: Dr W Hales, Mr A Goude and Prof F Poole, and to Prof E Lo and Mr G Cockerham who encouraged my return.

Thanks also to friends, colleagues, technicians and security staff.

Declaration

I declare while registered as a candidate for the University's research degree, I have not been a registered candidate or enrolled student for another award of the University or other academic or professional organisation. I further declare that no material contained in this thesis has been used in any other submission for an academic award.

Peter Gray

1. Introduction

1.1 Research Applicability

Flexible manufacturing cells (FMCs) are at the heart of some modern production methods. Flexibility, in rapidly changing markets, is a distinct competitive advantage. Being able to alter flexible production facilities to meet market demands requires a tool to facilitate the development of manufacturing control and to produce the software to implement that control and communicate that control to the machine tools and work handling equipment of an FMC. The tool, the research methodology, presented in this thesis, is a combination of two existing tools and a set of rules.

Flexible production seems to cope with current market conditions better than traditional methods. Markets want more variety, developed and manufactured quicker.

The introduction begins by comparing production methods, then outlines strategies aimed at improving manufacturing and product development times, and gives examples of their use in the car industry. Many of the strategies discussed achieve flexibility through the use of computers, in product development (CAD), for future FMC layouts (CAPP) or in manufacturing control (CAM) of currently operating machine configurations, and are considered for planned and actual transputer implementation. The introduction ends with the evolution and boundaries of the research and the structure of the thesis.

1.2 Traditional Production Methods

Until the mid 1980's the predominant reliable production techniques available to manufacturers of discrete components were craft, mass and batch production[Mair 1993].

In craft production, a worker skilled in a range of manufacturing processes can produce a low volume of high quality products to the customers' specifications. The worker, like his environment the job-shop, is general-purpose and flexible. The worker is able to manufacture the components in any order, in any suitable route between machines, and fits them together. Generally one product is made at a time, and premium prices can be charged for quality and exclusivity.

In mass production, companies specialise production (machines and labour) to make high volumes of toleranced components and assemble them into a limited variety of standardised products. Material flow in manufacturing and assembly is linear and continuous. High stocks are made at all stages of production to prevent stoppages. This is crucial in assembly, but due to low quality many products must be repaired or scrapped. Low cost per unit is attained by high volumes, specialisation and standardisation.

In batch production, components are manufactured within tolerances, and are taken to each manufacturing process in the same lot size. Batches vary in size and production frequency. [Duff 1992] indicates that the smaller the batch the shorter the manufacturing queue, however for every change in batch there is a change in set-up.

In mass production, machine tools are customised to manufacture individual components, whereas such specialisation is absent in craft production. General-purpose and semi-automatic (e.g. capstan lathes) machine tools and batch-specific jigs and fixtures are used in batch production. This leads to batch production being more efficient but less flexible than craft production, and more flexible but less efficient than mass production.

1.3 Flexible Production

Customers have always had an infinite variety of wants [McCormick 1977], but are limited by their income, and therefore make choices and demand products with the lowest prices. Manufacturers can provide a high volume of standard low cost products (mass production), a low volume of high cost customised products (craft production), or a variety of medium to high cost products in low to medium quantities (batch production), but have been unable to manufacture a wide variation of products at relatively low cost and at a quality and quantity to suit the customer.

Flexible production [Morales 1994] has presented an extra dimension to production, with the maturation of reliable computerised and co-operating development and production processes. Rather than mass production competing on cost per unit, craft production on quality and exclusivity and batch production interposing, manufacturers, using flexible production, are finding and serving niche markets and competing on flexibility whilst achieving relatively low cost and high quality. The benefits of flexibility include: introducing new products quicker, producing many variations, coping with different order sizes, supplying customised items and limited editions, and manufacturing for new markets.

To fully service the competitive advantage of flexibility, manufacturers must reduce time-to-market (which includes product concept, development, manufacture and launch) and/or order-to-delivery time (which includes manufacturing lead time). In market or customer orientated competition, responses must be swift and flexible. The marginal cost of flexibility using computerised development and manufacturing processes is relatively low, but the premium prices (possible through satisfying customer demand) must be charged to recoup the high capital investment in advanced technology plant and the necessary changes in corporate organisation, strategies and management processes. Strategies available to help shorten the manufacturing lead and product development times are given in sections 1.4 and 1.5.

1.4 Shortening Order-to-delivery and Manufacturing Lead Time

Order processing, production and delivery must be able to cope with batches of the size of a typical order. Work-in-progress and through-put time must be minimised, and bought-in parts must be available quickly. Production processes will have to be flexible to cope with different products and fast changes in batches of products. Strategies to help shorten order-to-delivery time are group technology, cellular manufacturing, flexible manufacturing, just-in-time and flexible manufacturing systems:

Group Technology (GT)

Men, machines and materials are grouped together in such a way to make groups or families of parts [Wemmerlov 1987]. All of the parts in a family are able to be manufactured on the same group of

processes or machines, which greatly reduces set-up times per batch. The lay-out of the machines can either be random (job-shop flow), or can reflect the manufacturing order of the processes/operations (uni-directional or line flow), but in both cases the close proximity of the machines reduces material handling time and improves through-put.

Cellular Manufacturing (CM)

[Williams 1994] gives three views of cells:

- cells usually contain a small number of closely co-operating machines sharing dimensional data, floor-space and human workers
- biologically - cells are the smallest autonomous unit capable of sustained production
- automated manufacture- cells are a subset of group technology

CM [Opitz 1971] uses batch production, the parts' families and machine layout techniques of GT, and generally uses uni-directional material flow. A family of parts can be manufactured on one group of machines, and those machines are grouped close together in a cell. Movement of material between cells is also minimised to reduce material handling and improve through-put. A cell is relatively independent, this and its other characteristics need simple control, which renders it ideal for automation and computer control.

Flexible Manufacturing (FM)

Automation [Mair 1993] improves the quality and the through-put speed of components, via precise, repeatable, fast and prior determination of machine set-ups (via cam, pneumatic, or electrical actuators). It reduces the need for, changes the function of, and improves the working condition of labour. Integrating computerised controllers with automated machines and materials handling equipment makes them more flexible. Computer numerically controlled (CNC) machines, such as CNC machine tools, machining centres and robots, can perform a sequence of operations that are initiated by a given, or stored, list of instructions (a part program). CNC machines controlled by part programs offer many advantages:

- the number and order of automatic operations are not governed by mechanical (e.g. turret indexing) or hard-wired (e.g. peg-board) limitations
- machining complexity is not limited by the intelligence of the machine's CNC controller, because data can be supplied directly from a larger computer
- tools can be automatically selected from a carousel
- workers are not confined to operate one machine, but can mind many CNC machines, however safety becomes an issue
- CNC machines can be made to communicate with other CNC machines and computer devices

Flexible Manufacturing Cells (FMCs) and Systems (FMSs)

Integrating CNC machine tools and material handling equipment with a supervisory computer, to make them co-operate and manufacture a family of parts according to CM, creates an FMC [Mair 1993]. Two facets of the integration are the manufacturing control and the communication of that control to the

machine tools and material handling equipment. The manufacturing control is run as software on the supervisory and controller computers at the same time (concurrently). The control is communicated between the supervisory and controller computers and on to the CNC machine tools and material handling equipment. Communication is handled by a computer network, which consists of computer nodes (supervisor and controller computers), cable and network software. FMCs can be fully automatic, and relegate workers to auxiliary tasks within the manufacturing environment, which can present a safety problem.

A co-ordinated collection of FMCs with inter-linking material handling makes up the FMS. Each family of components is manufactured in an FMC, and can be assembled with other components from other FMCs in a flexible assembly system (FAS) to form the products. The FMS can be a part of larger environments by integrating design with manufacture as in CADCAM, or integrating many corporate processes as in CIM (see section 0). An FMS can exhibit machine, operation, process, product, routing, volume, expansion and production flexibilities (see Table 2-1).

Team-working

A team of multi-disciplined labour works together in a group or cell [Gray 1994]. The team is empowered to make decisions necessary to maintain and improve high quality, flexibility and through-put. Teams are responsible for scheduling stock and meeting delivery times. Long-term education is planned by the team for its members.

Just-In-Time (JIT)

JIT [Voss 1988] concentrates on flow, flexibility and a chain of supply, with strategies of minimal stock and team-working. It adopts batch production, CM and FM, but develops a chain of co-operating suppliers. To facilitate the short through-put times and low work-in-progress, measures such as preventative maintenance, quality, team-working, inventory control and design for simplicity of manufacturing are generally implemented.

1.5 Shortening Time-to-market and Product Development Time

[BS 7000 1989] gives steps in the evolution of a product, and indicates the 'creation' processes which are the design and manufacture disciplines of product development. Statistics [Gould 1992] indicate that 60-80% of the product cost is pre-determined at the design stage, and that 50% of design time is spent on redesigns sent back by the manufacturing department because the design could not be made. Strategies to help shortening time-to-market (TTM) are concurrent engineering and design for manufacture:

Concurrent or Simultaneous Engineering

Many of the serial product development processes of [BS 7000 1989] can be done at the same time, or at least overlapped [Gould 1992]. A multi-disciplined team of engineers and marketers can follow the product from concept to launch, thereby improving the quality and speed of product development. Decisions made at the outset are no longer solely marketing decisions, because those whose interests are focused later in product development can anticipate problems and improvements, and can express them

in the feasibility study and specification. Also, late changes in manufacturing can immediately be assessed in terms of market response or design implications. The breakdown of barriers between corporate departments is necessary, including the integration of their computer systems into a CADCAM or CIM environment.

Design for Manufacture (DFM)

DFM implies that decisions made earlier in product development (in design) will influence those made later (in manufacture), so should be made with regard to them, or even in conjunction with them [Morgan 1988]. The design or manufacturing engineer is fully stretched by the demands of his or her discipline, so it is unlikely that an individual will be effective in both disciplines unless aided by an integrated computerised CAD and CAM tool. Here the designer could try a number of designs and the tool could quickly highlight or simulate the manufacturing, and possibly the production, consequences. This could preclude the building and testing of a physical model or prototype, and speed up the overall product development process.

1.6 Integrated Flexible Production Systems

Two integrated production environments that combine some of the above strategies for improving order-to-delivery and time-to-market are CIM and lean production:

Computer Integrated Manufacturing (CIM)

The use of computers can improve flexibility, processing, control and communication, as seen in FM and FMC above. Increases in flexibility and processing are evident in the application of computers in the manufacturing, design, production engineering, storage and transportation, and production planning departments in CAM, CAD, CAPE, CAST and CAPP systems [Yeoman 1985]. Information about components, machines, costs, personnel and other resources is common to many of these manufacturing business processes, and should not be held separately and so be replicated. CIM [Thompson 1987] is a corporate-wide information technology (IT) skeleton to service and share manufacturing information. Information entered via one computer system can be accessed by anyone in the company on another system. It does not have to be entered again, and everyone's information is up-to-date. Systems with different computer hardware, operating system and application software must be integrated so connections are invisible to the users. 'Open systems inter-connect' (OSI) [BS-ISO 7498 1984] is an international standard to assist in this connection.

Lean Production

This hybrid adopts many of the techniques to improve time-to-market and order-to-delivery time [Gray 1994]. Its aims are to make the most of flexibility, quality, human resources, working capital and floor-space. It therefore uses JIT inventory control, FMSs, quality control, team-working design for manufacture and concurrent engineering.

Production in the Car Industry

Some of the above strategies were developed by automotive producers [Gray 1994]. General Motors pioneered the Manufacturing Automotive Protocol (a specific form of OSI) and Toyota developed JIT. Lean production is being pursued by almost all of the world's major car companies to produce high quality and low cost products, in the high volumes and increasing variety demanded, with flexibility and shorter delivery times, and shorter time-to-market for new models. The result is that the average time to assemble a car has reduced by about 20% and TTM by about 25%.

1.7 Fast Reconfiguration Requirement in FMSs

Manufacturers who work to a TTM of a few months for a variety of components or products often have to change the floor layout or machine configuration of their FMS. Mostly the changes will be planned, but unpredicted alterations will also have to be made during normal operation, whilst the FMS is under real time control. A change in configuration could be:

- a machine replacement due to up-grade or maintenance
- an expansion of the FMS with a new FMC or more work handling, or of the FMC with a new machine or robot
- a cell replacement because of a new family of parts
- a re-routing due to machine breakdown or special component requirement

A conceptual reconfiguration or simulation must enable feasible alternatives to be developed, before the chosen configuration is implemented. Swift and dependable reconfiguration is needed for planned and unpredicted changes for safe, reliable and correct operation.

The planning, implementation and control of manufacturing strategies, and the manufacturing control, are largely the concern of the manufacturing engineers, while the communication of that control in an FMC is dealt with by the computer hardware and software engineers. Where speed of connection and re-connection of CNC machines and network computers is important, then a modular approach with common CNC and computer interfaces may be needed. In this way, once the configuration is decided, the implementation includes locating the appropriate machines and material handling, plugging together the network computers and plugging the machines and material handling into the network and loading the appropriate control software. This will enable the manufacturing engineer to manage the reconfiguration more quickly, and with reduced need for consultation with computer hardware and software engineers.

The transputer, "the computer on a chip" (see section 3.5.5), is a relatively cheap, reliable and fast microprocessor. It consists of processing, memory and communications, which makes it an ideal building block for use in networks. A variety of hardware is available to interface transputers with the CNC or PLC controllers of the machine tools and material handling equipment.

The fast reconfiguration requirement of FMSs is to improve TTM during planning, and to improve order-to-delivery time during unpredicted operation. By substituting the conventional computer network with a transputer based network, then the reconfiguration can be done more rapidly still.

The connection of hardware is trivial compared with the design of manufacturing control and communication software. The properties of transputers, and the programming languages that run on the transputer, e.g. occam, lessen both problems. However, there is a need for rapid development of software. This thesis describes a methodology, which enables the rapid development of flexible, comprehensible and dependable occam code.

1.8 The Research and Thesis

Research Evolution

The original research was to transputerise the University's FMC, and to formulate and carry out performance comparisons. Occam immediately was chosen as the transputer programming language, and general place-transition Petri nets were found to be a suitable modelling tool. A Petri net model of the FMC was constructed, and software was written to run on a single transputer. The formal characteristics of occam and Petri nets strongly influenced the change in research to a transputer controlled FMC. A relationship between occam and Petri nets was found that confirmed earlier work [Carpenter 1987]. Limitations in their inter-relationship and the difficulty in understanding large Petri nets, refer to the 'labyrinth' Petri net graph (inside the back cover), were the motivation for the methodology documented in this thesis.

Research Boundaries

FMSs are becoming more widely used, in different industries, containing various processes, producing larger output volumes in smaller batch sizes, so require reconfiguring more often and quickly. There is a need for a methodology that can help plan the manufacturing control and then help implement the communication of that control on a network of transputers and transputer-fronted CNC machine tools and material handling equipment.

The methodology is confined to the control and the communication of control for an FMC. It must:

- enable the modelling and design of synchronising and communicating concurrent manufacturing processes in an easily understandable way
- enable the production of correct, reliable and safe software
- allow the swift reconfiguration of machine tools and material handling for both planning and control
- facilitate future growth of, and modifications to, the cell
- handle, in real time, the variety and quantity of data communicated

The methodology exploits two mathematically based tools, Petri nets and occam, but its rules make no reference to mathematics. This should be attractive to industry, where there is a reluctance to adopt formal methods because of their complexity and difficulty of use. Consequently the research is non-mathematical, and the thesis only mentions mathematics in further work and appendices.

Thesis Structure

The thesis is divided in to three parts. Chapters 1 and 2 present the motivation, and example FMC, for the research. Chapters 3 and 4 include tools, techniques and issues that contribute to the development of the

methodology. Chapters 5 and 6 describe and discuss the methodology, and how it can be implemented in the example FMC. The remaining chapters highlight the work done, the work to be done and the conclusions.

The University's FMC is used as an example to illustrate the problems involved with an FMC. The execution of its function, the FMC's manufacturing control, and its means of distributing the function, the communication of control, are detailed in chapter 2. The safe, reliable and correct operation of an FMC can only be achieved if the constituent hardware and software have been developed to operate reliably and correctly. Chapter 3 discusses these issues, and compares alternative suitable tools and techniques, and examines the chosen modelling and programming tools, Petri nets and occam. Software engineering for DCSs is discussed in chapter 4, focusing on comprehensibility throughout development.

Chapter 4 ends by summarising work from it and previous chapters in the conclusions. Chapter 5 begins by defining the aims of the methodology, which are derived from the conclusions. Techniques and considerations to achieve the aims of the methodology are discussed, and are specified as the goals and strategies of the methodology. The methodology is then defined in terms of steps, tasks and examples. Chapter 6 discusses the methodology through example and by comparing it against its aims. The thesis ends with conclusions and recommendations for further work. Because of the novelty in the research, there are many recommendations.

The overall Petri net graphs before and after application of the methodology to the FMC are located inside the back cover. The occam code is presented in the appendices.

Formats used in the thesis are as follows:

- quotes from referenced work are in speech marks
- new or highlighted words or phrases are in quotation marks
- square braces indicate this author's additions, and will not be confused with bibliographical referencing
- names in Petri net graphs are in arial typeface
- occam names are in `courier` and occam reserved words are upper case

2. The Flexible Manufacturing Cell and Its Communication

2.1 Introduction

This chapter examines relevant features of FMCs, describes the School of Engineering's FMC, how the FMC was built, its flexibilities and the type of component which the FMC manufactures. It discusses the control and communication of control of the FMC, and highlights the problems encountered by PC based control and communication of control, and ends by summarising the manufacturing requirements.

2.2 Flexibility and Cellular Manufacturing

As indicated in section 1.4, there are several techniques to improve manufacturing lead times. Many of these rely on the flexibility of labour, machine tools, materials handling, their configuration and their control. In this section, flexibility and cellular manufacturing are examined further in terms of FMSs, and the test FMC (refer to section 2.3) is classified into a specific type of FMS.

Cellular manufacturing [Greene 1985] came from and is a specialisation of 'group technology'. Group technology is the "bringing together and organising of common concepts, principles, problems and tasks to improve productivity"; cellular manufacturing is the "physical division of the manufacturing facilities machinery into production cells"; "each cell is designed to produce a part family"; and a part family is a "set of parts that require similar machinery, tooling, machine operations, and/or jigs and fixtures".

Computer controlled manufacturing equipment (machine tools, work handling, sensors and actuators) can be made to co-operate under the direction of the manufacturing control software. Selecting equipment with appropriate flexibility and deploying it according to a flexible manufacturing strategy will result in an FMS. Eight categories of flexibility have been identified [Browne 1985] for FMSs, refer to Table **2-1**.

Flexibility	Description
Machine	The ease, management and sophistication of change of tools, fixtures and part programs of the machines.
Process	The different processes available.
Product	The ease and speed of changing to a new parts family.
Operation	The reordering of the sequence of operations
Routing	The different routes a component may take, due to machine replication or break-down.
Volume	The range of profitable batch sizes.
Expansion	The ease and speed of changing number and type of machine or handling equipment.
Production	The spectrum of operations available in the FMC.
Machine flexibility is necessary for process, product and operation flexibilities, and routing flexibility is necessary for volume and expansion flexibilities. Production flexibility depends on all the other flexibilities.	

Table 2-1 Flexibilities of FMSs

An FMC can be viewed as a flexible implementation of cellular manufacturing incorporating computerised machine tools and work handling, or viewed as a subset of FMSs. Control in cellular manufacturing [Greene 1985] is divided into cell loading and cell scheduling. Cell loading allocates

components to cells. Cell scheduling addresses the internal control of jobs within a cell, and is the "determination of the order of the components onto each machine and the determination of the precise start time and completion time of each job on each machine". In practice however, "most viable control schemes do not perform cell scheduling but rather employ cell sequencing". "Sequencing is limited to the determination of the order of the jobs onto each machine, and does not address timing."

An FMS [Browne 1985] is a generic term for "an integrated, computer controlled complex of automated material handling devices and numerically controlled machine tools that can simultaneously process medium sized volumes of a variety of parts' types". Four types of FMS have been identified, and are summarised in Table 2-2. The test FMC (see section 2.3) has two machine tools, so is not a flexible machining cell; its conveyor is fixed but enables route variations, so is not a flexible transfer line or multi-line. It is closest to a flexible machining system, but is not as sophisticated.

Type	Flexible:	Description	Route
I	machining cell	CNC machine tool, automated materials handling, input and output buffers	single
II	machining system	several different type I's, extra automated materials handling	many
III	transfer line	fixed route through a line of different type I's	fixed
IV	transfer multi-line	several type III's	many fixed

Table 2-2 Four types of FMS and their routing flexibilities

2.3 Description of Existing PC Based Cell

It is necessary to discuss the FMC for the following reasons:

- The FMC is the chosen example system for the development and validation of the methodology
- The issues highlighted during the design of the system provided the motivation for the methodology

The FMC includes a CNC controlled Beaver 2 ½ axis milling machine, a MHP CNC lathe and a six axis vertically articulated Puma robot (the cell robot). The design of the FMC had to allow the lathe, miller and robot to be used separately and used integrally in the FMC, hence the layout of the machine tools of Figure 2-1 and Plate 1 on p18. Other considerations were: to choose a suitable component to be machined, which utilised the lathe and miller (i.e. chess pieces); to distribute that component physically from the raw material store to the lathe and miller and on to the finished component store (the conveyor track and transfer devices); and to distribute control to the lathe and miller and any necessary work handling (i.e. the local area network, LAN).

2.3.1 Choice of Component

The choice of component to be manufactured depended on the following criteria. The component had to:

- be able to be manufactured within the capabilities of the miller and lathe
- facilitate the illustration of flexibility of the FMC

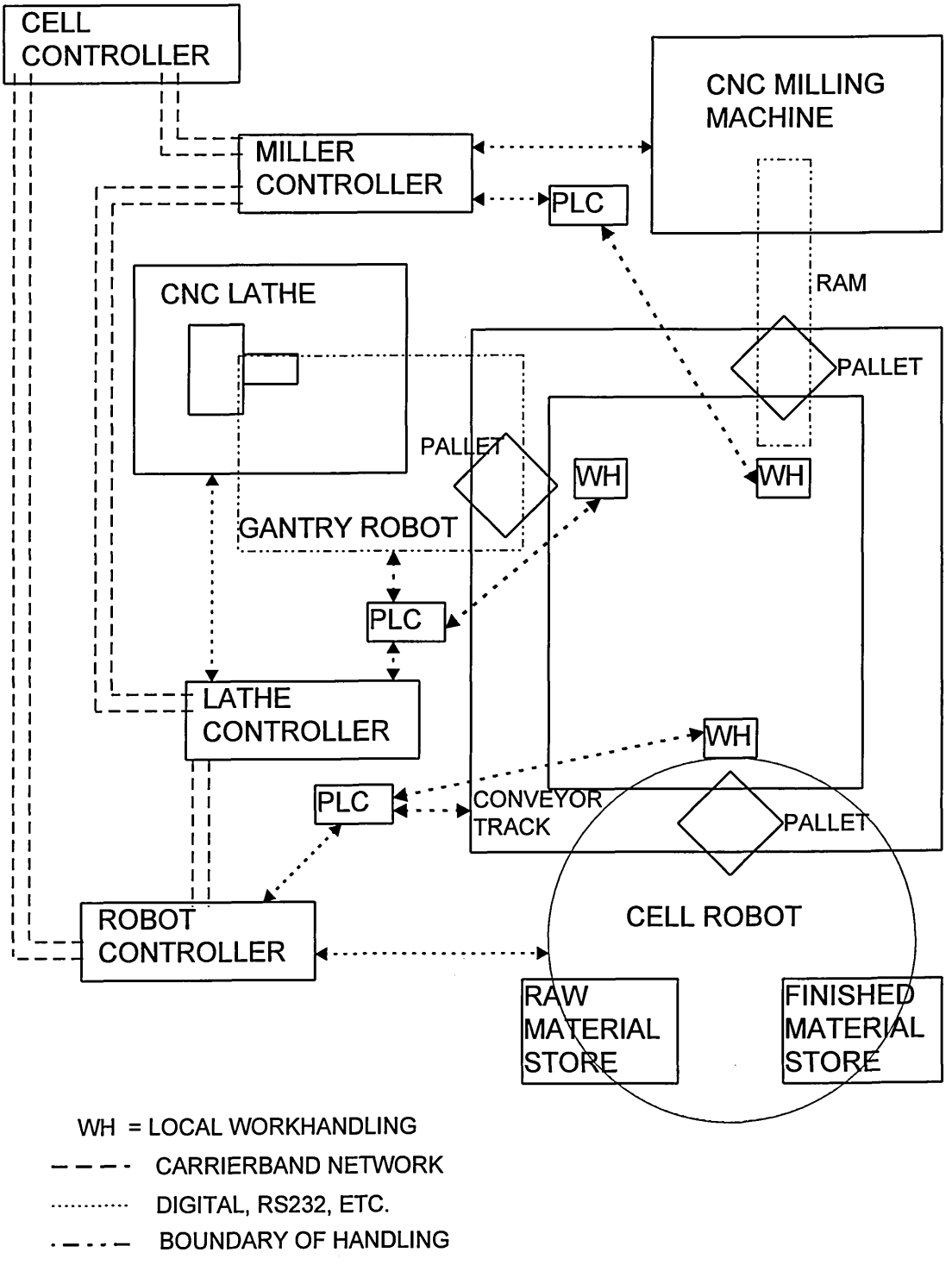


Figure 2-1 The layout of the FMC showing cell robot, lathe and miller workstations around the conveyor, and their controllers connected by the network

Chess pieces designed for blind players were chosen as example components (see Figure 2-2), because pieces and 'colours' have to be different to the touch; which provides the following manufacturing implications:

- A chess set consists of 8 pawns, 2 rooks, 2 bishops, 2 knights, a queen and king; and is machined for each 'colour' (which illustrate machine flexibility, see Table 2-1)
- Chess sets can be made of different dimensions or styles (product flexibility)
- Some components, for example pawns, need only be turned, so will not have to be loaded into the miller (routing flexibility)

Table 2-3 presents a translator from 'chess piece to code to part program' for subsequent job list production. For example, if one white size 4 king is needed, then the code 1K4W generates part number 1 in the job list of Table 2-4. If, also, one black size 3 pawn and two white size 2 rooks are wanted, then codes 1P3B and 2R2W produce three more parts for the job list. LK(S,C) is the lathe part program to produce a king of size S (1 to 5) and colour C (black or white).

CHESS PIECE	Size (1-5)	Colour (B or W)	PART PROGRAM Lathe	Miller
King			LK(S,C)	MK(S,C)
Queen			LQ(S,C)	MQ(S,C)
Rook			LR(S,C)	MR(S,C)
Bishop			LB(S,C)	MB(S,C)
kNight			LN(S,C)	MN(S,C)
Pawn			LP(S,C)	No milling
Colour			Generate lists of lathe and/or	
All		(B & W)	miller part programs	

Table 2-3 Chess piece codes translates in to part program

JOB NUMBER	LATHE	MILLER
1	LK(4,W)	MK(4,W)
2	LP(3,B)	
3	LR(2,W)	MR(2,W)
4	LR(2,W)	MR(2,W)

Table 2-4 Simple job list for the FMC generated from the codes of Table 2-3

2.3.2 Conveyor Track

The conveyor track physically distributes components to and between machine tools. It is rectangular in shape, refer to Figure 2-1 and Plate 2. There are three identical pallets on the track and no sidings or buffers, for simplicity. There is one pallet per station, which provides a simple control law. The conveyor track runs continually, so dogs are used to halt the pallets at the appropriate position in front of the stations. When the pallets are to be transported, the dogs are opened and are made to close once the pallets have passed the dogs, and is termed 'indexing'. This means, for a three pallet and three station FMC, there is only one pallet behind each dog at any time, no queues and no vacant stations occur.

One feature of a conveyor track system with no queues and no buffers is that transportation between stations and transfers within stations can not happen at the same time. This synchronisation greatly

influences the control law. It can be noted at this stage that indexing and transferring are sequential for this FMC, which can be significant in lead times for components which require short machining operations.

2.3.3 Pallets

The purpose of the pallets is to hold the components while they are transported by the conveyor track between stations. Pallets consist of a base, which is made to suit the conveyor track, and a detachable vice, refer to Figure 2-3 and Plate 3.The jaws of the vice are 'v' shaped to accommodate various diameters of component (product flexibility), and are opened and closed by a pneumatic toggle-clamp, which is bolted to the vice. The vice is only detached at the miller, and this operation is also activated pneumatically. The air supply to open the vice on the pallet and most of the pneumatics is part of the local work handling equipment, described next.

2.3.4 Transfer and Local Work Handling Equipment

The components are transferred between a) miller and conveyor, b) lathe and conveyor, and c) raw and finished component stores and conveyor. Each of the three stations contains local work handling to perform transfers, which include 'transfer devices' and auxiliary equipment, refer to Figure 2-4. The work handling is different at each station for teaching purposes.

In the case of the **robot** workstation, the cell robot performs the majority of the local work handling. Here, the component is loaded by the robot from the raw material store to the pallet, and when all machining of the component is complete the component is unloaded from the pallet and onto the finished component store, refer to Figure 2-5 and Plate 4. The stores hold the components in the correct order for machining and in the correct orientation to allow the robot to pick up and return the component easily. To load a pallet (ignoring the downloading of part programs), the robot moves to the raw material store, grips and picks up the component, moves to the empty pallet, puts the component in the (already opened) jaws of the vice attached to the pallet, the jaws close gripping the component, and the robot releases the component and moves away. To unload a pallet, the operations for loading are reversed, but the component ends up in the finished material store.

The type of work handling employed to service the **lathe** is a pneumatically operated pick-and-place gantry robot, refer to Figure 2-6 and Plate 5. To load the lathe, the gantry robot clasps the component whilst the component is still in the jaws of the vice on the pallet, then the vice on the pallet is made to open before the gantry robot carries the component to the lathe's chuck. The transfer of component between gantry robot and lathe chuck requires the component to be held by the lathe's chuck before the gantry robot will release the component and moves away. A reverse sequence of movements is needed to unload the component from the lathe to the empty pallet.

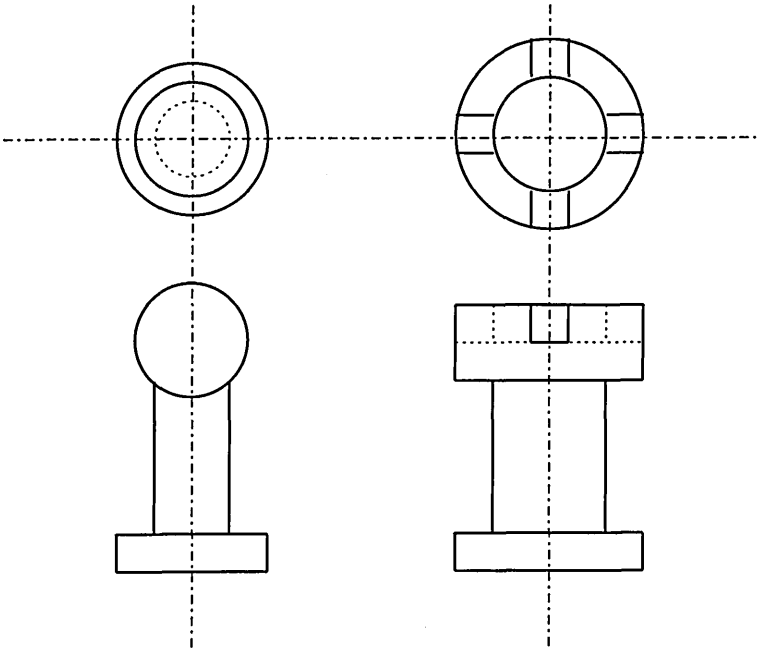


Figure 2-2 Two typical parts, one for turning only, one for turning and milling

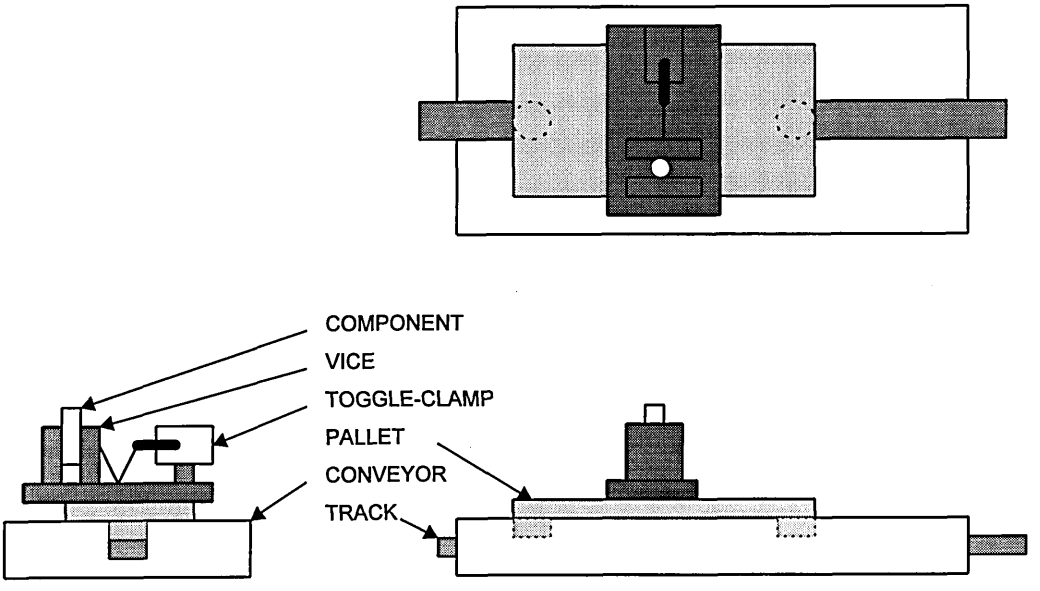


Figure 2-3 Three views of a component, vice, pallet and conveyor track

The transfer of component between conveyor track and **miller** includes the transfer of the vice from the pallet to a fixture on the milling table. This has an influence in the control law, because some pallets on the conveyor track will have vices, and those that have deposited a component at the miller will not. The transfer of the vice precludes the need for changes of fixtures on the miller. In the transfer operation, the vice is pushed off the pallet and on to the machining table of the miller by a pneumatic ram, refer to Figure 2-7 and Plate 6.The position of the vice on the miller's table is therefore dependent on the length of the ram. The control law to transfer the component (and vice) from pallet to miller, and from miller to pallet, is much simpler than that of the lathe.

Other facilities common to more than one workstation are:

- a compressed-air supply to the pallets to enable the vice jaws to open and close (local to the lathe and cell robot)
- electrically operated dogs- to halt the pallets in front of the workstation and end indexing
- proximity sensors- to detect the presence of the pallets in front of the dogs

This completes the physical and functional description of the example FMC. The other important factors are the manufacturing control and the communication of that control to the machine tools and work handling equipment.

2.4 Control and Communication of Control

The control and the communication of control in a CIM environment can be represented in separate but related hierarchical structures [Weston 1991].

In the control hierarchy, enterprise decisions are made least frequently, are of greatest consequence, and appear at the top of the structure, refer to Figure 2-8a. Shop floor operations (e.g. sundry purchases or actuator movement) are made most frequently, are of smallest consequence, and appear at the bottom level of the structure. Decisions are, or control is, so important to manufacturing enterprises that departments, sections and offices are usually created, and people are employed, to reflect the structure of the control. In the hierarchical model, each level makes decisions and controls the levels beneath. For example, actuators, sensors and their controllers are made to co-operate and work for the workstation controller; and in turn the workstations and their controllers are made to co-operate and work for the cell controller. This strategy is only possible if the sub-systems are modular, where modules in the lowest level respond to commands from modules in the level above; and modules in other levels respond to commands from modules in the level above, and issue appropriate commands to modules below.

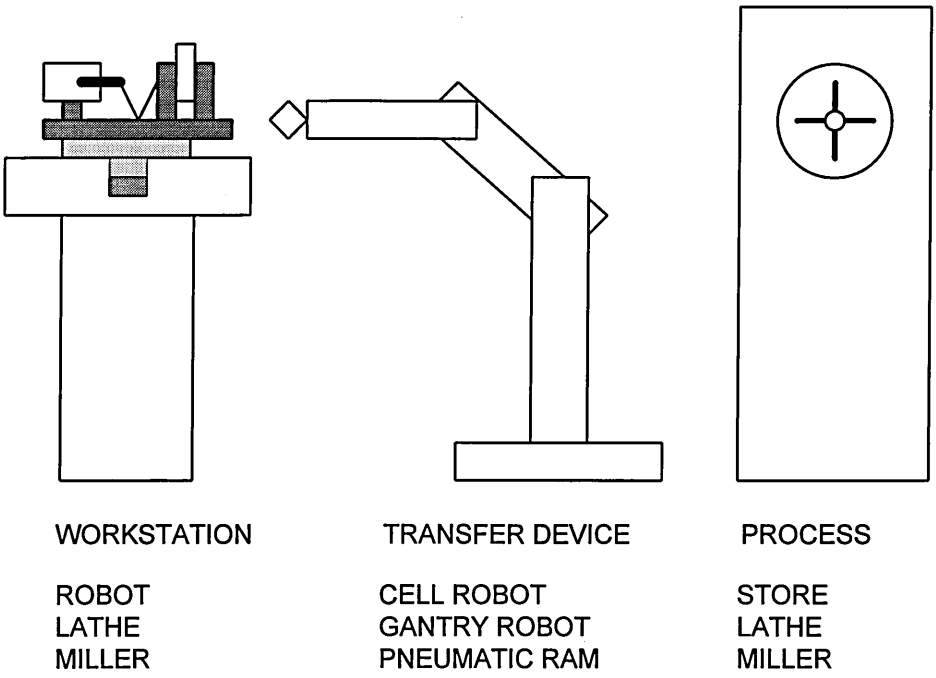


Figure 2-4 Workstation modules consist of process, transfer device and local work handling

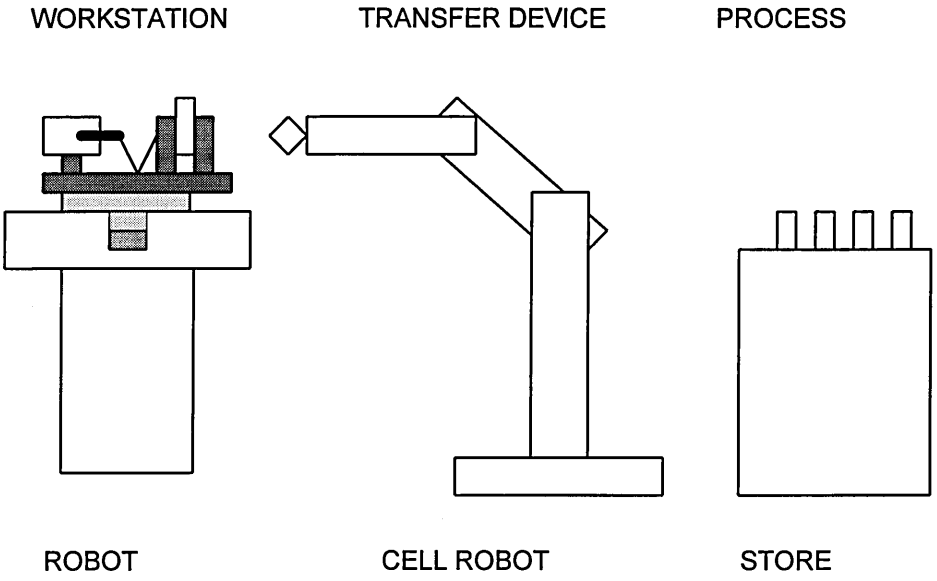


Figure 2-5 Robot workstation consisting of cell robot, raw material and finished stores and local work handling

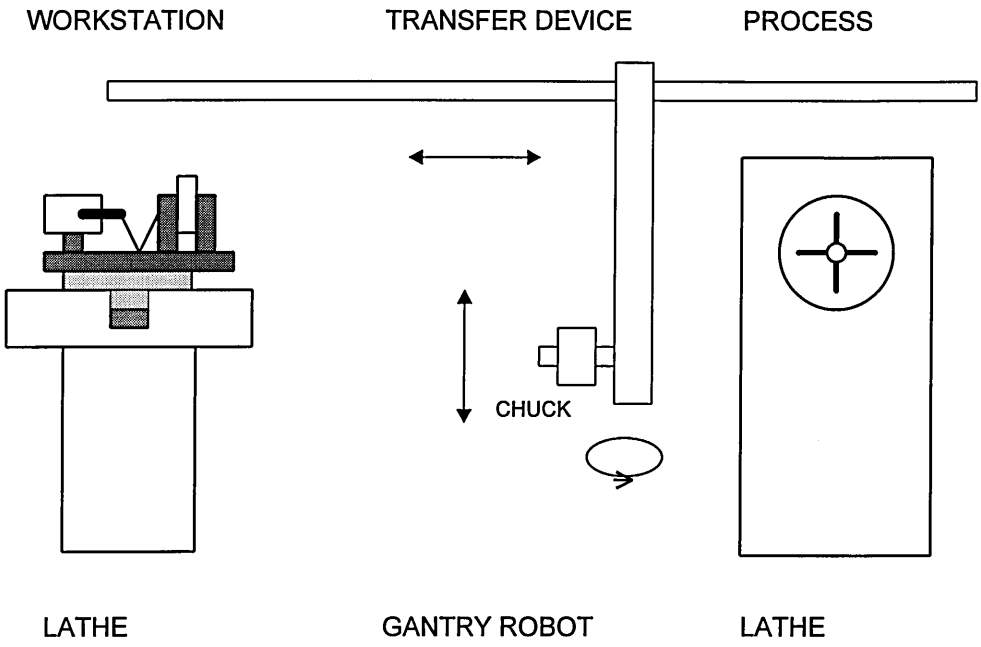


Figure 2-6 Lathe workstation consists of lathe, gantry robot and local work handling

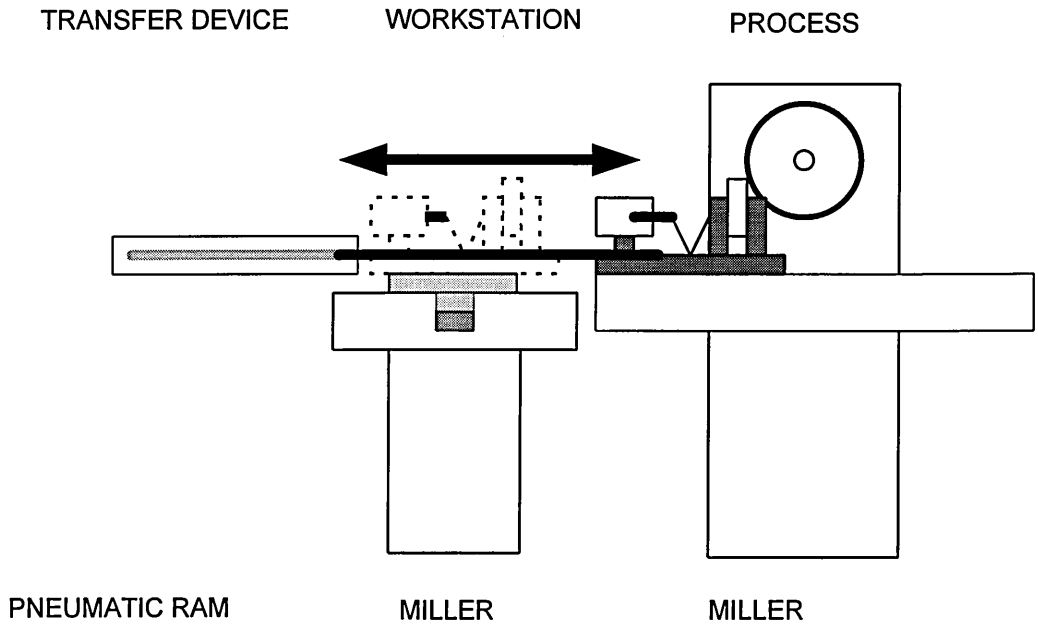


Figure 2-7 Miller workstation consists of miller, pneumatic ram and local work handling

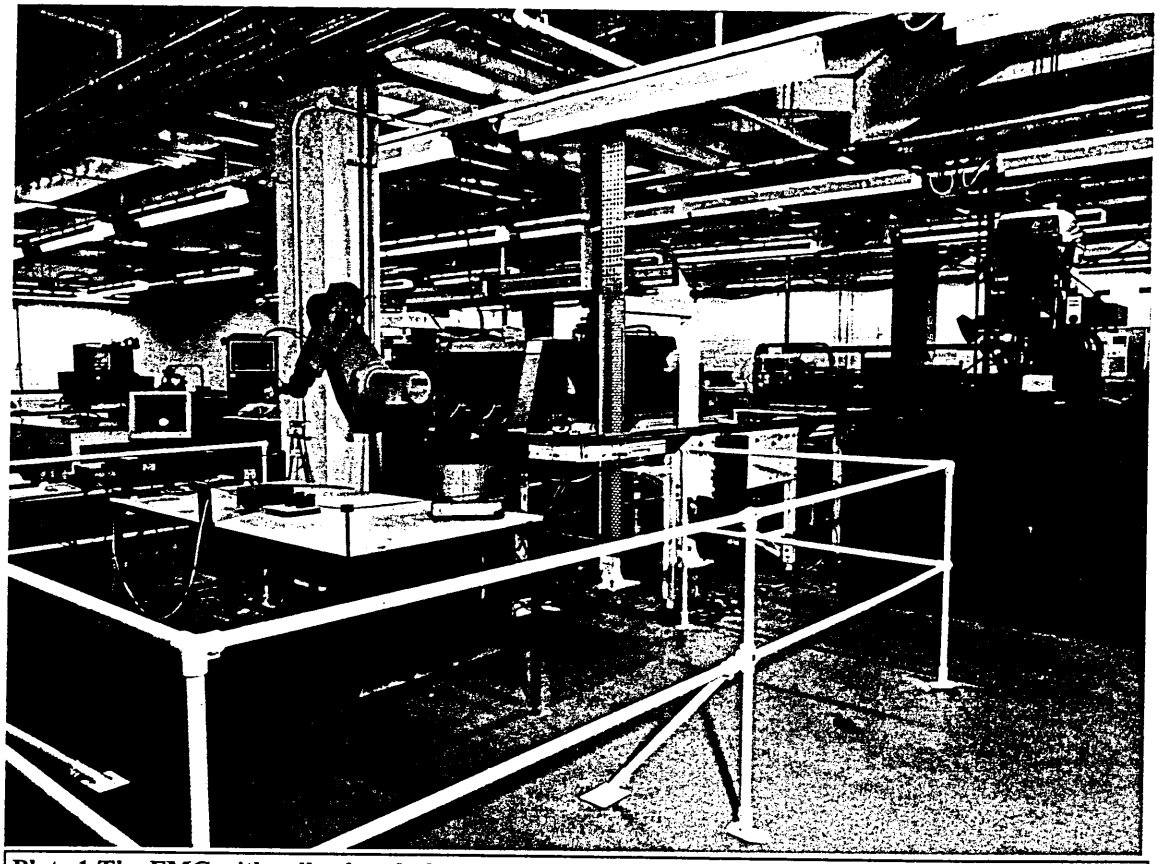


Plate 1 The FMC with cell robot, lathe and miller workstations

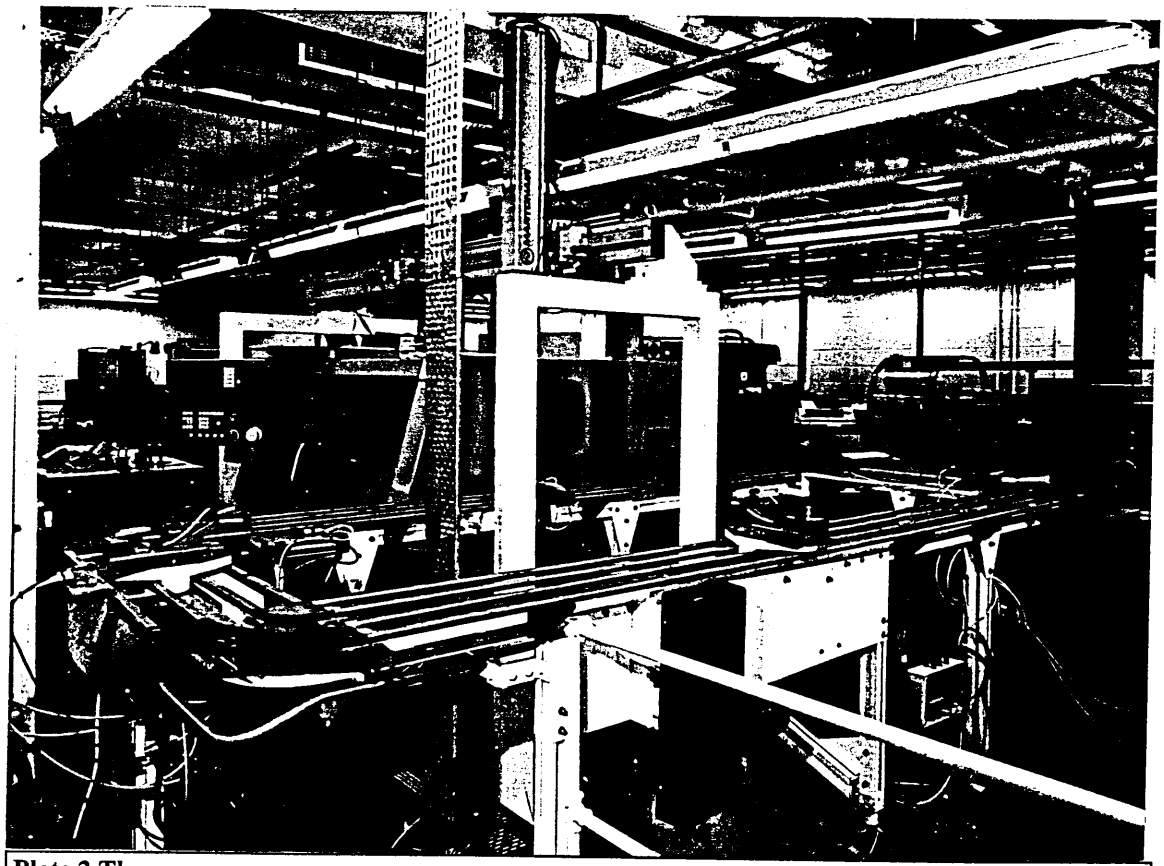


Plate 2 The conveyor

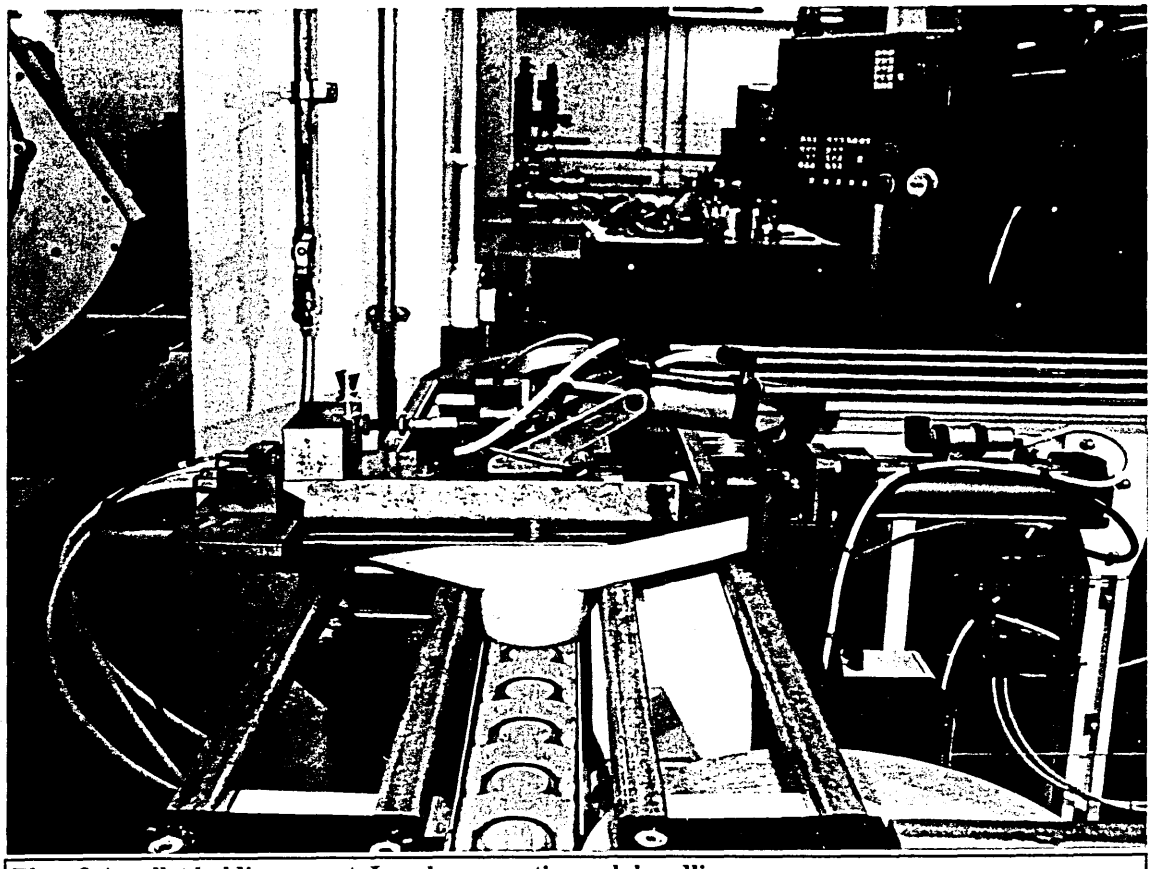


Plate 3 A pallet holding a part. Local pneumatic work handling

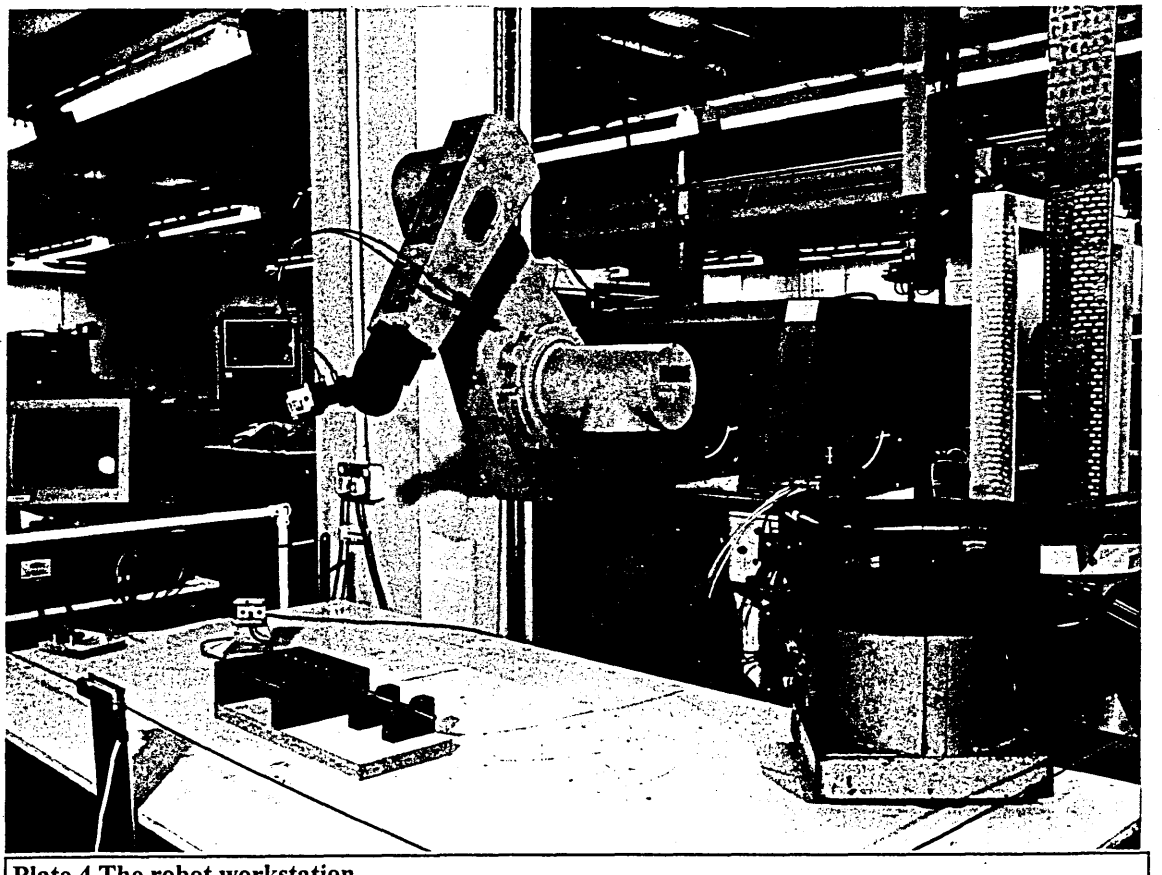


Plate 4 The robot workstation

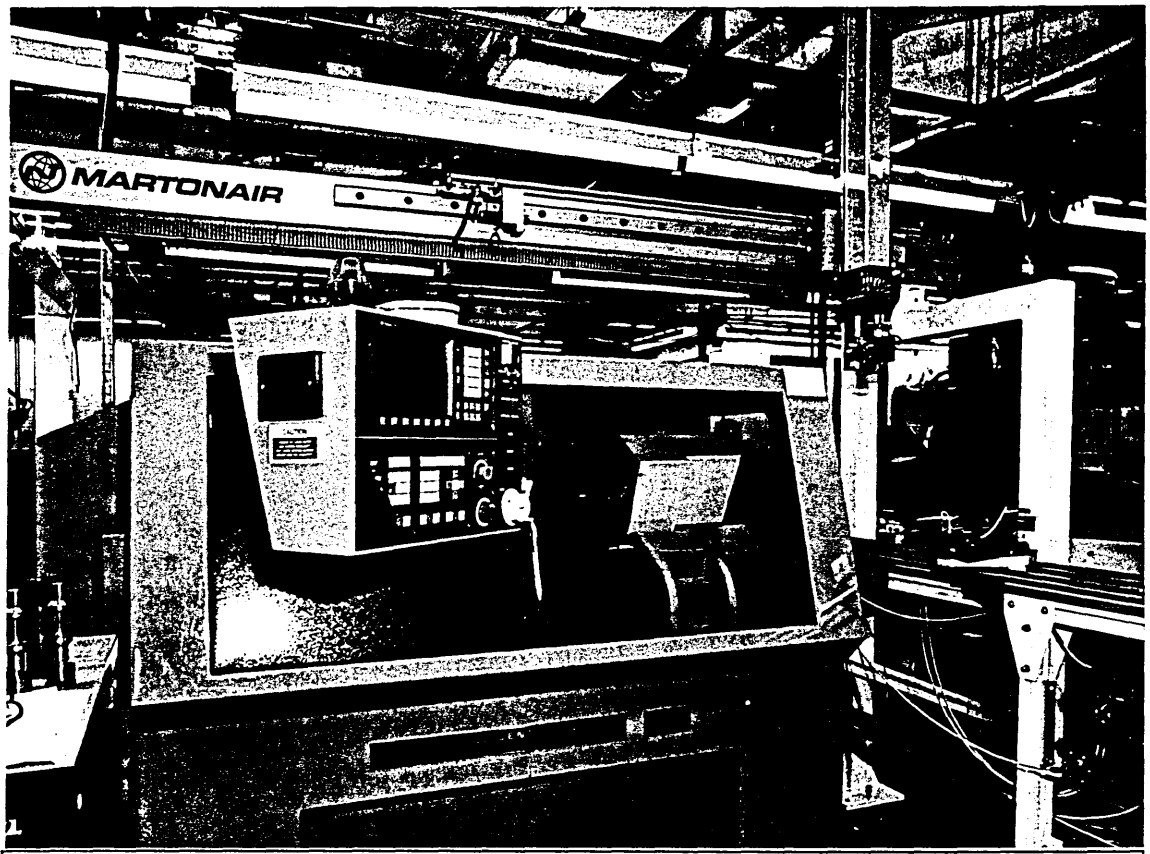


Plate 5 The lathe workstation with lathe and gantry robot

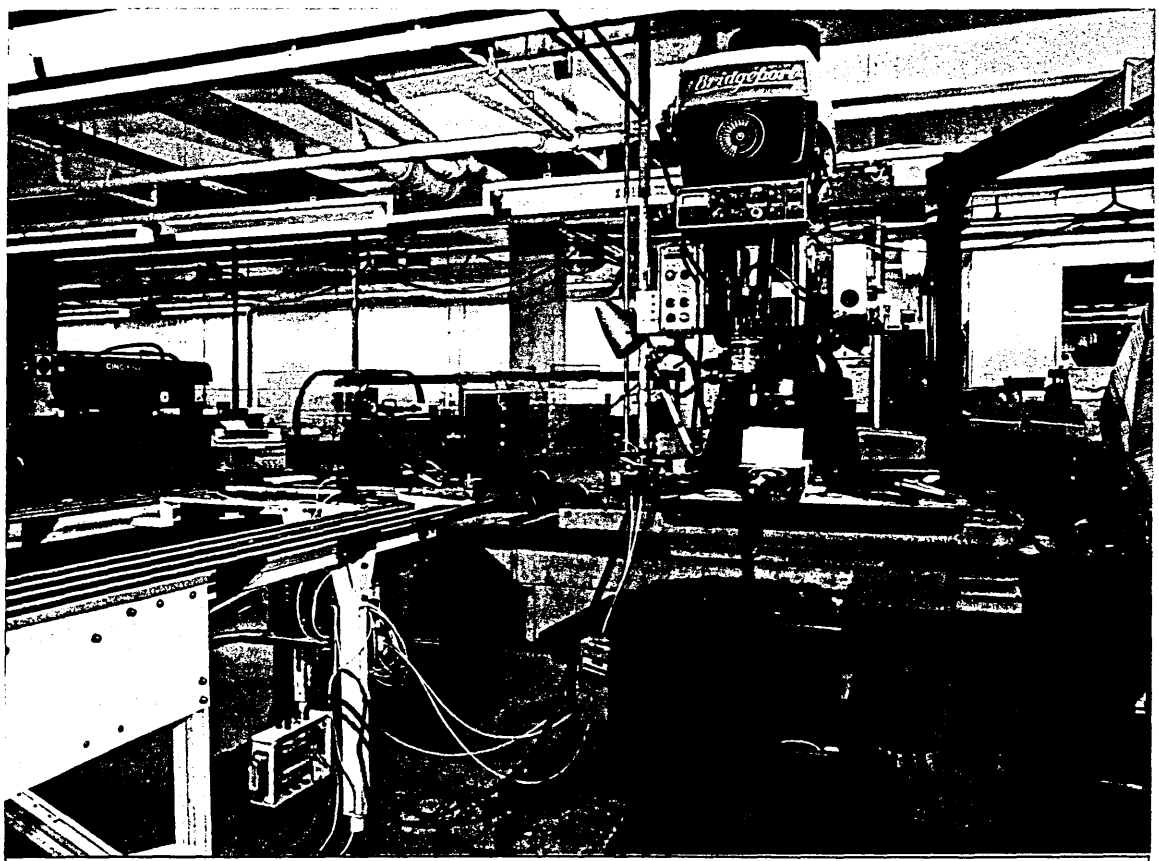


Plate 6 The miller workstation with miller and pneumatic ram

The hierarchy for the communication of control is less formal than the control hierarchy, and has fewer tiers, but it can be arranged in such a way that the layers of communication relate to the levels of control, refer to Figure 2-8b. The structure of the hierarchy is more dependent on the communication hardware and the physical location of the decision creators or users, rather than on the nature or grouping of decisions. Issues such as data size, data dependability and data transmission speeds depend on both the nature of decisions and hardware considerations, but because they are implemented in hardware they appear in the communication of control hierarchy.

2.5 The Control of the Existing PC Based FMC

The control of the School's FMC is hierarchical and follows the standard one described above for cell, workstation and equipment control levels. The FMC is isolated, and is provided with a simple sequence of jobs, i.e. the job list. At the highest of the three levels, level 3, is the cell controller, or supervisor. The cell controller determines what tasks should happen at, and records the responses from, the second level workstation controllers.

There are three workstation controllers, one each for the lathe, the miller and a controller for the robot and conveyor track. The robot and conveyor are combined in one workstation because the control for the conveyor is trivial, due to the fact that the pallets must not be transported at the same time as the robot is loading or unloading a pallet and to save on resources. The level 2 controllers manage the contents of their workstation, so issue instructions and part programs to, and receive responses from, the level 1 controllers. The level 1 controllers are either the work handling PLCs or the controllers in the CNC machine tools or cell robot. The lathe workstation consists of a CNC lathe, a PLC controlled gantry robot and local work handling equipment (also controlled via the gantry robot's PLC). The miller workstation consists of a CNC miller, and PLC controlled local work handling equipment, which includes a pneumatic ram (to transfer the vice and component).

To illustrate the control, consider the case of loading the lathe with a component which is in the waiting pallet. From the current status (the lathe is idle, the gantry robot is idle, the pallet is at the lathe and is full, the part on pallet requires turning), the control program at the cell controller determines to load the lathe, so the command is sent to the lathe controller *'load lathe'*. The lathe controller must then synchronise operations between the local work handling and the gantry robot and between the gantry robot and the lathe. The following instructions (ignoring down-loading of the part program) are sent by the lathe controller to the CNC lathe and PLC controlling the gantry robot and the local work handling:

- gantry robot to move to and grip the component which is in the pallet vice
- local work handling to open the pallet vice
- gantry robot to move to and to hold the component in the jaws of the lathe's chuck
- lathe to grip the component in its chuck
- gantry robot to release the component and to move away

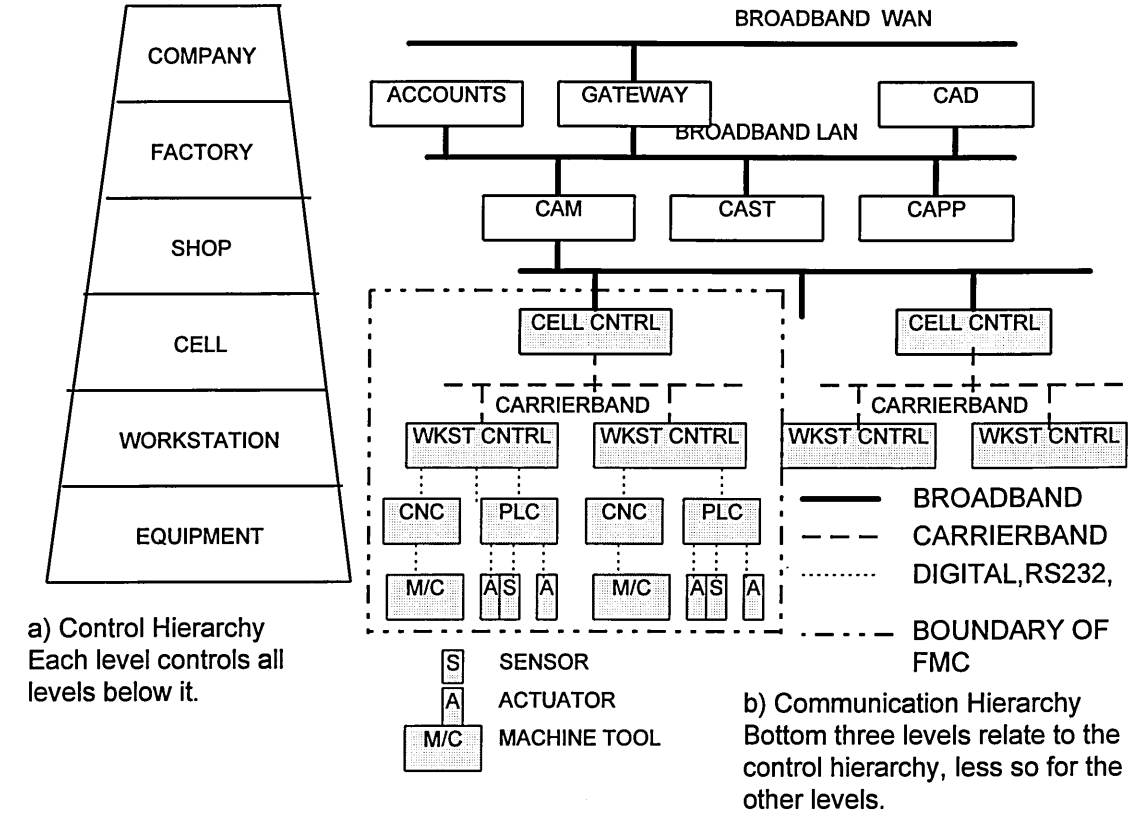


Figure 2-8 Corporate CIM hierarchies for a) control and b) communication

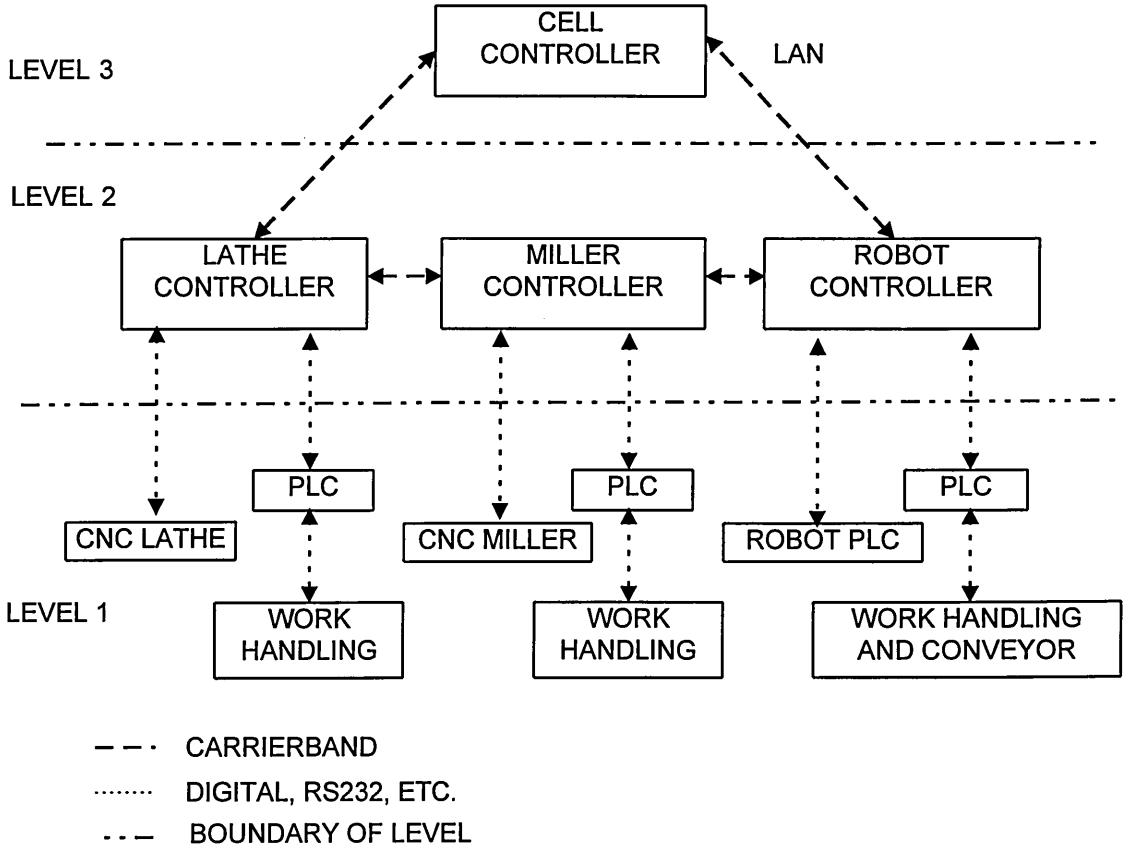


Figure 2-9 A combined control and communication hierarchy of the FMC

After each of the 5 operations is complete, the level 1 controllers (the lathe controller, and the combined gantry robot and local work handling controller) respond to the level 2 controller that an operation has ended. When all five operations are finished then the lathe workstation controller informs the cell controller that *'load lathe'* is complete, then statuses in the cell controller are updated. Errors are handled in a similar way. If an error occurs in any of the five loading operations at level 1, then this will also manifest as an error in the *'load lathe'* instruction at level 2, which in turn will be made known to the cell controller.

2.6 The Communication of the Existing FMC

Most of the control is physically distributed via a '9Tiles' Register Insertion Buffer Ring LAN [9Tiles]. The cell and workstation controllers are Intel 80286 based PCs linked in a ring, refer to Figure 2-10; PLCs monitor and control the work handling; the PCs communicate directly with the CNC lathe, CNC miller and PLCs which control the cell robot, dogs and sensors, conveyor track and local work handling. These are described in detail below.

The LAN communicates between the level 3 cell controller and level 2 workstation controllers, and must be able to cope with messages which are issued simultaneously from any or all of the other controllers. It must also be able to transmit information in the form of commands, files, errors and acknowledgements.

The mode of communication between the level 2 workstation controllers and level 1 manufacturing equipment PLC controllers is point-to-point. The protocols are as follows:

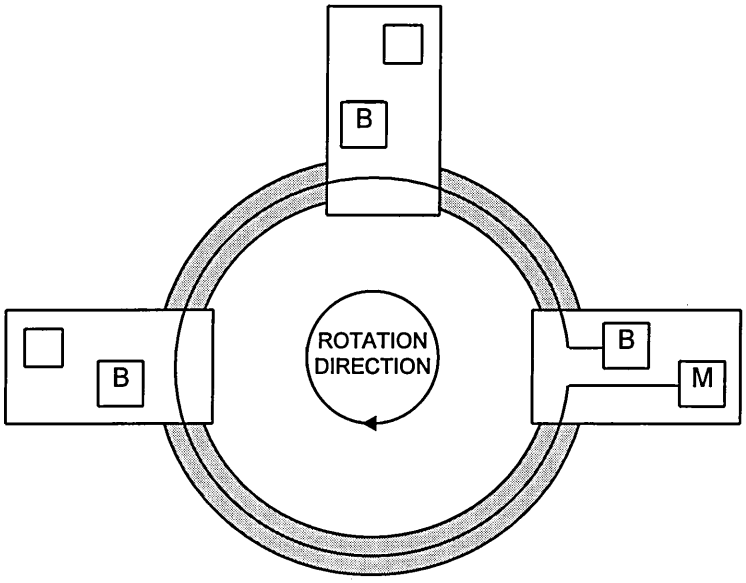
- RS232 PC to lathe, PC to miller
- digital I/O 24 volt PC to PLC

The mode of communication between the level 1 PLC controllers and the manufacturing equipment is either:

- internal: lathe, miller, cell robot
- digital I/O 24 volt

The actions of the manufacturing equipment controlled by a workstation are not all sequential, so interaction occurs at the end of operations. The absence of communication between concurrent processes at level 1 precludes the need for a sophisticated LAN. To emphasise the synchronisation in the point-to-point communication employed, then consider again the *'load lathe'* instruction. The cell controller issues the instruction to the lathe workstation controller via the LAN. The lathe workstation controller must synchronise the operations of the manufacturing equipment controllers to transfer the component at the pallet to the lathe's chuck by the gantry robot:

- the local work handling opens the pallet's vice once the gantry robot is holding the part
- the gantry robot can release the part once the lathe's chuck is holding the part; so issues instructions according to the five control operations, above



TO SEND, ANY OTHER TRANSMISSION IS BUFFERED IN 'B', AND
THE MESSAGE 'M' IS INSERTED IN ITS PLACE

Figure 2-10 Buffer insertion ring network with 3 nodes, one transmitting

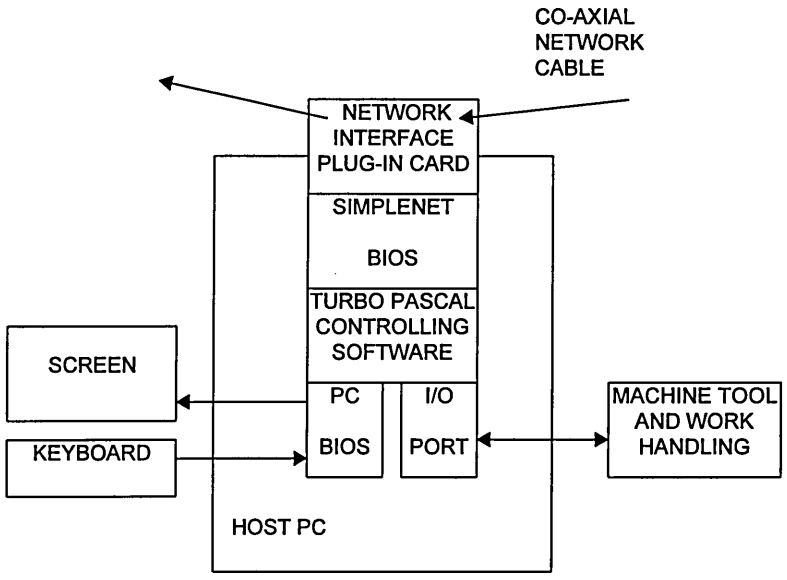


Figure 2-11 The ports of a PC node: network, workstation, screen and keyboard

The main functions of the LAN and the PCs are as follows:

The cell computer is the level 3 controller executes control programs, communication software and:

- issues instructions to the workstation PCs determined by the control programs
- receives messages (statuses and errors) from the workstation PCs
- stores the statuses of the cell (machine tools, work handling, pallets and components)
- starts and assesses the status of the LAN
- would be the interface to a larger manufacturing environment

The workstation PCs are the level 2 controllers, and have the following functions:

- receive instructions from, and inform completion of operations to, the cell controller
- send instruction to, and receive responses from, the controllers of the machine tools and work handling (including cell robot)
- store relevant part programs for machine tools and work handling (including cell robot), and load them when directed by the cell controller
- synchronise operations between local work handling, transfer devices and machine tools
- report errors to the cell controller

2.6.1 The LAN and PC Hardware and Software

The cell and workstation controllers are all connected via a LAN. The function of the LAN is to convey instructions and data from one PC to another. The LAN package includes: IBM plug-in cards, which fit in one of the expansion slots of the PCs; software, which is run by the PCs and the plug-in cards; and cables.

Each PC has a 286 micro-processor, 640 Kbyte RAM, 20 Mbyte permanent storage and a mother-board bus. The bus provides access to the expansion slots and on to the LAN's plug-in cards. The instructions and data are held in RAM, and the micro-processor runs the control code to determine what instructions or data will be sent to which workstation controller. Typically PCs communicate: over the LAN via the plug-in cards; with the machine tools via RS232; and with work handling PLCs via digital communication, refer to Figure 2-11.

The function of the LAN hardware, in the FMC, is to convey instructions or data between the PC controllers of the cell, lathe, miller and robot stations. The LAN card is designed to fit in one of the PC's expansion slots, so one of the card's functions is to convert from parallel signals of the expansion slot into serial signals for the LAN. It must cope with incoming messages from the LAN and the manufacturing control program at the same time. At the heart of the LAN card is a Z80 micro-processor. It performs the conversion, performs signal integrity checks, runs the software of the LAN's operating system and handles simultaneous messages. The Z80 chip also controls the LAN's 'buffer insertion' 'media access control' (MAC).

2.6.2 LAN Software

The operating system is a set of programs that sit between the user application software (written in Pascal, described below) and the software that controls the network hardware. SimpleNet is the name of

the operating system, and SimpleNetBIOS (SNBIOS) is the 'basic input output system (BIOS)' which controls the hardware. SimpleNet can interface with industry standard NetBIOS, which supports the 286 PC. SimpleNetBIOS is a set of network routines that permits software running on a PC to access a Superlink plug in card. However it is also possible for user software to interface directly to the Superlink card.

Borland's Turbo Pascal was chosen as the programming language to write the manufacturing control code and the communication routines. The manufacturing control is not trivial, so requires the capabilities of a general purpose high level programming language when writing the code. ISO Pascal is a strongly typed, modular and compilable language with records, procedures, functions, data hiding and some library routines. Turbo Pascal version 4 provides additional features including a programming environment consisting of integrated editor, debugger, compiler and contextual help. The communication routines need all of the capabilities, and most of Turbo Pascal's additional features, including various compilation levels and extensive library routines such as access to registers and interrupts.

A service, such as sending data, requires five calls to SimpleNetBIOS:

- open a channel in the sending node (between PC and plug-in card node)
- onward connect to the receiving node
- onward connect to the service, so the receiving node knows what is happening
- transmit the data
- close the channel

These five calls must be written in the application control program. Each service call from the application program is of similar construction and includes: LAN number, response, service command, destination address, length of message and message. The message and length of message are only used for the transmit service, but all services must include the LAN number, service command and destination address.

The service commands (open channel, onward connect, transmit data, receive data, close channel, offer service and end service) can either wait for the LAN card to respond to the application program before execution or be executed as soon as possible. It was found that waiting for the response gave reassurance that the node understood the service command, and gave better evidence of when a message was sent. However, the response required capturing, and a malfunction in the receiving node will prevent a response thus rendering the sending node inoperative.

The service call construction is written in Pascal as records. Procedures are code modules, and can be made into interrupt routines by specifying INTERRUPT at the beginning of their definition. When the interrupt routine is called, the process descriptor (local variable work space and instruction pointer) is saved, and is restored when the routine is complete. This enables pre-emptive communication, with the manufacturing control running in the foreground and the communication running in the background. The interrupt routine, to communicate with the LAN, requires access to the LAN card. This is done by calling

the library procedure INT($5C, register), where $5C is the PC's address for NetBIOS, and the register consists of the segment and offset of the service call's interrupt routine.

To implement sending data with capturing the response requires five issuing and five response interruptions to the execution of the manufacturing control program. The cell controller can receive information from any other controller at any time, so the LAN card has to cope with simultaneous incoming LAN messages, outgoing LAN messages, incoming PC interrupts and responses to the PC. There is uncertainty regarding how this operates and the latencies involved. The code required to enable communication was found to be a significant portion of the program when compared with the manufacturing control code. At the end, it was found to be a significant portion of the code. This led to difficulties in reading the code, which led to coding errors. Software maintenance was fraught and off-putting.

9Tiles' Superlink supports four classes of user:

- Dumb: for hosts that have no knowledge of the network, e.g. printer
- VDU: for a VDU connected via a full duplex RS232
- Intelligent: intended for PCs that can make full use of the network
- OSI: a packet orientated version of the intelligent mode satisfying the transport service of the ISO OSI 7 layer model

Before two users communicate, they must both 'offer a service', which means that they must be able to run LAN software in the host's memory. The dumb class of user cannot communicate, but only receive data. When node A wants to read data from node B, then node A 'onward connects' to node B to see if it busy (or if the service is still available). If node B is busy or unavailable, then the communication programmer must decide how often and for how long node A should retry, and what action to take when attempts fail, or what action to take when attempts are interrupted by an incoming message. Errors encountered in the message format are also left for the programmer to field. If node B was not busy, then the node interrupts the program running on its host and its host's program should respond to node A's request to read, find the information, onward connect to see if node A is busy and then transmits the information. The program running on the host of node A will be interrupted by node A so that the program can use the information. This contention problem, absent in simple point-to-point communication, is solved by a network's method of media access control (MAC) and logical link control (LLC), which constitute the OSI data-link layer.

2.7 Summary of Problems and Manufacturing Requirements

Problems encountered by using the 9Tiles network are summarised as follows:

- The node is a circuit board containing many components that are of unknown reliability and safety
- The host PC is itself known for its lack of reliability by the number of times it becomes inoperative
- NetBIOS and SNBIOS are also of unknown reliability and safety

The network was difficult to work with, and came with inadequate documentation. No tried and tested routines were provided for network services, and many of the problems arose in writing simple communication and service routines:

- Turbo Pascal was chosen as the programming language, because Pascal is a structured language with strong data typing. It can also access registers and support interrupt handling, but the compiler does not check for errors in such features
- No run-time debugger or network test software was available
- The implementation of communication within the manufacturing control software led to less readable and much more code
- The execution of the control code was uncertain, because there was no way of knowing when the code would be interrupted by the node and when messages were being sent and received

The functional manufacturing requirements are as follows:

- The lathe and miller must operate in parallel with the conveyor and cell robot, but the conveyor and cell robot must operate sequentially
- The local work handling for the lathe, miller and robot should co-operate with the lathe, miller and robot respectively during component transfer between pallet and process
- The lathe, miller and robot must be computer controlled
- The FMC should manufacture a family of parts. Chess pieces vary in piece, colour and set proportions (product flexibility)
- Not all pieces need milling, so the conveyor must allow variation in transportation (process flexibility)
- The FMC should cope with a variety of batch sizes (volume flexibility)
- The FMC should be re-configurable (expansion flexibility) to allow the swift inclusion, replacement and reduction of machine tools and materials handling (process flexibility)

The non-functional manufacturing requirements are as follows:

- Safety is the underlying concern
- No performance targets are set

3. Tools and Techniques in Dependable Distributed Control

3.1 Introduction

Chapter 2 described the manufacturing requirements of the FMC and the current control and communication of that control of the test FMC. This chapter discusses the component parts of the methodology, Petri nets and occam, and describes the benefits of a formal life cycle in the development of dependable software.

The chapter begins by examining dependability and assesses the required of the FMC's DCS. The dependability of the transputer and occam are highlighted, and their suitability in controlling and communicating that control is investigated by comparing them with an established industrial LAN and Fieldbus, with international standards and with the existing PC based network.

3.2 Dependability

For the purpose of this thesis, the author provides some working definitions regarding dependability. A system is:

- dependable when it is correct, reliable and safe
- correct when it does what is wanted
- reliable when it never fails
- safe when it harms no-one and damages nothing

Safety can only be achieved if the system is correct and reliable. These terms are discussed in sections 3.2.2 to 3.2.4. Section 3.2.5 relates these to hardware and software.

3.2.1 Software Engineering

The goal of software engineering is to identify the customer's desires and to realise them by means of a software implementation. The desires are for a design which either fits in with the current environment, or forms (possibly a significant) part of a new regime. Here the design consists of a system, its users, its environment and their interaction.

Where the environment is hostile, or where the users are or the system is under potential danger, then the design must make safety and/or security a priority. Customers desire the system to be dependable and flexible, but will only want to pay for appropriate safety, reliability and correctness, depending on their consequences and the constraints imposed.

As with all development processes, getting it right at the earliest stages is most important, because the later stages build on the foundations of the earlier ones. Often it is these early stages that are the most difficult to get right. This is certainly true with software, because in the later stages the designer is catering for the precise needs of a machine, but earlier on the software engineer is serving the less than complete desires of a human customer.

Software development is made up of four basic processes:

- requirements- identify what the customer desires
- system specification- define what the system will do
- design- describe how the requirements will be satisfied within the imposed constraints
- implementation- code the software to satisfy the requirements

Much work must be done within and between each of these processes. Figure 3-1 shows the largely sequential nature of software development, and is drawn as a Petri net, refer to section 3.3. It shows processes as active and deliverables (e.g. documentation and code) as passive. However, development is often accomplished iteratively, such as in verification at the design and implementation stage and validation at the requirements and system specification stages.

There are a number of software development life cycles (sometimes called development process models). [McDermid 1991] discusses a model of formal methods, the canonical model, the transformational model, Cohen's contractual model, the waterfall model, the 'v' model and the spiral model. These models have various stages or phases, but all include the requirement, system specification, design implementation processes. Prototyping is an alternative to the conventional life cycle. It aims to validate an implementation of the functional requirements and then develop that code, rather than validating a complete system specification and fully implementing a complete design.

Mathematical Modelling

In all scientific disciplines, shorthand notations are used to help describe and communicate ideas within a field. Where the ideas express relationships, then the discipline often describes them mathematically. Mathematical models of the real world can be shown to be valid through experimentation and verified to be consistent with similar ideas through mathematical analysis. The mathematical modeller can choose to model in either continuous or discrete mathematics.

The mathematics that underlies software engineering and computer science is called discrete mathematics [McDermid 1991]. It is based on logic, sets, functions, relations and algebras. Mathematical techniques used in software engineering include algebraic specifications, model oriented specifications, process algebras, statistics and graphs, while computer science is mainly based on automata theory, language theory and data structures and algorithms. Figure 3-2 shows their relationship corresponding to software development life cycle.

3.2.2 Correctness

For concurrent programs to be correct, all processes must be used and terminate successfully [Ben-Ari 1990]. This is also true for sequential programming, but demonstration of correctness is often avoided until testing, if at all. Concurrent programming can become so complicated, so it is prudent to approach correctness from a mathematically provable or formal perspective, refer to section 4.4.

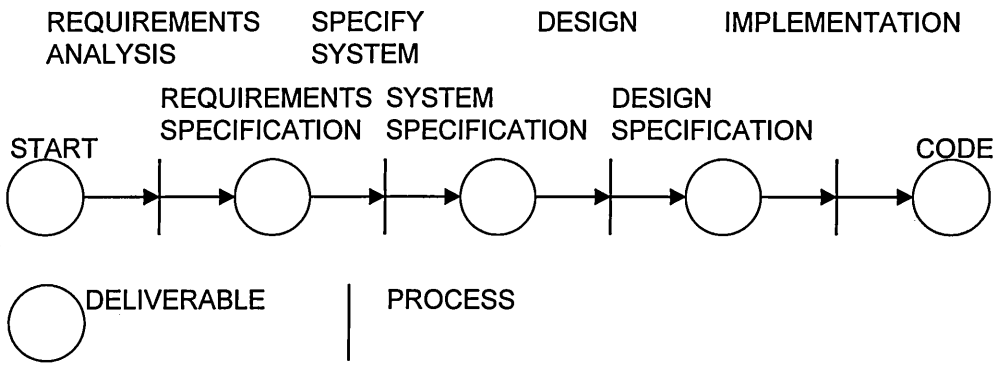


Figure 3-1 Software development life cycle, showing processes and deliverables

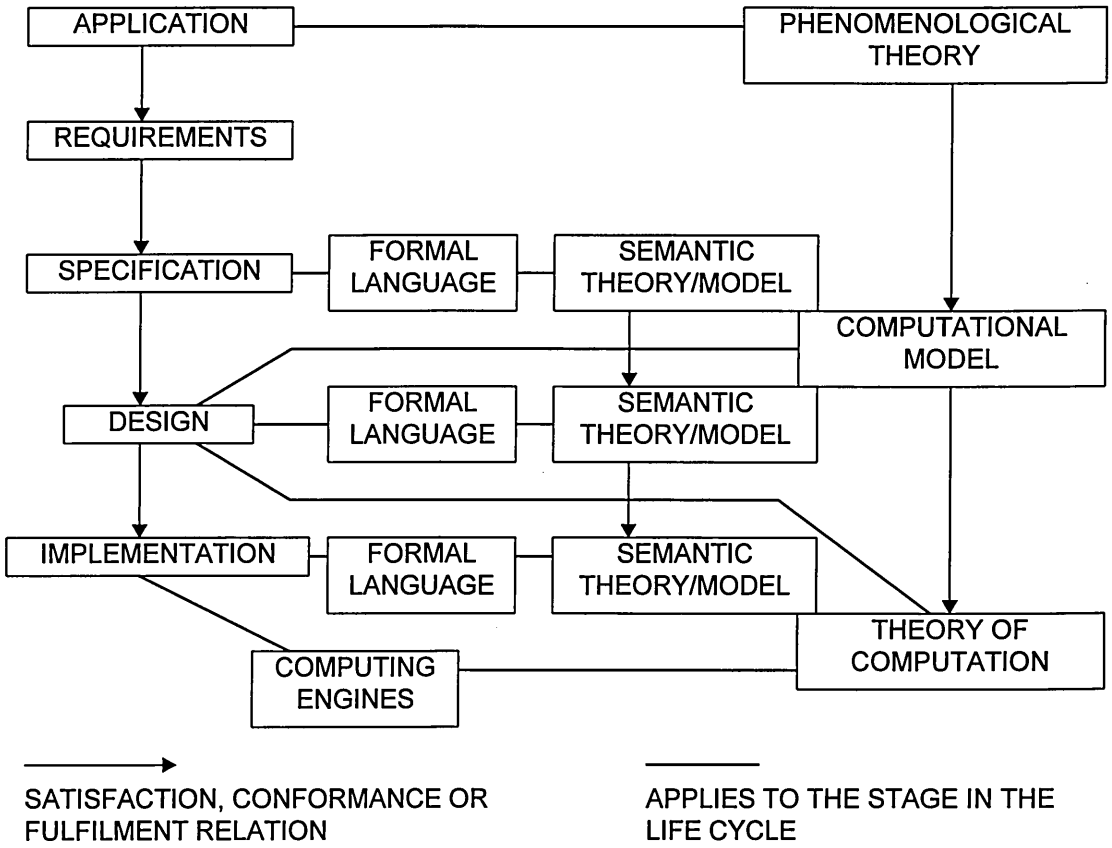


Figure 3-2 An example formal software development life cycle [McDermid 1991]

Partial and total correctness are defined as follows:

Partial correctness: For all execution sequences, if the correct values are input AND the program terminates when started with those values, THEN the processing of those inputs to give output values is correct.

e.g. if a>=0 AND square_root(a,b) terminates THEN b=√a

Total correctness: For all execution sequences, if the correct values are input, THEN the program terminates correctly when started with those values AND the processing of those inputs to give output values is correct.

e.g. if a>=0 THEN square_root(a,b) terminates AND b=√a

Sequential programs can always be made to terminate, but some concurrent programs are designed never to end, such as operating systems and process controllers, which only terminate on error. For this reason the definitions of partial and total correctness are qualified with 'for all execution sequences'.

Types of Correctness Properties

Safety properties must 'always' be true[Ben-Ari 1990]. For example in 'mutual exclusion', where two concurrent processes must not have access to a shared resource, such as the printer example in section 4.9. Another example is 'deadlock', where one of two concurrent processes prevents the other proceeding because it has stopped before it is due to communicate, refer to section 4.10. However, safety properties can be satisfied if a program does nothing, therefore another type of property is needed to ensure correctness in terms of degrees of 'liveness'.

Liveness properties must 'at some stage' be true. Contention for resources is won by one process, but will be eventually available for others. Fairness properties describe how contention and starvation are resolved. The first-in-first-out property of fairness is the strongest and is used as a basis for many schedulers.

3.2.2.1 Validation and Verification

The goal of validation is to show that something is true, i.e. valid, while the aim of verification is to show that one thing conforms to another. There is a distinction between 'producing the right thing' (validation) and 'producing the thing right' (verification). In the former, the desired entity is produced in either the correct or the wrong way, while in the latter, the entity is produced in the correct way but may not be what was wanted. The customer decides when a system specification is valid, so must be able to understand it.

Once the system specification is shown to be valid, then it is better to verify that a formal design specification conforms to a formal system specification, than to show that an informal design specification is as valid as an informal system specification.

3.2.3 Reliability

The failure of a system [Anderson 1981] occurs when the system deviates from its specification, and a fault is that deviation from the specification. The detection of an error is the evidence that one or more faults exist. Thus faults are manifest as errors, and can lead to system failure. In a computer system a fault will be either a hardware component fault or a software design fault.

Four reliability metrics can be used as criteria for comparing fault tolerant systems:

- mean time to detection (MTTD) - between fault occurrence and error detection
- mean time to repair (MTTR) - between error detection and completed repair
- mean time between failure (MTBF) - between two failure occurrences
- system availability - a fraction of (system up time) / time

Faults

Faults can be addressed at two stages in the software development life cycle. They are introduced as design faults, but are most costly as faults discovered during operation. Operational faults are categorised as: either localised or distributed, producing fixed or varying errors, and either permanent, transient, intermittent or latent.

Design faults can be:

- specification errors - bad algorithms, poor choice of hardware and software
- implementation errors - bad translation of specification to hardware and software

Software design faults tend to be permanent, though memory faults can render them transient. Transient faults usually lead to latent faults and finally to permanent faults.

Software reliability improvement techniques [Shimeall 1991] include:

- fault avoidance - prevent the introduction of faults during development
- fault elimination - locate and remove faults before use
- fault tolerance - operate in the presence of known and unknown faults

Fault avoidance involves the production of a correct system specification and implementation by employing appropriate (formal) development tools. Section 4.4 discusses formal development, and the methodology adopts Petri nets, section 3.3, to design and specify the system, and occam and transputers, sections 3.4 and 3.5.5, to implement and execute the system.

Fault elimination is not considered fully in the methodology, and is left for further work. Section 5.3.14 discusses the adopted testing techniques. The pro-active approach described in section 5.3.1 aims to minimise the need to test.

Fault tolerance is mentioned here, but is left for further work. Four principles are highlighted in fault tolerance [Anderson 1981]:

- error detection
- damage confinement and assessment
- error recovery
- fault treatment and maintenance

Five hierarchical levels exist in most computer systems: hardware, micro-code, macro-code, operating system and application, refer to SIFT in section 4.11. Fault tolerance can be applied to each of the levels, but application developers tend only to have access to the application level. Faults occurring at a low level can be manifest at higher levels and detected at higher levels still.

A safety system should be able to tolerate any single failure, and as many multiple failures as possible. If control is lost completely, the system should always be 'fail safe'.

3.2.4 Safety

3.2.4.1 Safety Standards

[Jesty 1993] compares two sets of draft standards aimed at the safety of computers and programmable electronic devices, and introduces a third.

Def Stan [00-56 1991] concerns hazard analysis and safety classifications, and, where the safety integrity level is found to be significantly high, determines the use of Def Stan 00-55. [00-55 1991] specifies the procedures and tools to develop safety critical software, namely the use of formal methods for specification, design and verification.

[IEC/SC65A] WG9 & WG10[1] are concerned with the software and the system respectively. Once the system safety integrity level is defined with WG10, then WG9 is used to select the appropriate design technique.

Two main criticisms in the standards are in the imposition of the use of unagreed methods, and the relationship between safety integrity and probability of software failure. This motivated Jesty's proposed standard, which differs little from the other two.

Firstly, although 00-55 deals only with safety critical software, it provides strict definitions as to what constitutes a suitable formal method, thereby ruling out some methods where no consensus lies in the safety critical community. In contrast WG9 includes all integrity levels, but determines which method types to use for each particular level.

Secondly, software faults, unlike hardware faults, are purely systematic, see section 3.2.5, so the probability of failure of software cannot be based on randomness. Safety integrity levels, in 00-56, imply specific minimum failure rates, but because the failure probability of software is not related to the integrity of the software development method, then this means that good practice does not guarantee good software. For this reason Jesty introduces confidence levels. Confidence levels can be associated with probabilities of failure, and can be mapped onto levels of integrity. The strategy is: the more the design and implementation processes are known, the greater confidence can be placed on the software produced by them. It is therefore optional to use formal methods at any integrity level, but it is only at the highest level where formal methods are mandatory.

---

[1] Now the standard IEC 1508 Functional Safety - Safety Related Systems July 1995

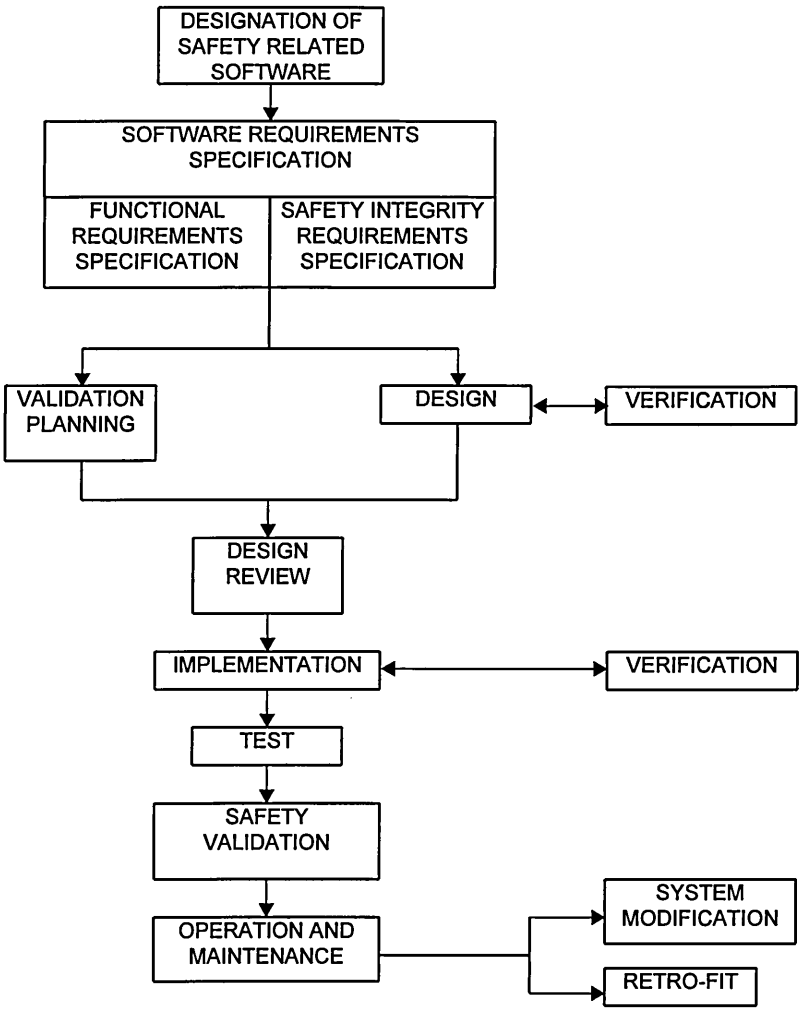


Figure 3-3 A life cycle for safety critical software

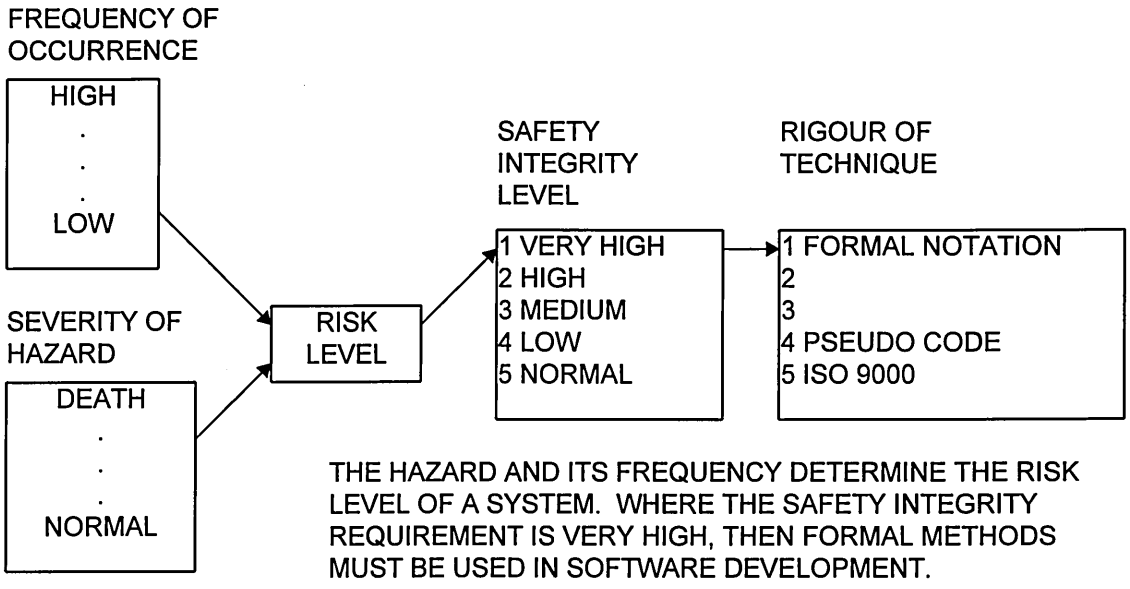


Figure 3-4 Risk level determines safety integrity and rigour of technique

The proposed standard borrows its safety critical life cycle from IEC, and is outlined in Figure 3-3.The designation of safety related software is established through hazard analyses. The requirements specification is made up of functional requirements and safety requirements. The latter includes measures to avoid faults and measures to control faults. The design and implementation processes make frequent consistency checks ensuring their validity by verifying with themselves and the requirement specification. Testing and safety validation are performed before the software is deemed satisfactory and set into operation.

Defence Standard 00-55

Def Stan 00-55 defines that a formal method must meet several criteria before it is suitable for safety critical applications:

- it should be recognised as a formal notation, capable of expressing specifications and designs mathematically, either by itself or with another formal method
- the design steps should be verifiable, so should have a proof of theory and guidance on how the theory can best be exploited
- it should be in the public domain, with accessible courses, textbooks, case studies and industrial tools
- it should be standardised and preferably be a standard.

Examples given of formal methods for reasoning about sequential properties are VMD and Z, and about concurrent and communicating properties is LOTOS.

The 00-55 standard discusses validation and checking the formal specification by:

- mechanical checks for consistency
- proof checkers and editors
- symbolic execution tools
- animation for validity, completeness, consistency, reachability and redundancy

Another way is to produce a (verified) executable prototype:

- from an executable subset of the formal method, and gives objex for OBJ as an example
- by translation into a logic programming language e.g. prolog
- by translation into a language with strong data typing, e.g. Pascal.

3.2.4.2 Safety Integrity Level

Very early in the development of a system, the software engineer must assess the hazards associated with the system and the likelihood of them happening. Significant hazards are: 1 loss of human life, 2 injuries to, or illness of, persons, 3 environmental pollution, 4 loss of, or damage to, property, and likelihood is the frequency of occurrence. Where the combination of the hazard and its frequency, the risk level, is too high then the system should be treated as 'safety related' or even 'safety critical', and more stringent levels of dependability must be applied.

The major difference between a system designed for safety and one which is not is that all other (functional and non-functional) characteristics of the system are subordinate to safety, and that every

effort must be made to avoid the hazard happening and to minimise its effects. This means that it can be better to 'fail safe', i.e. stop, when a fault occurs than to recover and continue with reliable service.

CATEGORY	DEFINITION
Catastrophic	Multiple deaths
Critical	Single death/multiple severe injuries
Marginal	Single severe injury/multiple minor injuries
Negligible	At most a single minor injury

Table 3-1 Accident severity categories

PROBABILITY	OCCURRENCE during operational life
Frequent	Likely to be continually experienced
Probable	Likely to occur often
Occasional	Likely to occur several times
Remote	Likely to occur some time
Improbable	Unlikely, but may exceptionally occur
Incredible	Extremely unlikely that the event will occur at all

Table 3-2 Probability ranges

	CATASTROPHIC	CRITICAL	MARGINAL	NEGLIGIBLE
Frequent	A	A	A	B
Probable	A	A	B	C
Occasional	A	B	C	C
Remote	B	C	C	D
Improbable	C	C	D	D
Incredible	D	D	D	D

Table 3-3 Risk classification of classes

RISK CLASS	INTERPRETATION
A	Intolerable
B	Undesirable, only accepted when risk reduction is impracticable
C	Tolerable with the endorsement of the project safety review committee
D	Tolerable with the endorsement of the normal project reviews

Table 3-4 Interpretation of risk classes

Standards on safety in computer controlled systems recommend methods to determine the level of risk associated with the system. Once established, risk levels indicate the 'overall system integrity level', and recommend the minimum level of rigour of technique to adopt when developing the software. Figure 3-4 shows that software written for a system with a very high level of risk should be developed using formal methods, but [ISO 9000] must be followed when developing software for systems at all risk levels.

3.2.4.3 Hazard Analysis

It would seem hypercritical to advocate the adoption of standards without even partial use of them. It is for this reason that a cursory assessment of safety integrity is made. Def Stan [00-56 1991] is used because it specifies hazard analysis and safety classification and it is currently available, albeit in interim form.

It is assumed that an industrial FMC will not be as fully guarded as is the test FMC. It is considered that the cell robot is the most dangerous hazard, because it could 'launch' a 5 kg mass with a linear velocity of 3 m/s. The estimated consequences are "a single death and/or multiple severe injuries", however this would be "unlikely, but may exceptionally occur". With reference to the accident severity categories (Table 3-1) and the probability ranges (Table 3-2), then a risk class can be determined (Table 3-3). A "critical" accident severity with "improbable" probability of occurring leads to a "C" accident risk class, which means "tolerable with the endorsement of the project safety review committee" (Table 3-4). The safety integrity level for the first safety feature must be S4 (Table 3-5a), because the accident severity is critical, and subsequent safety features must be S2 (Table 3-5b).

Safety features are devices or methods that reduce risk, either by reducing the probability of occurrence or by reducing the accident severity, and thus improve the safety integrity. Formal methods can be used as a safety feature in specification, design and verification, and are deemed as necessary in safety critical software as specified in Def Stan 00-55. Table 3-6 permits the claim of a 'remote' failure rate for integrity level S4 and 'probable' for S2. The minimum failure relates only to systematic failure.

Hazard analysis for the FMC determines it a safety critical system, which requires the use of formal methods.

3.2.5 Software and Hardware

Dependability in software and hardware is different. Correct hardware can become unreliable, correct software cannot. Similarly, only hardware can be unsafe.

Software is abstract, not physical, so cannot wear-out. It is merely correct or incorrect. Faults or bugs are introduced inadvertently during development, and may only appear under certain conditions. Such faults are known as systematic, because they are a function of the development process and the way the software is used.

Computer hardware components deteriorate or wear out just like any other physical object. Engineers have been able to predict such random failure, because it is a function of usage and time. Hardware, like

software, is designed, so is also subject to systematic failure. However, hardware is generally much simpler than software, so systematic failure is generally insignificant compared with random failure in the operational life of hardware.

Intermittent or permanent hardware faults can be caused by electromagnetic or other environmental effects. Some of which, like lightning, and are unpredictable and can only be dealt with by fault tolerance. Others, like high e.m.f.s caused by power machinery, can be forecast and should be part of the system specification. In a computer system, memory failure is a most common hardware fault. This can be reduced by minimising the memory requirement, using auto error correction memory, and localising it as far as possible [advocating distributed systems].

Accident severity			
Catastrophic	Critical	Marginal	Negligible
Level S4		Level S3	Level S2

a) Safety integrity for the only or first function

Failure probability of first function	Accident severity			
	Catastrophic	Critical	Marginal	Negligible
Frequent	Level S4			
Probable		Level S3		
Occasional			Level S2	
Remote				Level S1
Improbable				

b) Safety integrity for the second and subsequent functions

Table 3-5 Assignment of safety integrity levels

SAFETY INTEGRITY LEVEL	MINIMUM FAILURE RATE
S1	Remote
S2	Occasional
S3	Probable
S4	Frequent

Table 3-6 Claim limits

Faults can be localised if the system integrates processors asynchronously, rather than synchronously. This 'loose' form of synchronisation is synchronised by inter-processor communication and not by global clocks. Transient faults (e.g. caused by lightning) are more easily detected if processor executions are at different times (temporal separation). Unfortunately the loose synchronisation cannot detect inter-processor communication deadlocks, and prevents data on TMR input channels being identical.

3.3 Petri Nets

Petri nets are categorised into three classes: place-transition, general or normal (see section 3.3.1); sub-classes; and high level or extended (see section 3.3.2).

3.3.1 General Petri Nets

Petri nets [Peterson 1981] are a group of specification languages that can model, and help analyse, interacting concurrent processes. Petri nets consist of a set of places and a set of transitions, where places represent processes and transitions represent synchronisation.

3.3.1.1 Characteristics of Petri Nets

- Concurrency - two or more processes proceed simultaneously (see Figure 3-13 outputAND and inputAND)
- Synchronisation - two or more processes interact. This is done asynchronously, without reference to time (see Figure 3-12)
- Conflict - one process wants to synchronise independently with two or more other processes. Which independent synchronisation should occur? The conflict in synchronisation leads to non-determinism, but is the basis of choice and provides Petri nets with their decision power (see Figure 3-13 outputOR)

3.3.1.2 Petri Net Graphs

A Petri net graph is a graphical representation of a Petri net structure, and is described as a bipartite directed multi-graph. The places are depicted as 'circles' and transitions as 'bars'. 'Directed arcs' connect places and transitions. Input arcs leave places and enter transitions, and output arcs leave transitions and enter places, refer to Figure 3-5. The 'multiplicity' or arc weighting between places and transitions is depicted as multiple arcs or written as an integer, refer to Figure 3-6 [the former is easier to read for low weighting].The Petri net structure and Petri net graph are equivalent. See appendices for their formal definitions.

Tokens are introduced to indicate progress through a system, so that dynamic modelling, or 'animation', can occur. Tokens are depicted as 'dots', and reside in places. The marking of the tokens on the Petri net is called a marked Petri net. Initial conditions of a system can be modelled in a Petri net as the initial marking. Places can be bounded, which imposes a limit on the maximum number of tokens that a place can hold.

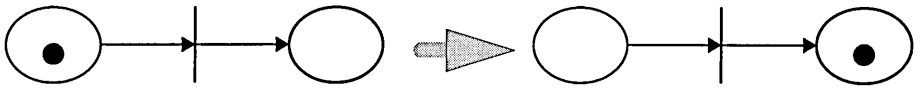
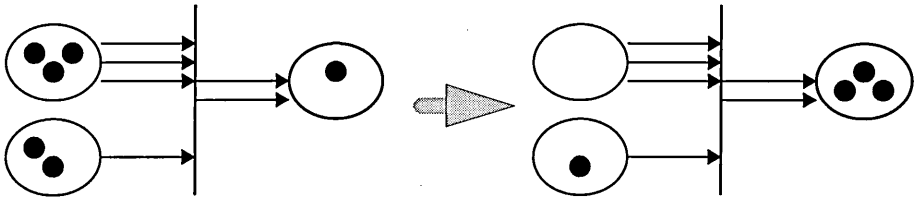


Figure 3-5 A Petri net graph consists of places, transitions, input and output arcs

Figure 3-6 Arc weighting or multiplicity shown with multiple arcs or a number.



Firing removes as many tokens as input arcs, and provides as many tokens as output arcs

Figure 3-7 A Petri net before and after firing

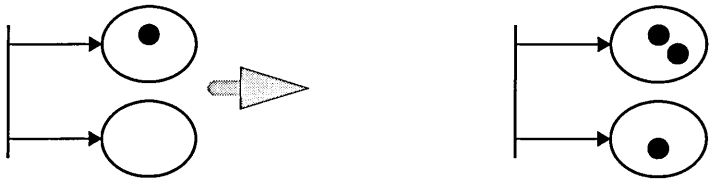


Figure 3-8 Source transitions generate tokens

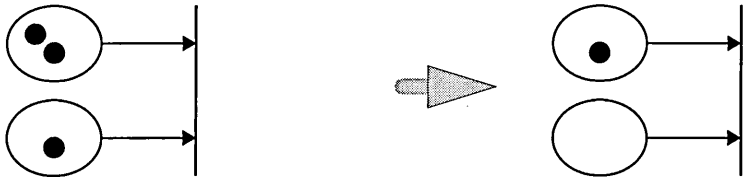


Figure 3-9 Sink transitions consume tokens

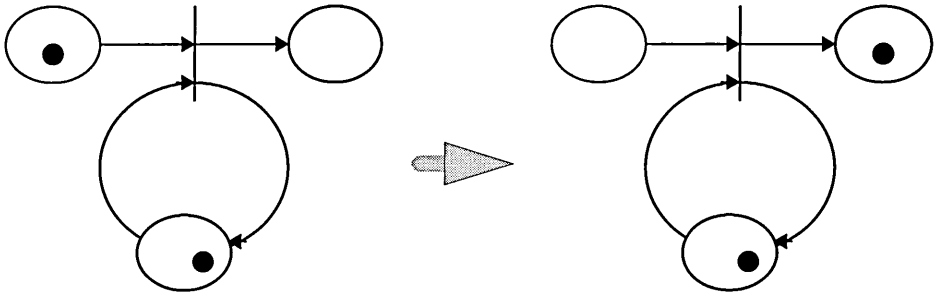


Figure 3-10 Self loops before and after firing

3.3.1.3 Firing Rules

A transition can fire when all its input places are enabled. A place is enabled when it contains enough tokens. When a transition fires, it removes a token from all of its input places and adds a token to all of its output places, refer to Figure 3-7. Where there is more than one input arc between input place and transition, then there must be at least that number of tokens in that place before it is enabled. Similarly after firing, each output place receives as many tokens as there are output arcs. If any of the output arcs are bounded, then the transition cannot fire if this will cause an output place to exceed its capacity.

A 'source transition' has no input places, therefore can always fire and generate tokens, refer to Figure 3-8. A 'sink transition' has no output places, so consumes tokens when enabled, refer to Figure 3-9. A 'self loop' is where an input place of a transition is also an output place, refer to Figure 3-10.

The 'state' of a Petri net is its marking. The 'next state function' of a Petri net is a change function. The reachability set for a Petri net is defined as all markings that are reachable from that marking.

3.3.1.4 Petri Net Properties

The system properties, of conservation, liveness, boundedness, safeness and proper termination, can be modelled and analysed in Petri nets. Compare them with the correctness properties in section 3.2.2.

Conservation: A Petri net is 'conservative' if the total number of tokens in the net remains constant. This is achieved when the number of arcs entering a transition is the same as the number leaving.

Liveness: A Petri net is 'live' if there always exists a firing sequence allowing any transition in a net to fire. By proving a Petri net is live, the system will not 'deadlock'.

Boundedness and safeness: A Petri net is 'bounded' if, for each place in the net, there exists an upper bound to the number of tokens that can be there simultaneously. If the upper bound is set at one token per place, then the Petri net is 'safe'.

Proper Termination: A Petri net is 'proper terminating' if the net always terminates in a well defined manner such that 1) only one token remains in the net 2) the number of tokens used is finite. Proper termination ensures no side effects on the next usage of the system, or on other systems.

Petri Net Languages

Reachability is concerned with 'what' markings are reachable. Petri net languages [Peterson 1981] are concerned with 'how' such markings are reached. A sequence of transitions is called a 'string', and a set of strings is a 'language'. The sequences of strings will show how the markings are reached. The set of all possible strings 'characterises' the behaviour of the Petri net model and the modelled system. Two systems modelled as Petri nets can be compared by examining their characteristics. The two Petri nets are identical, or equivalent, if their languages are equal.

'Optimisation' is where a new Petri net is produced which is equivalent (same language) as the old one, but has fewer places, transitions and arcs. Thus optimisation reduces the number of transitions and places without altering the behaviour of the model.

3.3.1.5 Petri Net Analysis

Petri net analysis is either applied to an unmarked Petri net or to a Petri net with its initial marking. The former is concerned with the structural properties, and the latter to behavioural properties.

Petri net properties of safeness, boundedness, conservation and coverability can be determined by reachability tree and matrix analyses. Properties of reachability, firing sequence and liveness are not guaranteed to be solved using these techniques due to the lack of unique solutions. [This means that it is better to prevent deadlock than to detect it.]

Petri net theory offers various analysis techniques [Heiner 1994]: prototyping, static analysis, dynamic analysis and model checking, as follows:

Prototyping of functional characteristics by animation or the 'token game' is a visual check, and is used for validation with the customer.

Static analysis includes net reduction, structural analysis and net invariant analysis, and does not require the construction of a reachability tree. Net reduction and structural analysis are context checking techniques, while net invariant analysis is a verification technique.

Dynamic analysis involves the construction of a complete or reduced reachability tree. Reachability tree or state space analysis is used for checking dynamic properties, such as dynamic conflict.

Model checking superimposes temporal logic on a reachability tree. This enables questions to be asked of the tree, and helps manage large trees.

3.3.1.6 Reachability Tree

The reachability tree is another graphical tool, but is made up of nodes and arcs only. The first node is always the initial marking. One arc is drawn from this node for every next state from that marking. For example, if two transitions can fire from the marking, then two arcs are drawn pointing to two new nodes and are given labels corresponding to the transition which fired. Each new node represents the new marking given that particular firing. The step is repeated until one or more of the following:

- end markings of the reachability tree already appear higher up the tree (duplicate nodes)
- end markings are duplicate nodes, except one is pre-determined (proper termination)
- end marking are duplicate nodes, except one is the initial marking
- end markings arise because no further transition can fire (terminal nodes)
- one or more of the places has an ever growing number of tokens
- the reachability tree grows without any patterns forming

The reachability tree cannot solve the reachability or liveness problems or be used to define or determine which firing sequences are possible, because of the infinity symbol in the reachability tree. The infinity symbol denotes an ever growing marking, and as such represents an abbreviation of information, which prohibits proof of these solutions.

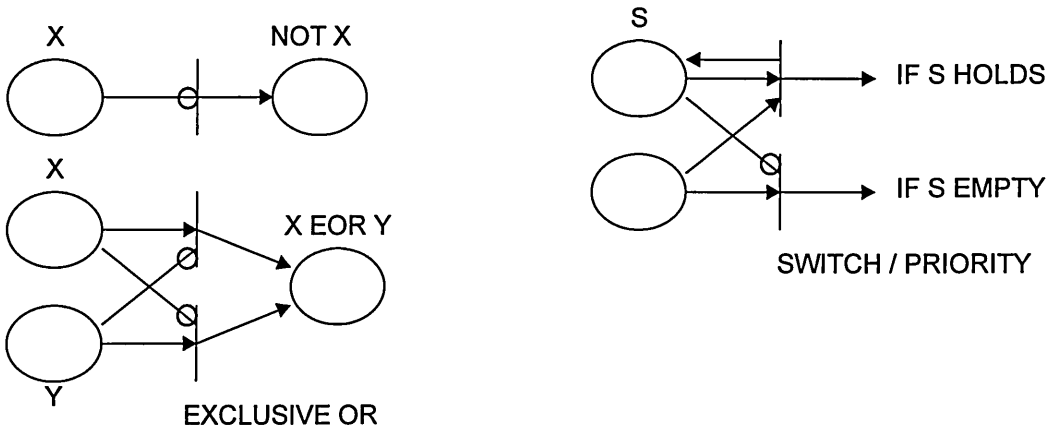


Figure 3-11 Inhibitor arcs can model: NOT, exclusive OR, switch and priority

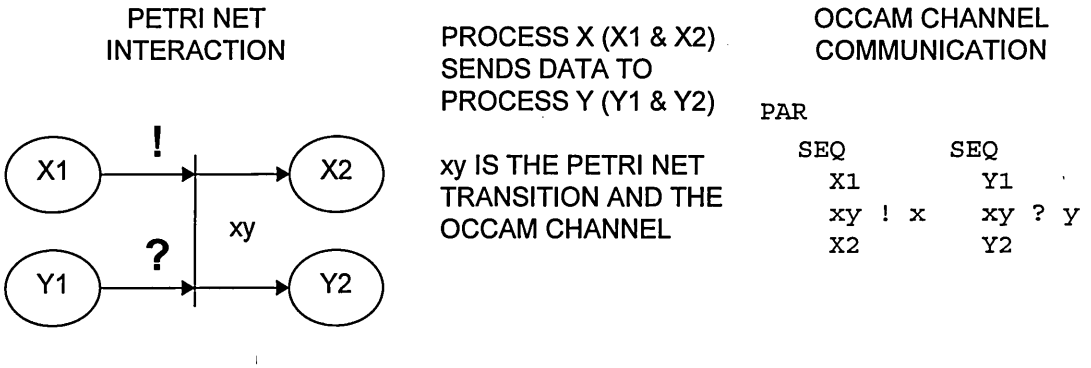
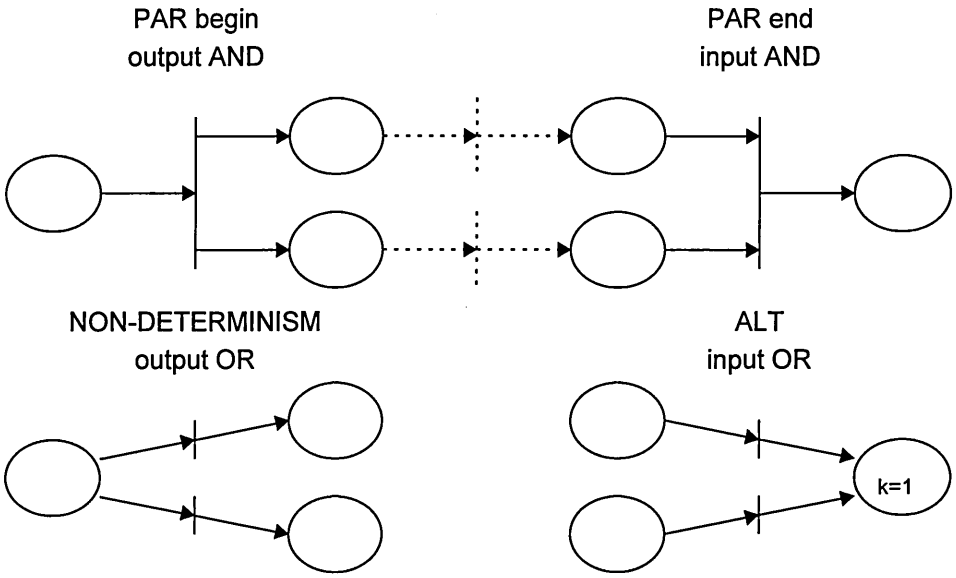


Figure 3-12 Communication in Petri nets and occam



FOUR PRINCIPLE PETRI NET CONSTRUCTS:
'output AND' AND 'input AND' REPRESENTED BY OCCAM 'PAR', 'input OR' REPRESENTED BY THE OCCAM 'ALT',
BUT THE PETRI NET 'output OR' HAS NO OCCAM EQUIVALENT

Figure 3-13 Petri net / occam equivalences

A bounded Petri net, whose places cannot contain more than a specified number of tokens, has a finite reachability tree. A safe Petri net restricts the maximum number of tokens per place to one. Unbounded Petri nets do not necessarily have an infinite reachability tree, but this cannot be guaranteed. A conservative Petri net has a finite reachability tree, and conservation is demonstrated by summing each reachable marking.

3.3.2 Other Petri Nets

Petri net extensions generally aim to improve the modelling power of Petri nets. However, the trade off between modelling and decision power suggests that such extensions will reduce decision power. Petri net restrictions aim to improve decision power or safeness (at the expense of modelling power) by limiting the structure of Petri nets.

3.3.2.1 Petri Net Sub-Classes

Examples are:

- state machines restrict transitions to one input and one output arc, leading to high decision power but low modelling power. They are conservative, safe and have a finite reachability tree.
- marked graphs restrict places to one input and one output, leading to high modelling power but low decision power. They are conservative in cyclical structures, so are live and safe (if bounded to one)
- free choice Petri nets isolate the potential conflict inherent where one place inputs to many transitions. By preventing any other place to input to any of these conflicting transitions, firing does not disable other places, rendering them live and safe.
- ordinary Petri nets restrict the multiplicity of arcs to one
- pure Petri nets contain no self-loops

3.3.2.2 Petri Net Extensions

The simplest Petri net extension is the 'inhibitor arc', which is an input arc that allows the test for zero tokens in the input place. In the Petri net graph, the inhibitor arc is differentiated from normal input arcs by replacing the arrowhead by a small circle. Inhibitor arcs improve modelling power (at the cost of decision power), for example: NOT, exclusive OR, switch and priority, refer to Figure 3-11.

Nets that are considered modifications rather than extensions often augment places or transitions. Timed Petri nets add deterministic firing times to transitions, while stochastic Petri nets impose probability to the firing of transitions.

3.3.2.3 High Level Petri Nets

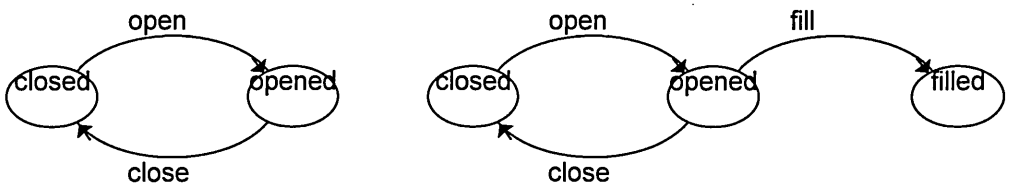
High level nets are folded versions of general Petri nets (if finite). Unfolding the high level Petri net produces a set of places, where one place was, which held the token's multi-set, and produces a set of transitions, where one transition was, which fired according to the function associated with the input and output arcs. Folding a general Petri net into a high level Petri net is the reverse of this.

High level and general Petri nets are read quite differently. In place-transition Petri nets more emphasis is given to reading the places and transitions rather than the tokens and arcs, while in high level Petri nets more emphasis is on reading tokens and arcs rather than places and transitions.

The major high level Petri nets are predicate-transition nets [Genrich 1987], parameterised (hierarchical) Petri nets [Gracanin 1994] and coloured Petri nets [Jensen 1991].

3.3.2.4 State Transition Diagrams (STDs)

STDs [McDermid] are a special case of finite state machine (FSM), and consist of a set of states S, a set of events E and a transition function T. The transition function maps the current state and event into the next state. For example, a box with states {opened, closed}, and events {open, close} will have transition functions t(closed, open) = opened and t(opened, close) = closed, in the left STD below.



In the right STD above, a new state {filled} is added with event {fill}, which can only happen when the box is opened, and has the transfer function t(opened, fill) = filled.

State machines are a sub-class of Petri nets [Peterson], because Petri nets can model a state machine by restricting each transition to having exactly one input and one output. State machines are therefore conservative, resulting in a finite reachability tree (hence the name FSM) A state machine is live if its initial marking has at least one token, and safe if it has exactly one token.

State machines have high decision power, but low modelling power. Transition functions can represent the decisions, i.e. a state can be subject to more than one event, e.g. fill or close an opened box. Modelling is limited to finite systems, and sequential ones, because the absence of synchronisation precludes concurrency.

3.4 Occam and its Dependability

3.4.1 The Occam Language and Programming Environment

Occam [Inmos 1988, Jones 1988], like Pascal, C and FORTRAN, is a high level programming language. It was developed from the mathematical foundation of communicating sequential processes (CSP) [Hoare 1985] to represent concurrent models and their implementation on one or a network of processors. The transputer is the hardware implementation of occam, and together they are used in real time, embedded, computational intensive, communication and distributed applications.

To highlight some of the significant characteristics of occam, a brief comparison is made with Pascal. Occam and Pascal are strongly typed, modular imperative languages, and make use of libraries of

standard and user defined software. The comparison is not straight forward, because occam code can only be fully implemented on the transputer, while source code written in Pascal is meant to be compilable on any processor which has a compiler written for it.

	Occam	Pascal
Types- data	INT, BOOL, REAL64, BYTE arrays	Similar
channel	CHAN OF data type. PROTOCOL chan IS data type	NE
time	TIMER	NE
Action	Assignment (:=), channel input (?) and output (!).	Assignment
Constructions	SEQuence, PARallel and ALTernate.	Sequential
Processes	SKIP, STOP, constructions and actions.	Blocks
Block/Scope	Indentation	Begin..end
Conditional	IF	Same
Selection	CASE .. ELSE	Similar
Priority	PRI PAR, PRI ALT. High or low.	NE
Replication	SEQ name = base FOR count. SEQ, PAR, ALT, and IF can be replicated	NE
Loop	WHILE Boolean expression	WHILE, REPEAT
Procedures	PROC parameters passed by value and reference	Similar
Functions	FUNCTION takes parameters passed by value	Similar
Configuration	PLACE processes on transputers, channels on links	NE
	NE = no equivalent	

Table 3-7 Comparison of occam and Pascal

Table 3-7 shows most of the occam syntax and the Pascal equivalent. Some of the significant dissimilarities between occam and Pascal are as follows:

- Channels provide point-to-point communication, and are configured as links between transputers. Protocols allow channels to carry messages containing one or more different data types. Variant protocols enable one or more selected data types to be transmitted through one channel, by first specifying the protocol tag.
- High and low priorities exist for process and timer scheduling. Low priority processes (default) use a 64 microsecond tick, 15625 ticks per second, while high priority processes utilise a 1 microsecond tick. A high priority process is scheduled before a low priority process.
- Processes are modules of code, whose scope is bounded by line indentations. SKIP processes start and immediately terminate, while STOP processes start but never terminate, and are useful in software development and testing.
- Procedures allow processes (i.e. modules of code) to be named, but cannot be used recursively. Functions are named processes that produce results, and consequently are used within expressions.

The function body forbids assignment to free variables (i.e. declared outside the function), communication, alternations or parallel constructs. In this way, functions are free from side effects.

- In every conditional, such as IF and CASE, at least one choice or option must be TRUE, otherwise it will behave as a STOP.
- Replication and loops are different. In occam and Pascal, loops repeatedly execute processes or blocks of lines. Replication in occam reduces the size of source code, because the compiler expands it in the object code.
- Configuration assigns processes to processors and channels to one of the four communication links. This enables transputer systems to run without the resources provided by operating systems.

3.4.2 The Occam Toolset

Various occam 'programming environments' exist. The one used in the development of code was the toolset for DOS. The toolset consists of a number of tools which are run from the DOS prompt, and are generally applied to the code in sequence, as follows: checker, compiler, linker, debugger, a tool to boot a single transputer, a tool to boot a network of transputers. A folding editor is supplied with the toolset. It has a highly modular way of grouping, hiding and moving lines of code, but requires learning a totally new set of key strokes to operate.

In addition to all of the rules of occam, the toolset compiler understands directives such as #USE and #INCLUDE. #USE links in compiled code, while #INCLUDE brings in other source code.

With few exceptions, concurrent occam code written for one transputer can be compiled to run on a different transputer or a network of different transputers. For example, a transputer with no floating point unit (FPU) must use an appropriate library to work with high precision real numbers in the source code, whereas a transputer with a FPU has no need. Other differences are accommodated by the toolset.

Implementation Difficulties

The occam language is defined with few limitations [Inmos 1988], and the toolset does not document them all [iTools]. Three difficulties were encountered during the development of the code.

- There is a maximum size for the condition IF. The procedure `index.wanted()`, in file `cellCon.occ` in the appendices, has two IF constructs where only one is wanted.
- There is a maximum size for sequential protocols. The channel to `update` the cell controller with statuses from the status handler was to have 57 items, but ended up being split into one of 10, three of 15 and one of 2, and transmitting them sequentially.
- Only two included header directives could be implemented above a procedure successfully.

3.4.3 Occam Safety

[Croll 1990] examined occam, as to its safety, against criteria taken from [00-55 1991]. The results were then compared with those of six other languages, which were subjected to the same examination conducted by [Cullyer 1991].

The criteria from Def Stan [00-55 1991] were:

- Wild jumps: determinable control flow - no GOTO statements
- Overwrites: prevention of arbitrary memory allocation
- Semantics: sufficiently formal to allow static analysis
- Model of maths: formal models of integer and real arithmetic
- Operational arithmetic: adequate checks
- Data typing: strong enough
- Exception handling: tolerate runtime errors
- Safe subset: existence of a 'safe' subset
- Exhaustion of memory: prevent runtime stack or heap overflow
- Separate compilation: type checking across pre-compiled modules
- Well understood: can programmers 'use' the language safely

The six languages that Cullyer chose were assembly and subset, C, CORAL 66 and subset, ISO Pascal and subset, Modular 2 and subset, and Ada and subset.

Cullyer made the following conclusions:

- SPADE (the Pascal subset), 'safe Ada' and SPARK (Ada subsets under development) are best suited for high integrity systems. They should enable designers to satisfy the requirements of national and international standards for safety critical systems.
- Ada, modular 2 and ISO Pascal, with only slight restrictions, are suitable for low risk hazards, and are commonly available, but will not satisfy the strictest test requirements of safety standards.
- Newly developing languages, such as NewSpeak will provide added security in 'exception free' or runtime fault recovery facilities.
- Subsets, designed for safety critical systems, should be developed along side the programming language.

Croll compared the results of the two examinations, and concluded that occam performs very favourably, and that "occam is a prudent choice for safety purposes".

3.4.4 Deadlock Correctness

[Jones 1988] outlines criteria for correct occam code:

- it is important that the data values passed in messages and stored in variables cannot be incorrect
- it must be shown that the system cannot become deadlocked: that there is always something that the system can do

- it must be shown that the system cannot become livelocked: that by doing something that it can do, the system makes progress

The glossary in [Inmos 1988] defines:

- deadlock- a state in which two or more concurrent processes can no longer proceed due to a communication inter-dependency, refer to section 4.10.
- livelock- a divergent process, one which remains internally active but does not perform further communication, i.e. it behaves like the following process:

```
WHILE TRUE
  SKIP
```

Livelock, in contrast to deadlock, is a state that can potentially proceed, but one or more processes starve others of resources. An arbiter, or code that guarantees fairness, would eliminate livelock by imposing a fair order in which processes are executed (e.g. FIFO).

The Inmos occam toolset compiler [iTools] checks for usage, such as abbreviations, subscripts and replication indices, but not all code structures, guards and communication values are allowed at compile time. Compiler options regulate the scope of such checks.

The toolset does not implement the code in a fair manner, e.g. the first in a list of alternatives to be ready in a looped alternative is executed, even though alternatives lower down the list are ready at the same time. Using guarded alternatives, or ensuring the logic avoids this, are solutions to the unfair implementation. The following process illustrates the unfairness:

```
WHILE TRUE
  ALT
    channel.ready.and.always.successful
      process.executes
    channel.also.ready.but.never.successful
      this.and.interacting.processes.starved
```

The occam STOP starts, but unlike the occam SKIP, it never terminates. It can force a program to deadlock, or to not terminate. If a process contains a STOP before the communication, then this and the process with which it communicates cannot proceed. However, if the STOP occurs after the communication, then only this process (and the whole program) will not terminate.

3.5 A Comparison of Transputers and other Shop Floor Controllers

It is evident from the introduction that manufacturing flexibility is necessary in today's market. This can only be achieved if manufacturing control is flexible and the distribution of that control is flexible. This section introduces communication networks, and compares common features and special requirements of office and shop-floor networks. Synchronisation and communication models that enable satisfactory interaction, when competing for shared resources and when sharing information, are given. International standards defining communication in general and shop-floor communication are included. The following

sections highlight their importance, and describes how they might be implemented. Two 'proven' shop-floor DCSs are compared with the 9Tiles network and a transputer/occam implementation for suitability.

3.5.1 Distributed Control Systems and Networks

Computer networks connect processing elements and enable them to operate concurrently, interact and communicate between each other. They vary between closely connected networks, as in processor arrays, and distributed. A distributed network is a medium of communication between remote users and resources. The physical media or carriers are mainly wire, fibre-optic, micro-wave and radio-wave. Networks enable messages to be conveyed between nodes which use the same protocol, and gateways connect two networks of different protocols. Modern computer networks allow users to have access to other users, and to information, all over the world. Messages might travel via various networks, of different characteristics and speeds, and be converted between a number of protocols.

DCSs can be described by a number of common characteristics, which are speed, media characteristics, reliability, protocol, hardware and software and connectivity. Many networks require special characteristics such as security, timing and environment. These are outlined:

- Transmission speed or data rate is measured in bits/s.
- Two media characteristics are bandwidth and noise capacity: bandwidth is the frequency range capable of being carried and is measured in Hertz. Signal-to-noise ratios are measured in decibels.
- Reliability of transmission medium and network nodes are measured in bit error rates, message losses and node availability.
- Protocols are specifications for conformity, and can be the way data is packaged and transmitted, how messages are given access to the network, or how messages are constructed.
- Hardware and software: a node, connecting user to network, has a processor which runs software to send, receive, generate and manipulate messages.
- Connectivity describes how different users connect to the network, and how networks connect together.

Special networks have additional requirements such as:

- security- to prevent messages being 'tapped into', accessed or corrupted
- timing- for very fast or time critical timing
- environment- noise immunity
- safety- reliable and correct operation

User software (or application software) and users exploit the facilities of the network. Network design or selection would depend on how these features best satisfy the user requirements. Two examples illustrate different network demands in office and shop-floor networks:

Office networks: usually connect computers, mass-data-storage and printers to enable office users to interact and to access, manipulate and print data-files. Generally, networks need no special requirements, and often join homogeneous or directly compatible equipment together. For example, rather than each computer being connected to its own printer over a point-to-point RS232 cable (physical media protocol)

sending and receiving ASCII characters (message protocol), the computers could share one printer via a network. When many users want to print at the same time, the network could buffer all data-files, or could control computer access to the printer.

Office networking is by far the most common application, and as such software supplied with the network often includes office related features. However, apart from embedded systems, where human interaction in negligible, this is justified.

Shop-floor networks: have special environmental, timing and safety requirements as well as a variety of users and connectivity. Highly powered machine tools generate a great deal of electromagnetic noise, which interferes with transmissions based on electromagnetic waves (e.g. wire, micro-wave and radio-wave). This and other environmental factors (temperature, humidity, dust) reduce the reliability of all the components of the network. Many manufacturing operations must be monitored and corrective action taken to maintain safety.

Digital signal processing (DSP), polling, safety, deterministic and other real time and time critical requirements, imposed by shop-floor operations, must be met by the network. In today's markets, quick response to network layout modification and growth requires the network hardware to be re-connected swiftly and the software to be re-configured fast. Networks should cope with unusable or faulty nodes, users or carriers. Safety and any fault tolerance or recovery should cut-in quickly.

Network nodes may be required to support users with a variety of intelligence, from an on-off switch to a supervisory computer, and using different analogue and digital signal protocols. Different networks may be needed in the same factory due to long communication distances or differing operating requirements, so gateways or other relaying devices would be needed.

Common topologies for physically distributing information are: point-to-point, ring and bus networks, refer to Figure 3-14. For long distances, or many nodes, point-to-point is inefficient, in terms of cabling costs and transmission times, compared with ring and bus networks.

3.5.2 Models of Communication and Synchronisation

The method of synchronisation and communication has a large effect on the performance of the DCS. Concurrent processes [Ben-Ari 1990] need only interact: where there is contention for a shared resource, or to communicate and share information. When two computers, connected to the same printer, want to print a document, and the first has access, then the second should be prevented from printing until the first is finished. If no such 'mutual exclusion' was in operation, then the printer's output would be a mixture of both documents. Two tools, usually provided by a computer's operating system to provide mutual exclusion, are semaphores and monitors. Schedulers, arbiters and flags can also provide mutual exclusion, but are not atomic - requiring other components and/or processor time.

Communication can be synchronous or asynchronous. In synchronous communication, all processes must stop processing at the same time to communicate, and means that all except one process are waiting for the other, whether sending or receiving. In asynchronous communication, an intermediary can obviate the

need for the other processes to stop and wait. This is so when sending, but a receiving process will have to wait if the information is yet to be sent and can not proceed without it. These are analogous with telephone and e-mail communication respectively, where the intermediary of e-mail is a buffer (stored file) which is usually waiting to be read by the receiver.

There are several communication models, and include: point-to-point, master-slave, client-server and producer-consumer models. Point-to-point communication is usually peer-to-peer, synchronous, unbuffered, uni-directional, connects two processes and closely mimics its hardware implementation, so is simple to provide. It is possible for such communication to preclude identification of sender, receiver and channel, like an intercom, but processes often communicate with more than one process so identification is needed, as in dialling the receiver's telephone number.

In the master-slave model, the master determines when the slaves may send or receive, and the roles do not change. In the client-server model, processes communicate peer-to-peer, so users can take either role. To communicate, the client requests a service from a server and the server acts on it and returns the result to the client. The producer-consumer model allows peer-to-peer communication with role reversal, but employs a buffer to provide asynchronous communication. The producer adds to the buffer, while the consumer takes from it. However, the producer can not add to a full buffer, and the consumer can not take from an empty one, either case requires the user to wait. To compare client-server and producer-consumer models, consider node A reading from node B, in the examples below.

In the **client-server** model of Figure 3-15, clients 'control' and servers 'serve', so to read, node A becomes the client and transmits a 'read request' to node B (the server). The server receives the message as a 'read indication' and deals with it when possible, and returns the reply as a 'read response'. The client receives the message as a 'read confirm', and processes it when ready.

In the **producer-consumer** model of Figure 3-16, node B has information wanted by node A, so node A becomes the consumer and B the producer. To read, the consumer places a 'read request' from node B in its buffer. Copies of the buffer are distributed to buffers of all nodes, but only the addressed node (the server) will read it. When the producer has processed the request, then it places the information in its buffer. Copies of this 'read response' are distributed to buffers of all nodes, but only the addressed node (the consumer) will read it. Broadcasting or multi-casting of the same information is possible.

In real time applications 'latencies' become important. The time taken: for messages to be transmitted between nodes, for messages to be processed by the users, for network services between transmission and user processing (stack latency) and the time taken for waiting for access to the network. Implementations exploit the features of both communication models. OSI and MMS (see below) use the client-server model, while FIP (see below) uses the producer-consumer model. The main differences are that: the network services in a 7 layered stack are endured four times in a client-server 'read', but only completed twice in a producer-consumer 'read'; and broadcasting the same information in the producer-consumer model is inherent, but the user in the client-server model must send separate messages to each server and hope the information remains valid for all servers.

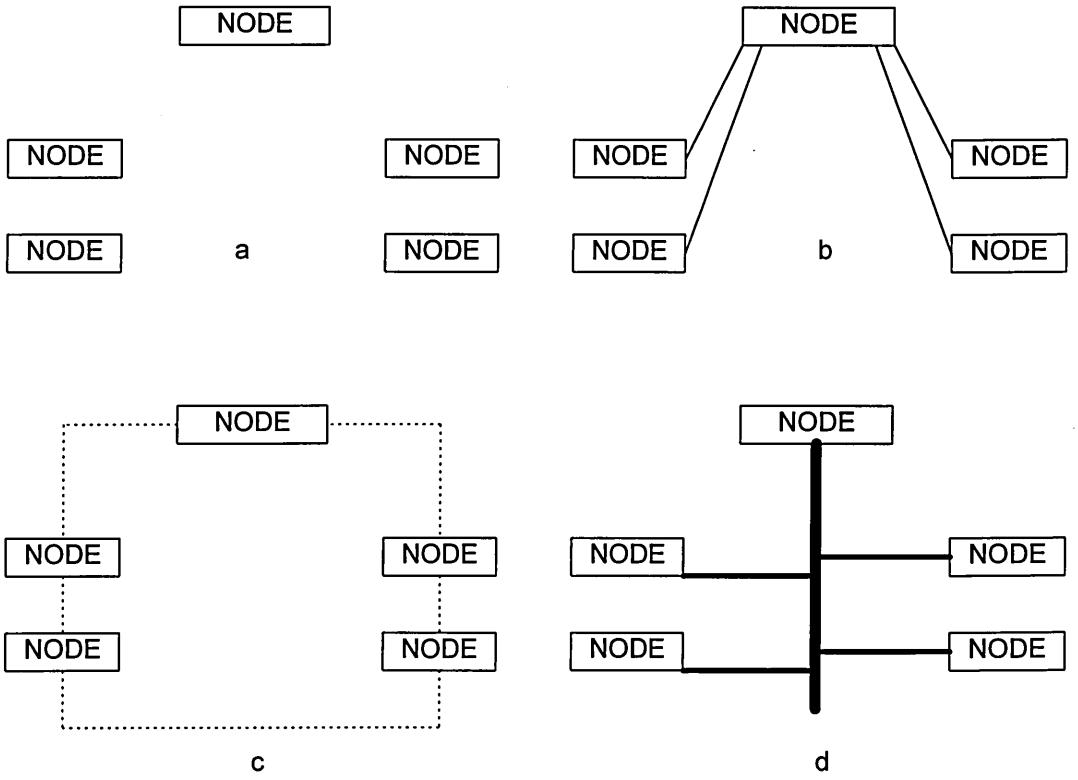


Figure 3-14 a) 5 nodes connected by b) star c) ring and d) bus.

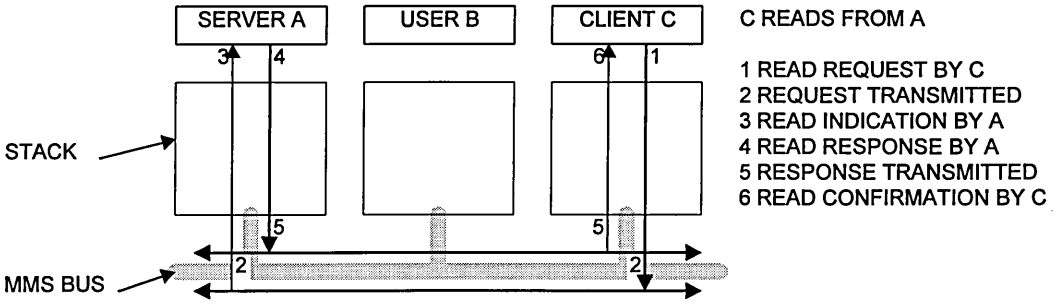


Figure 3-15 Client-server model of communication in MMS

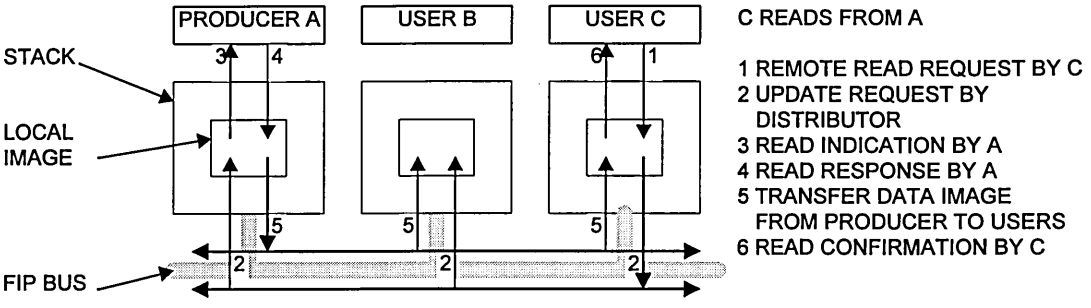


Figure 3-16 Producer-consumer model of communication in FIP.

3.5.3 Communication Standards

The OSI reference model [BS-ISO 7498 1984] is a third hierarchical structure (the others are the control and the communication of control, in section 2.4), and is an international standard for the inter-connection of communication systems (hardware and software). It provides a standard interface for manufacturers to adopt and guarantee connection and communication with corresponding systems.

The reference model sits between the application software and transmission media of both sender and receiver. The application software and the media can be different, such as sending information from a database on node 'A' to a spreadsheet on node 'B', where node 'A' is on a network with a twisted pair transmission medium and node 'B' is on a radio network. The transmission must appear as simple as sending data to a directly connected printer.

The reference model consists of 7 hierarchically modular layers, where a layer uses the services of its immediate subordinate. Each layer adds information when sending, and uses information when receiving [Kirk 1991]. When data is transmitted from one user to another, 'information-envelopes' start being added at the layer closest to the application software (layer 7), and information-envelopes continue to be added until reaching the physical transmission medium at the bottom layer (layer 1). The envelopes provide sufficient information between communicating nodes and any relaying devices, refer to Table 3-8 and Figure 3-17.

Layer	Function
physical	handle transmission characteristics, including encoding;
data link	route messages, detect errors and re-transmit to nodes of the same network;
network	route messages to nodes of other networks;
transport	provide high data integrity by returning acknowledgements and error recovery, and enable multiplexing and channel identification;
session	provide two way concurrent communication, synchronisation and channel naming;
presentation	re-format and translate data into standard formats;
application	provide users' application software with their communication requirements

Table 3-8 Function of the 7 layers of the OSI reference model

Relaying devices [Weston 1987] are either physical connections or nodes which are common between networks, refer to Figure 3-18 and Figure 3-19:

- Repeaters connect the same type of physical medium and perform layer 1 functions. They enable extensions to networks which otherwise would suffer distortion.
- Bridges can connect different networks (provided frames and addresses are consistent) and carry out functions of layers 1 and 2.
- Routers are nodes that connect quite different networks at layers 1 to 3.

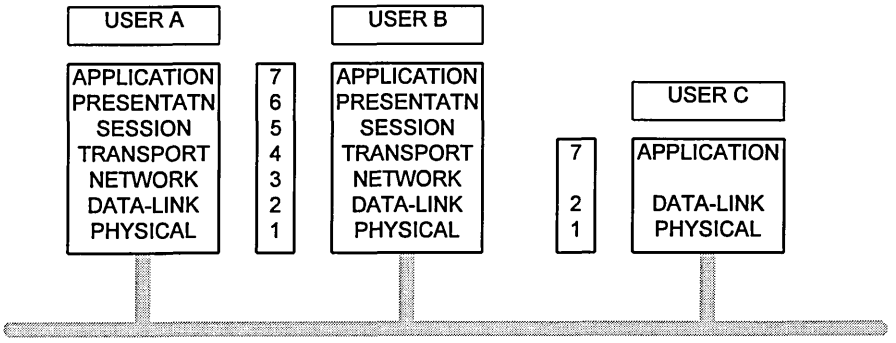


Figure 3-17 The full 7 layer OSI and cut-down 3 layer Fieldbus models

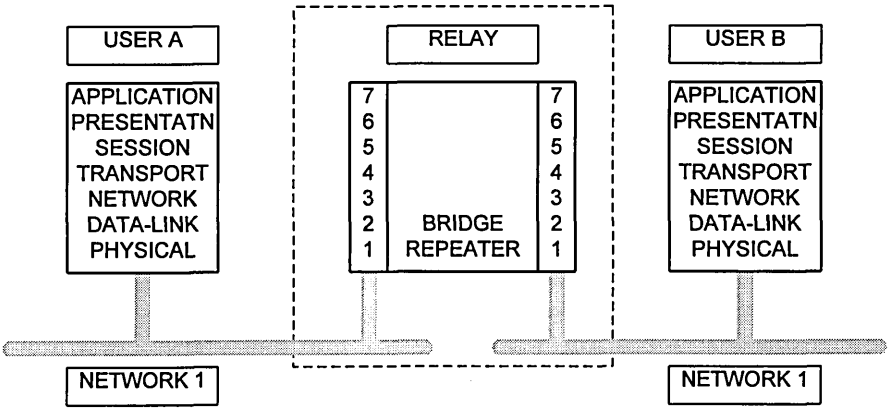


Figure 3-18 A network extended by a bridge or repeater relay, at layer 2 or 1

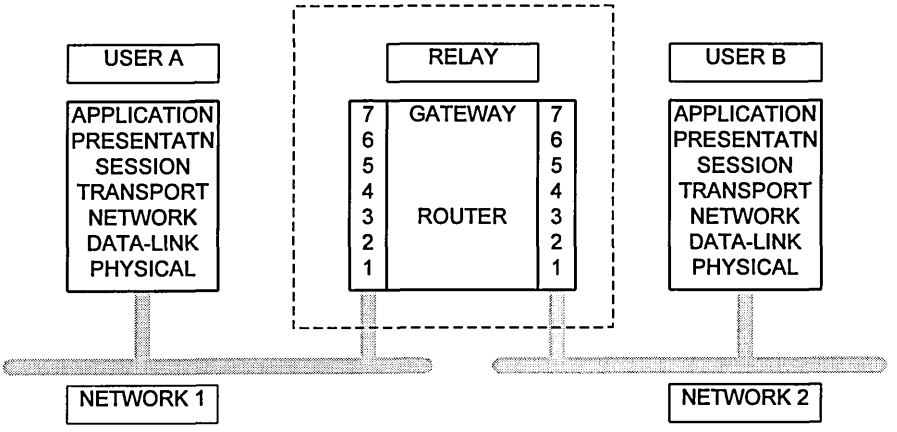


Figure 3-19 Different networks connected by router or gateway, at layer 3 or 7

- Gateways are nodes that connect different computer systems. An OSI/non-OSI gateway must convert all seven layers with the equivalent of the non-OSI system.

General Motor's MAP is a full implementation of OSI, and specifies permissible standards or products for each layer. For example, it allows ISO 8802.4 token passing bus at layers 1 and 2, which transmits a minimum packet size of 20 bytes at 10 Mbit/s [Linge 1986]. However, after the envelopes for each of the seven layers are added, then data rates typically fall to 20 kbits/s [Weston 1987]. This may be acceptable for office communication, but is not for many shop-floor requirements where a few bits of data may have to be sent at 5 Mbit/s, such as DSP and safety operations. Alternative standards are being developed for shop-floor applications, such as Fieldbus and Mini-Map.

3.5.4 Shop-floor Communication

Where Mini-Map and MMS (see section 3.5.7) seem to have been developed top-down from Map and OSI respectively, proprietary Fieldbuses (see section 3.5.6) seem to have evolved bottom-up from remote sensor and actuator control. They are now competing for adoption as standard for shop-floor control (discrete and/or process manufacture). The difficulties from both approaches [Kirk 1991] are: that international standards of open systems in real time shop-floor control have ignored timing (as in MMS), or have yet to decide on an implementation (IEC SC65C and ISA SP50); that groups of companies want their own evolutionary different Fieldbus version to be the adopted standard (e.g. WorldFIP, ISP and FINT [Reeve 1993]); and that many of the sponsors of the standards are these companies. The most common features of the Fieldbuses and Mini-MAP are: a collapsed stack with 3 of the 7 OSI layers (physical, data link and application); and the partial or full adoption of MMS as the application layer.

The Fieldbus model uses simplifications of layers 1,2 and 7 of the full reference model, refer to Figure 3-17. It was developed for use between sensors, actuators and other low complexity but intelligent industrial devices. Layers 3 to 6 of the full OSI model provide inter-networking and other services, so their absence prevents Fieldbus users communicating over other networks, and layers 2 and 7 must interface. Some advantages of Fieldbus are low cost, fast and efficient data transfer, inherent safety, a range of transmission media and the ability to balance between number of users, transmission distances and transmission speeds.

Benefits of Fieldbus over conventional point-to-point control are the savings in cabling costs and space, self configuration and diagnostics, easier expansion and assured connection of all compatible products, as revealed at the end of section 3.5.1.

3.5.5 The Transputer and Occam

The transputer and occam were developed together, and are used in various industries for embedded systems, numerical analysis, image processing and real time control [Inmos 1989b].

3.5.5.1 Transputer Reliability

The transputer and occam were developed together from the mathematical notation 'communicating sequential processes' (CSP) [Hoare 1985]. Consequently, there can be confidence that the design of the transputer is correct.

All transputer products undergo a rigorous quality assurance process [Inmos 1989a], but choosing transputer hardware that conforms to standards, such as [Mil-Stan 883C 1990], will improve confidence in hardware reliability.

3.5.5.2 Transputer Communication

Communication between transputers is simple compared with the other networks. The transputer, the 'computer on a chip', consists of communication, processing and memory, which operate concurrently. Transputer communication is managed by (four or two) link interfaces, and information is sent over transmission-links which consist of two uni-directional wires. All messages to be sent are directed to one of the link interfaces (determined by configuration, below). A link interface cuts the message into byte (8 bits) sizes, and envelopes the byte with the protocol, transmits the data packet over one of the two wires and receives the consequential acknowledgement in the second wire. As soon as the link interface receives the acknowledgement, it knows that the sent packet was received and that the other transputer is capable of accepting another packet, so it can envelop and transmit the remaining parts of the message. The data protocol consists of a start bit, a one bit, the 8 data bits and a stop bit. The acknowledge protocol is only a start bit and a zero bit. Other protocols are the link interfaces are TTL compatible (like the rest of the transputer and other computer products), and transmit at 10 Mbit/s (5 and 20 Mbit/s are possible). Transputer communication does not comply with OSI, but transputer products are available to interface with IEEE 802.3 Ethernet (IMS B431), IEEE 488 GPIB (IMS B421), RS422 differential link buffer (IMS B415) and other communication standard products.

3.5.5.3 Transputer Communication Reliability

Communication over links [Inmos 1989b] is asynchronous at bit level, precluding accurate distribution of clock pulses. Synchronisation happens at byte level with the 'send message with acknowledge' protocol, and enables communication with slow devices but at high transmission speeds.

Links are TTL compatible, and are intended to connect with other transputers over short distances (<30 cm) via back-planes (e.g. B008, below) or wire cables (e.g. twisted pair). A link length of up to 10 m is possible if 'series termination' is used, and up to 20 m if buffered. Effects caused by electrical noise in a manufacturing environment and other noise generated as signals travelling in wires, and wires close together, can be reduced if the wires are screened. For longer distances, it is best to use transputer products such as IMS B415 and B431, where lower transmission rates are specified. Optical fibres are immune from such electrical noise and can transmit over thousands of metres.

The simple link protocol and selection of appropriate physical transmission medium ensure high transmission reliability, but no checks are made as to the data integrity. This provides an immense saving

in transmission times, because transmitted packets contain no checking information, thus are smaller and no processing is needed to check. Where information is corrupted during transmission, but the packet remains legal so the acknowledge is returned, then the data is compared with the expected data-type and will STOP the receiving process. (An occam STOP halts only the issuing process.) The next transmitted packet will STOP the sending process when no acknowledge is returned. If a data packet is not received, or is illegal, then no acknowledge is returned and both processes will STOP. Library routines are provided to help recover from communication errors, either by 'time-outs' or by 'killing' the channel, followed by resetting the channel.

3.5.5.4 Occam and Transputer Communication

Communication of two occam processes on different transputers is logically the same as if they where on the same transputer. Where communication between occam processes on the same transputer is done by re-scheduling, on different transputers it happens when the acknowledge is received. Only when the final acknowledge of the whole message is received, can the sending process be re-scheduled (become active).

Occam code can be written to be run on one transputer or a network of transputers. Two alternative software development tools are the Transputer Development System (TDS) and iTools, and two hardware platforms are the B008 and the TEK:

The IMS B008 is an IBM PC plug-in daughter-board, and provides four features:

- communication with the IBM PC mother-board (and on to screen, keyboard, disks, etc.);
- support for up to ten transputer modules (TRAMs);
- a 32-way programmable crossbar switch to allow the interconnection of links to be re-configured; and
- a 37-pin connector for eight communication links and four system-links to communication with other transputer products.

The CSA TEK (transputer education kit) is an IBM PC plug-in daughter-board, providing three features:

- communication with the IBM PC mother-board
- a transputer with four or two links
- four or two DIN connectors for links to communicate with other transputer products

TDS and iTools are essentially the same, but TDS is an integrated windows environment and iTools runs from the host computer's operating system. They require a transputer with at least 1 Mbyte of external RAM, even to write code which can run on the transputer's internal RAM.

3.5.5.5 Booting

The memory on a transputer is volatile, so does not hold data or code when power is removed. Once power is returned the transputer is ready to receive code through one of its links or from external ROM (non volatile memory). This initial process, 'booting', loads configured code onto the transputer or transputer network. Usually, code is written, configured and booted via the development tools from the host computer. However, code is booted from ROM in embedded systems, where changes are rarely if ever made.

3.5.5.6 Configuration

Configuration maps processes to processors and channels to links. There is no logical limit to the number of processes that can be placed on a transputer, but the major restriction in configuring a transputer network is that transputers only have four or two links. Programs can be written and tested on a single transputer, before work is partitioned onto the transputer network. In this way, the logic of the program can be made correct before task distribution is considered. Configuration code merely specifies which compiled processes will run on which transputers, and which channels will physically communicate between transputers over the four links. The actual allocation is made when the network is booted, where code is distributed from the root transputer (the transputer connected to the host), booting the other transputers through the links.

The IMS B008 supports up to ten transputer products, and has a programmable crossbar switch (C004) which enables their links to be theoretically re-connected by control software. The code controlling the C004, written using 'Mother-board Module Software', and the configuration code must be consistent. Re-configuration of a transputer network can be done swiftly. Re-configuration of a transputer network consisting of CSA TEK boards and no programmable link switch is much slower, because the links must be re-connected manually. Such a network is beneficial where the transputers are located with the machines they control.

Transputer network structures are restricted by the number of transputer links and determined by the application. Transputer DCSs are often configured into a master-slave (e.g. processor farm) or ring arrangement.

3.5.5.7 The Transputer and the OSI 7 Layer Reference Model

Occam and the transputer inherently provide several of the OSI specifications. These are returning acknowledges (layer 4), and point-to-point communication, synchronisation and channel naming (layer 7). Other commonalities are possible through occam programming, such as message routing, error detection and re-transmission (layer 2), and error recovery and multiplexing (layer 4). Features common to all transputer products are: physical communication characteristics (layer 1), standard data formats (layer 6) and user application software, written in occam, runs on all transputer products. Connecting transputer networks with other networks or other computer equipment (layer 3) is possible (e.g. Ethernet, above), but compilers can not check for integrity of non-occam (or non-transputer based) programs. The services and information available in the 3 layered model and MMS can be represented in occam. The abstract point-to-point communications between two MMS users, semaphores and positive confirmation are inherent in occam channels and transputer links.

Time critical communication between transputers and real time processing on transputers are made possible by the transputer's on-chip clock, scheduler and link interfaces. These components, and the high and low priority process queues, are only visible in occam, as channels, high and low priority processes and high and low priority timers.

3.5.6 FIP (Factory Instrumentation Protocol)

[FIP] is a bus based network with a bus arbitrator and an emphasis on timing. FIP adopts an extension of the producer-consumer communication model, the producer-distributor-consumers model, where the distributor (arbitrator) provides broadcast, multi-cast or point-to-point communication. The use of the image-buffer (see below) enables correct data to be sent to one or more users, and calculated global synchronisation is possible.

At the application layer all users are treated as objects, and are addressed or identified by a unique object name. Each network node understands the object name and abilities and communication requirements of its user. Network management services include a database of all users' information. The multi-cast communication and database facilities at the data-link layer allow an application layer message to consist of only two data-frames, i.e. source address and function (command or data). The consumers can make full use of these two pieces of information with the help of the database. The succinctness of such messages and the minimum data-frame size of 24 bits improve communication time.

The image-buffer sits on the top boundary of the OSI data link layer, and has space for a copy of the variable and a validity-timer. When sending, the producer writes a copy of the variable to the image-buffer and the timer is set (to count down) with the 'production period'. If the distributor is invoked before the timer zeros, then it transmits the image and a 'transmission period' to the image-buffer and validity-timer of all consumers. If the consumer's image-buffer had received the message within the 'transmission period', then the consumer can read the valid variable from its image-buffer. The distributor is invoked by a producer, a consumer or by a third party. When a consumer wants a variable, it must first issue a 'request for distant reading', and the relevant producer will respond. Image-buffers can provide 'indications' which allow producers and consumers to confirm or synchronise the start or end of a transmission. This form of acknowledge and that conforming to ISO 8802.2 are optional.

Real time operation is possible through adding the production, consumption and transmission periods between producer and consumers to the processing times of the users.

The physical layer defines transmission speeds of 1 Mbit/s, 31.25 kbit/s and 2.5 Mbit/s over twisted pair, shielded pair and fibre-optics. Bits are encoded onto the transmission medium in Manchester II code to allow time and data to be carried together. The data link layer is not concerned with time, but it implements other concepts from the application layer discussed above.

Fault tolerance and reliability are contributed by all three layers. The physical layer provides for frame delimiters and cyclic redundant code to achieve an error rate of one in a thousand over 20 years of service. Acknowledges are offered at the data-link layer. At the application layer, validity timers in the image-buffer are checked to ensure valid data leaves the producer's buffer and arrives at the consumer's buffer, and either the producer or consumer can ask for fresh data. Where validity timers indicate invalid data, then the invalid data is removed and a copy of the original data is reset.

FIP and the OSI 7 Layer Reference Model

FIP is a Fieldbus, so does not support layers 3 to 6. It is an integrated design, cored by the producer-consumer model of communication to achieve real time determinism. This contrasts with the client-server communication and the lack of timing in MMS, as such FIP is more suited to periodic traffic while MMS is more suited for aperiodic traffic. FIP provides most of the MMS services and a gateway to OSI.

3.5.7 Mini-Map with MMS

Mini-Map [Weston 1987] specifies the three layers of the collapsed reference model. Layers 1 and 2 use token passing bus [IEEE 802.4 1984] with 5 Mbit/s carrier-band transmission and connectionless and acknowledged connectionless data-link control (compared with MAP which adopts connectionless data-link control only, but 10 Mbit/s broad-band and 5 Mbit/s carrier-band transmission). Single band transmission reduces cabling, connection and node interface costs. Although a slower transmission speed improves safety (by less heat generation), response times are relatively high at about 20 ms [Armitage 1988]. The top of the three layer model adopts [MMS 1990], which is virtually the same MMS used in layer 7 of the full reference model.

The MMS protocol only specifies the message construction and communication in the application layer. Users, of varied intelligence, are likely to involve NC, PLC or robot control, so the MMS standard includes specifications of these in addition to a common core.

MMS allows point-to-point communication between: machine and machine, computer and computer, and computer and machine, and adopts the client-server model of communication to do it. Computers tend to be the clients, and the items of shop-floor equipment tend to be the servers. A client can request or manipulate information of a server, while a server can only report information to a client. Information can include: variables, programs, semaphores (for synchronisation), statuses and human interaction. A service, such as a request from the client, will return a positive confirmation and the information, if successful, or a negative confirmation, if not. Some services do not require any conformation, such as emergency or unimportant data.

MMS only refers to function and not implementation, so consequently there is an absence of timing requirements. [Grant 1990] MMS does not have its own programming language, but its 'object-orientated' specification suggests code would be best written in SmallTalk or another object-orientated language.

3.5.8 9Tiles

The LAN is called Superlink, and is a buffer insertion ring network [9Tiles]. It is unlike a 'token ring' network, where nodes have to wait for a token before they can transmit [IEEE 802.5 1984]. However transmission is similar to a 'carrier sensing bus network' [IEEE 802.4 1984], where nodes will not begin to transmit if a transmission is sensed on the network. Ethernet, a carrier sensing bus network, adopts 'collision detection' to resolve the problem of two nodes starting to transmit at roughly the same time. However because Superlink is a ring network, contention for transmission will only occur between the

node's own transmission and transmissions which the node must send on from its previous node. To overcome this contention, when nodes transmit they buffer the incoming transmission. To receive data, the node will not send on the transmission. If no node receives, then the sending node recognises its transmission and removes it from the ring. A message consists of a start bit, a one bit, eight data bits and a stop bit, while an acknowledge is a start bit and a stop bit.

Connections request packets are broadcast around the ring, while data packets are addressed directly to the destination node. Ring speeds of 250 kbits/s and 1.5 Mbits/s are possible. Further details are in section 2.6.

9Tiles and the OSI 7 Layer Reference Model

The 9Tiles LAN consists of Superlink hardware and SimpleNet software. One of its four modes of operation is 'OSI mode', which conforms to OSI layers 1 to 4 and some of layer 5, and supports clock rates of up to 500 kbit/s over 9Tiles' ISO gateways.

Other modes of operation provide suitable communication for a full range of shop-floor equipment. Dumb, VDU and intelligent modes are for actuators/sensors, RS232C and personal computer standards of equipment. But none is compatible with the 3 layer model and MMS due to the absence of layer 7.

3.5.9 A Comparison Shop-floor Control Systems

The benefits of a complete and appropriate communication standard include:

- guaranteed connection between compatible equipment (connectivity);
- invisibility and simplicity of communication;
- suitable capabilities of function, including real time and dependability;
- a core philosophy that enables future modifications.

MMS and Mini-MAP, 9Tiles, FIP and the transputer and occam network all claim to be suitable for shop-floor control, but how do they compare? The first two criteria, above, are implementation dependent and the second two are specification dependent.

MMS is the standard to which the other systems work and work around. It is an open protocol, confining itself to message formats for all potential manufacturing equipment, and does not specify implementation. As a consequence, MMS does not accommodate time critical requirements [Kirk 1991]. Its client-server communication model precludes multi-casting such as global time synchronisation. The MMS core specifies how new manufacturing equipment can become compatible with MMS, but because it is all embracing it appears very complicated. The token passing bus of Mini-MAP presents a response time of 20 ms, which is too slow for many shop-floor needs. It is an OSI recommended network for manufacturing, and its common usage implies its suitability.

Transputers and occam were designed together to facilitate the communication between concurrent processing elements. This philosophy, based on CSP, is universal throughout transputer products. The communication convention is the data packet and the acknowledge transmitted by, and controlled by,

four or two 10 Mbit/s serial transputer link interfaces, and appears as occam channels in software. The on-chip clock enables time critical performance, but only two priority levels are provided. Gateways to other networks are available. Fault tolerance is not inherent, but can be programmed in occam.

Real time and time critical performance is possible with 9Tiles' buffer insertion ring network, but programming this capability in application software is very complicated. Communication is by no means invisible and simple at the application level. Programmers are not required to learn a network-language and can use their favourite language, e.g. Pascal. The lack of a network specific language and programming environment, and the reliance on office based personal computer technology, may lead to the difficulties in implementing real time and communication in manufacturing control software. The network does not follow any of the communication models described in section 3.5.2, is not compatible with either the 3 or 7 layered models and was not designed specifically for shop-floor use.

FIP is an integrated design, was developed for controlling remote actuators and sensors and is MMS compatible. It provides complete time determination, and it allows periodic traffic to be compiled to reduce response times to under 5 ms. Fault tolerance is implemented invisibly in all three layers, but can be altered in the FIP programming language.

Where programming languages and development environments are not provided, then DCS designers can choose their most favoured language, however this could lead to the choice of a relatively unsuitable or unsafe language. It is likely that code written in an appropriate language would provide the necessary timing and communication features thus be more succinct, easier to read, so easier to maintain and less likely to introduce errors. An environment providing suitable tools to check real time, concurrency and communication would be able to test for errors or even help to develop the code. FIP and transputer/occam include languages, but 9Tiles and MiniMAP/MMS do not.

All options can cope with the effects of a manufacturing environment. All can provide fault tolerance, either as part of the network services (MiniMAP, FIP), the physical layer (all), data link layer (all but transputers), application layer (MMS, FIP) or in the manufacturing control application (all).

Connectivity with manufacturing equipment (machine tools, work handling, sensors and actuators) and the 7 layered reference model is varied. Transputers mainly connect with other transputers, but interfaces are being developed. The origins of FIP ensure connection with nearly all manufacturing equipment and lately to OSI. 9Tiles' connectivity is improving, and MiniMAP's has matured.

It is clear that the 9Tiles network is the least suited shop-floor control option, but it is less obvious which is best. The slow response time excludes MiniMAP/MMS as best and FIP would be best if it treated aperiodic traffic as well as it treated periodic traffic.

The transputer/occam option is very simple, general purpose and has a mathematical basis, but lacks a history in shop-floor control, but overall seems suitable.

3.6 Similarities Between Petri Nets and Occam

Occam can be related to Petri nets quite closely. Occam's prime concepts: concurrency and communication are reflected in Petri net places and transitions respectively. Petri net places and interactions can be translated into occam processes and channels, refer to Figure 3-12. Of the four principle Petri net constructs, only the non-deterministic 'output OR' has no occam equivalent. Figure 3-13 shows how the Petri net 'output AND' and 'input AND' can be modelled as the occam PAR, and 'input OR' by the occam ALT.

Much of occam can be modelled in Petri nets [Carpenter 1987], and restricted Petri net models can be coded in occam. Such equivalences are necessary at the 'micro' level for modelling occam processes, but equivalences are needed at a 'macro' level to be useful in modelling and are described in chapters 5 and 6.

4. Distributed Control Development

4.1 Introduction

Chapter 2 described FMCs and the manufacturing requirements of the example FMC. Chapter 3 discussed the tools and techniques used in the methodology, and identified their appropriateness for developing flexible and dependable DCSs for use in FMCs.

This chapter discusses the development of dependable and flexible distributed control, in general, and to achieve comprehensibility. It begins by identifying the state of the art in software engineering for parallel systems. It shows that formal methods can improve confidence, achieve correctness and can be applied to all but the requirements stage of software development, and contrasts them with their disadvantages. It discusses the benefits of Petri net graphs and how pseudo-code can augment Petri net graphs to improve comprehensibility. The chapter ends by making conclusions from it and previous chapters.

4.2 Software Engineering for Parallel Systems

The special issue of the journal edited by [Jelly 1994a] presents various relevant papers, including a review of the current state of the art in software engineering for parallel systems.
The editorial identifies a set of problems concerning the use of parallelism, which are in addition to those associated with other computing tasks:

- identification of problem domain and solution domain parallelism
- incorporation of parallel activities in specification and design
- architectural influences on design and implementation, including use of virtual machines
- correctness and testing of parallel systems
- performance prediction, monitoring and evaluation of parallel systems

Some assertions are made:

- there is a slow acceptance for multi processor systems for general computing
- because of the unavailability of porting existing software and developing software
- there is much research into particular areas of software design and implementation
- there is a lack of satisfactory development methodologies, techniques and tools

Areas of future research are proposed:

- new development methodologies of parallel software systems
- applicability of existing software engineering methodologies for construction of parallel systems
- theoretical issues of specification, verification and validation of parallel software
- application of formal methods to parallel systems
- prototyping, testing and performance measures of parallel systems
- impact of parallel languages and architectures on development techniques
- CASE tools and run time support environments for parallel software construction
- software reuse technology for parallel systems

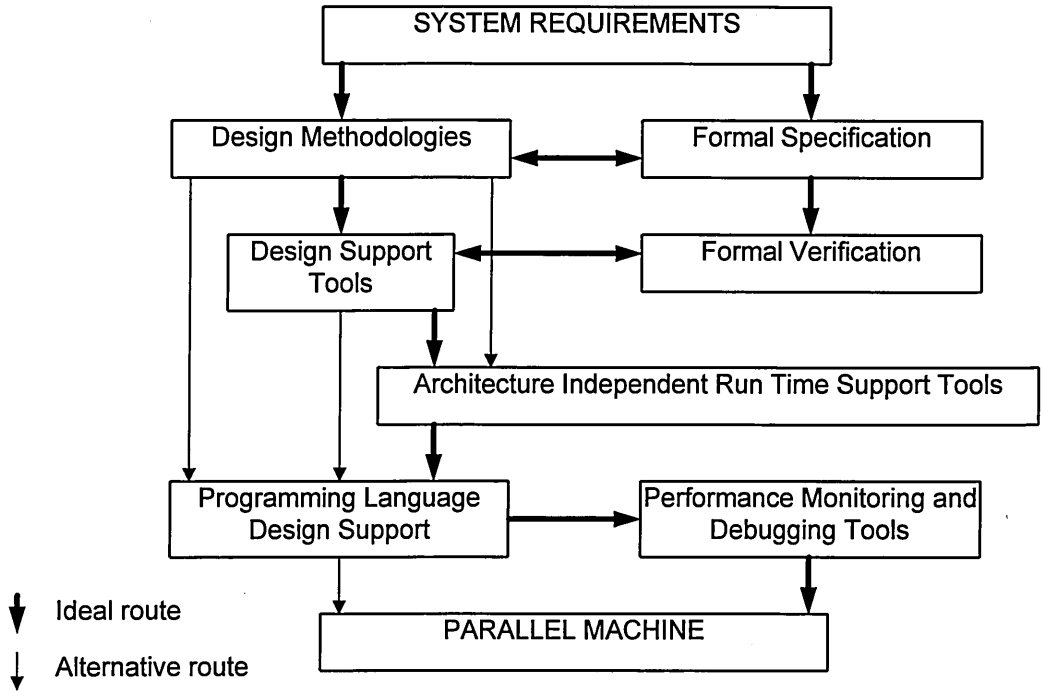


Figure 4-1 Parallel software development life cycle [Jelly 1994b]

Based on the papers in the journal, [Jelly 1994a] makes two conclusions relevant to this thesis:

- integration of formal analysis techniques in a development environment is a more conducive approach to adopting formal methods in parallel software engineering, than is following the formal life cycle (see Figure 3-2)
- Petri nets in their various forms are commonly adopted as formal modelling and analysis tools

4.2.1 A Review of the State of the Art

A review of the state of the art in software engineering for parallel systems [Jelly 1994b] is structured into: development, formal methods in development, design methodologies, CASE and support tools and performance.

Development is generally done in two stages:

- decomposition of the problem into suitable sub-tasks
- mapping the sub-tasks into hardware

This allows the function and parallelism, and other considerations such as safety or timing, to be completed before hardware constraints are applied. This enables the production of secure, well structured verifiable software, and the development of reusable and portable systems. However, such performance characteristics often are of significant influence or are the purpose of the system.

Similarly, software development life cycles for sequential and parallel software cannot be the same. Figure 4-1 shows the parallel software development life cycle, where formal specification and verification methods complement rather than replace design methodologies and tools. (Compare this with the formal life cycle of Figure 3-2.)

Examples of **formal methods** for parallel development are: "trace theory, process algebras, state based models and Petri nets, and have led to a number of formal language systems e.g. CSP, CCS and unity". A paradox exists: There is a stronger case for using formal methods in parallel software development, because parallel systems are more complex than sequential. However, formal methods are not adopted because of the "perceived difficulty in learning these techniques", and that current support tools cannot cope with such complexity.

Petri nets "are the only formalism which is widely used to model and reason about the behaviour of real parallel applications, and many parallel software development systems have incorporated one of more Petri net variants". They can be used in system specification, design, verification and validation.

Some Petri net issues mentioned are:

- the Petri net representations of behaviour closely models either the algorithmic or data flow models of computation
- Petri net graphs are intuitive, and marked Petri net graphs can show dynamic properties of a system
- Petri nets do not provide support for good software structuring of large systems
- stochastic Petri nets have been produced from occam, Ada and concurrent c for performance and structural analysis

Design methodologies must provide notation to represent processes, data structures and interactions, and offer guidance in partitioning the system sub-tasks, and in management for large and complex systems. "Graphical techniques seem to provide an appropriate way of managing this complexity", though solutions are biased towards those which are easily drawn. Examples of graphical techniques are data flow diagrams, entity-life models, object oriented models, MASCOT diagrams and entity-structures.

CASE and run time support tools provide computer assistance in one or more of the stages in the software life cycle, either as a largely independent software engineering environment or as a run time toolkit utilising the underlying processes, synchronisation and data access parallelism of the operating system. Their major disadvantages are in the narrow options in design tools and target hardware.

The barrier to the adoption of parallel systems is the immaturity of CASE as a technology, which is caused by the lack of commonly accepted and used development methodologies on which to base the CASE tool. Ideally CASE tools would include architecture independent modelling, analysis and verification tools and automatic code generation for a range of target architectures. Of the CASE examples given: VERDI is a graphical design language based on extended Petri nets for prototyping and system simulation; and TOTAL translates Ada code into a Petri net for reachability analysis.

Run time support systems separate the language from the underlying architecture, thus enabling portable and scaleable software production through the concept of a "parallel virtual machine". Again, there is a trade-off between flexibility and performance. Systems allow the separation to vary, so the user has more control of hardware. Of the examples given: Helios and Trollius are operating systems that provide automatic load balancing, message buffering and routing on transputer networks, but control of these is allowed if desired. PVM and Express are library routines that can be accessed by concurrent c and FORTRAN, but PVM is UNIX based and Express is portable between Intel, nCUBE and transputer based architectures.

Software performance engineering is another framework for parallel software development. It is based on, and supported by, tools and techniques to achieve performance objectives, which either monitor or predict performance.

Performance monitoring gathers information about hardware and software in real time, so must effect results as little as possible. Of the examples given: TRAPPER (refer to section 4.7.7.3) is a programming and monitoring tool that provides processor usage and process activities for embedded transputer architectures; and TOPSYS's PATOP provides performance data at system, node and process levels for various MIMD (including transputer) architectures.

Performance prediction does not need the code to have been written before performance can be considered. Predictions can be made via analysis, simulation or rapid prototyping.

Prediction through analysis is straight forward if mathematical tools were chosen earlier, e.g. queuing theory and stochastic Petri nets. GreatSPN is a graphical editor and analysis tool for time and stochastic Petri nets that can check performance.

Prediction through simulation is possible before the parallel code is written, e.g. in calling DEMOS simulation libraries in Simula, or after the code is written, e.g. constructing stochastic Petri nets from occam code for subsequent analysis (refer to section 4.7.2.1).

ALPAS is an example of performance prediction through rapid prototyping. The user specifies program, architecture and mapping characteristics and ALPAS generates the program, from which functional and performance characteristics can be identified.

The review makes two conclusions:

- "there is a need for techniques and tools for the construction of reliable, portable and scaleable software systems"
- this need is not adequately satisfied, but there is much research in this area

It also highlights issues to address:

- the role of CASE tools
- the usefulness and rigour of their underlying methodologies
- "the benefits of incorporating formal methods into the development life cycle"

4.3 Concurrent and Distributed Processing

It is useful to define some vocabulary concerning sequential, concurrent and distributed computing. [Ben-Ari 1990]:

- a process is a single sequence of instructions
- a program is a set of processes
- sequential programming involves a single processor, which executes a process
- concurrent programming involves a single processor, which executes a program in abstract parallelism
- distributed programming involves many processors, which execute a program in actual parallelism

[McDermid 1991] describes terms slightly differently:

- a distributed system has physically or conceptually separate components
- a parallel or concurrent system has concurrently operating components
- a communicating system exchanges messages between components which co-operate
- a co-operating, co-ordinating or synchronising system has components which co-operate
- a reactive system interacts with its environment

Distributed and Concurrent Systems Compared

[Alford 1982]highlights a number of advantages the distributed systems have over concurrent ones:

- load sharing - balance the processing load evenly
- resource sharing - equipment, e.g. printer, sharing
- data sharing - access to more than one database
- geographical location - control of equipment or access to data located far away
- logical structure - convenience may imply one processor per process
- reliability - systems made fault tolerant by switching in redundant processors

- flexibility - the ability to add or remove processors

Physical Differences

All parallel systems are logically the same [Alford 1982], but there are differences between distributed a multi-processor system and a shared memory or concurrent system. The communication bandwidth and communication latency in message passing can be significant. For example, the state of the sending process might change by the time the message is received. Such temporal problems depend on the amount of data exchanged between processes, on the distance between processors, and on the amount of local processing on each processor, and models of the synchronisation and communication adopted.

Concurrency, Distribution and Communication

There is a trade off between parallel processes, communication and distribution [Balbo 1992]. Parallel processes that are currently active are best performed on different processors. Communication between processors is relatively very slow, but communication within a processor is much faster. If distributed processes communicate a great deal, then they may be more efficiently be made to run on the same processor. However, when this happens, only one of the communicating processes can be running at a time. The gain in communication is contrasted with the loss in parallel processing. Such a trade-off is an optimisation problem of communication versus processing costs, which depends on the program's structure, and again on bandwidth, distance, synchronisation and communication model.

Termination and Deadlock

Sequential processes can always be made to terminate [Ben-Ari 1990], refer to section 3.2.2. A concurrent program terminates when all of its processes terminate. Deadlock in a concurrent program occurs when one process prevents the program from terminating, because other processes cannot proceed without communicating with the blocking processes. Detection of deadlock in a concurrent program running on a single processor is simple: the active process queue is empty. Deadlock in a distributed system, running processes of a program on many processors, is the same as concurrent deadlock. However, detection of deadlock by simultaneous examination of all of the processors' active process queues is impossible. Algorithms to detect and locate blocking processes are available. Section 4.10 discusses deadlock in more detail.

4.4 Formal Development

4.4.1 Informal, Formal and Rigorous Techniques

There will always be doubt that a system 'does the right thing' and/or 'does that thing right' (see section 3.2.2.1). This residual doubt can be reduced if appropriate measures are taken. [McDermid 1991] identifies four areas of doubt in Table 4-1.

Area	Doubt
Interpretation	doubt that the specification produced is based on a correct understanding of the requirements and we are placing the same interpretation on this information as was intended by those who wrote it.
Consistency	doubt that the specification is internally consistent, that is, it does not contradict
Completeness	doubt that all the information necessary to describe the system is indeed present
Validity	doubt that the information contained in the specification 'says the same thing' as the requirements

Table 4-1 Four doubts inherent in development specifications

[McDermid 1991] gives four general approaches to design, which lead to increasing confidence:

- Ad hoc design is no design. If the user is the customer and the developer, then the ad hoc design of a simple system is appropriate.
- Informal design (named in contrast to formal design, below) is the conventional approach. The canonical life cycle, for example, includes planning, structure, documentation, requirements, system specification, design, coding and testing. Testing is the major improver of confidence, because the code can eventually be shown to be valid and complete enough.
- Rigorous design consists of 'reasoned justifications of design decisions' and is bettered still if all stages were specified using a formal notation. In this way, each stage can be shown to be complete and consistent and the rigorous overall outline improves confidence in validity and interpretation.
- Formal design consists of many specifications, which start from the requirements specification and ends in the detail design specification. Each specification is a new formal representation of a design decision with a formal proof that the new specification conforms to the previous. The conformance, or verification, is in terms of validity, completeness, consistency and interpretation. This process of building by introducing extra detail and proving that it conforms to what was there before is called refinement.

Residual doubt, although reduced, can never be eradicated, because even formal design only ensures that the detail design and the functional characteristics of the requirements specification conform. Validating the specification of requirements with the customer is necessary.

4.4.2 Formal Methods

The survey by [Austin 1993] provides claims of the benefits, limitations and barriers of formal methods. However, many claims of the limitations and barriers are either refuted or apply equally to other methods, so are omitted here.

Benefits of Formal Methods

A formal specification:

- is unambiguous, which means it: removes ambiguities, highlights incorrect thinking and mistakes, clarifies the processes and management of development, and it can be used in validation with the customer

    - is as easy to understand as a computer program
    - is shorter than a computer program
    - is shorter than an informal specification
    - can be analysed
    - can highlight errors early in development, thus prevent costlier repairs later on
    - describes function, so does not give unnecessary detail, and leaves non-functionality until later
    - can prove properties only possible by formal reasoning
    - is used to prove properties between the program and itself
    - is necessary in safety critical applications
- Implementations can be constructed from formal specifications
- All software can be specified formally
- Large software packages can be specified
- Large commercial projects are adopting formal methods
- Development and maintenance costs can be lessened using formal methods

Limitations of Formal Methods

- Mathematicians are needed to write the specification and perform the proof, which is costly in time and wages
- Not all formal methods can be combined

Barriers to Formal Methods

- Prospective users have been put off by faults, when they were told formal methods guarantee perfect software
- Only mathematicians can use formal methods
- The software development cycle will change, so too the personnel

Classification of Formal Methods

Methods can be classified against six criteria[Roman 1985]:

- Formal foundation. The method's underlying theoretical basis can be described by its formality, analisability and constructability, and can be classified into finite state machines, data flow, stimulus response paths, communicating concurrent processes, functional decomposition, data oriented models, semantic models (denotational, axiomatic and operational), logic and probability theory.
- Scope. The range of functional and non-functional characteristics that the method can represent. Few non-functional requirements can be expressed with high formality, and few functional and non-functional characteristics can be represented by the same method with sufficient appropriateness.
- Level of formality. The degree a specification can be understood by a computer. Formal methods are more comprehensible to computers than is natural language.

- Degree of specialisation. Methods can be created for very special sub-classes of application. The advantages of a specific method over a general method are that they tend to be more appropriate, constructable, analysable, testable and produce specifications which are more comprehensible. However, if the application falls outside of the sub-class, then the method is inappropriate.
- Specialisation area. The reason for defining the requirements specification as consisting of functional and non-functional characteristics specified in terms of properties (appropriateness, etc.), is to enable full understanding of the problem, and provide a foundation for future stages of the development, at the outset. The requirements specification and future stages could be specified using methods specialised for one stage or be appropriate for more than one of the stages. However, "the key to across-life-cycle integration of design activities rests with the ability to relate design and requirement specifications".
- Development method. The discussion has concentrated on the requirements specification of the 'traditional' stages (requirements, design and implementation) of the Development life cycle. Indeed, the method examines each stage in great detail before proceeding to successive stages. This method can be described as a breadth-first approach compared with the depth-first approach of another method, 'rapid prototyping'. The latter method aims to produce a skeleton system as fast as possible, then allow it to be criticised by users, designers and implementers, before it is improved and subject to further criticism and improvement. In comparison, rapid prototyping "seems to lead to less code, less effort, and ease of use", while the traditional method has "better coherence, more functionality, higher robustness and ease of integration". Other methods applied to requirements specification are expert systems and artificial intelligence.

[Roman 1985] suggests that there should be a goal of "establishing a unified formal foundation that could bring together application and design oriented specifications, functional and non-functional requirements, the life-cycle phases, and requirements definition and design activities".

4.4.3 Formal Development Life-Cycle

[McDermid 1991] describes the five development stages of the canonical model of the software life cycle: requirements analysis, system specification, architectural design, detail design and implementation. The first two describe what the customer or users want, and the others describe how the system developers will satisfy the requirements. This model is similar to those in Figure 3-1.

4.4.3.1 Requirements Analysis

The customer's perceived ideas of the system and its environment are gathered. The documentation must represent concurrency and show the way in which elements in the environment and the system will co-exist. Safety critical applications must highlight the hazards to the system, users and the environment. It must be possible to note the customer's ideas, however simple, impossible or uneconomic they seem, and differentiate between the essential and desirable ones.

Even at this early stage the documentation should be structured, even formal, to reduce the level of doubt. After the analysis, the requirements specification cannot be verified, but it can be validated by reviewing it with the customer. However, the customer cannot be expected to understand the mathematical notations found in some formalisms, so either:

- translate the specification into an animatable or a non-formal specification
- adopt an animatable or a non-formal technique from the outset

The formal method used in requirements analysis should represent functional and non-functional (constraints e.g. timing) requirements under normal and extraneous (e.g. fault tolerance) operation. Failure-modes-effects analysis, fault-tree analysis and FOREST are given as examples of recommended techniques.

McDermid outlines Petri nets and states "understanding Petri nets, and relating them to the real world is not very easy, and consequentially the customer cannot easily understand them, and a certain amount of practice and experience is required to understand them if they are to be used successfully". However, this author and many references refute this, see sections 4.2 and 4.7.

4.4.3.2 System Specification

While the requirements specification is concerned with the system, its environment and their interaction in general terms, the system specification deals only with the requirements of the system, but at greater depth.

It is preferable to adopt a formal method, when preparing a system specification, which can specify inputs and outputs implicitly, and is best accomplished using algebraic specification methods, e.g. OBJ and PLUSS. However, the choice of method will depend on its suitability to the application. The system specification can be verified against the requirements specification, but any new additions must be validated against initial ideas in the requirements specification. While verification will probably be done informally, validation techniques, such as animation, specification execution or failure analysis, will be applied to appropriate parts of the system specification.

Many informal techniques enable the system specification to be presented in a structured way, and validate the system specification against the requirement specification in a review with the customer. A number of these techniques, e.g. SSADM and JSD, are applicable to the system specification and later stages in the life cycle.

In comparison, the informal techniques allow the customer to understand the system specification more fully. The customer can improve and correct it, and thus have confidence in what is to be delivered. Formal methods allow validation and some verification. However, the customer is unlikely to understand the formal system specification, and must believe it to be valid.

4.4.3.3 Architectural Design

This is the first stage in the design of how to get the system to operate. It describes the functionality, interfaces and structure of the system, and provides requirements for later stages in the life cycle. The constraints that were identified in the requirement analysis can heavily alter the structure as outlined in the system specification, because current technology cannot perform as demanded. For example, the required reliability might necessitate the use of 'triple modular redundancy', or timing requirements might dictate dividing the work on to two processors.

The same formal methods can be used in the architectural design as in the system specification, i.e. process algebra for the structure and communication and model oriented models for the behaviour. This is also true for some of the integrated informal techniques, and McDermid cites MASCOT3.

In comparison, it is of great significance that formal methods do not express many non-functional characteristics, because (as just stated) they have much influence on the design. Refinement in formal methods does not cope well with large structural changes, nor with loose interpretations of equivalencies between refinement levels. Integrated informal techniques do not suffer from these problems, because they were largely created to transform from system specification to architectural design or to a later stage in the life cycle. However, where formal methods are suitable they enable verification between system specification and architectural design, while informal techniques rely on consistency checks. So the trade-off is between flexibility and verification.

4.4.3.4 Detail Design

This stage transforms the architectural design in to sufficient detail for coding. Data structures, algorithms and software modules should be defined from the structure and interfaces specified in the architectural design.

Although this is the second refinement stage, any flaws concerning the architectural design, found at the detail design stage, should not be repaired, but the error must be returned to the architectural design for holistic correction. Thus structure is preserved between architectural design and detail design, and the transformation is a step-by-step hierarchical decomposition for informal techniques, and refinement for formal methods. Again SSADM is cited as a suitable example to apply to the detail design.

4.4.3.5 Coding

Software implementation or coding is the final stage in the canonical model of the software life cycle. It is where programs are written from the detail design, and tested. Tools are being created to automate the production and testing of software, but these are largely still done manually.

Tools, for generating code from a formal detail design and for verification, include those based on the 'constructive' approach. Here further refinement of the detail design is aimed towards the needs of the programming language, and verification is mostly proof of correctness between the code and the low level specification. Such tools are expensive, and mostly have been developed for sequential and critical

applications. Code generation for informal techniques are often manual, but fourth generation languages and program generators are available that automate much of the work, though they have limited ranges of application.

4.5 Comprehensibility and Creativity

4.5.1 Idea Generation

Individuals vary in the preferred method of creativity. Some like to imagine and develop ideas in their minds. Others take a visual approach and develop ideas on paper. Some use checklists or other prompts to spark off ideas. Alternatively, creativity can be improved by a group of idea generators, where ideas can be 'kick started' as in brain storming, or 'kicked around' and built on.

4.5.2 The Creative Process

It would be unusual for an idea to be realised in its final form without any intermediate steps, tools or techniques. The creative process generally includes:

- a medium for development
- a medium for storage
- a medium for communication
- a means of representing ideas
- a means of relating ideas

These can be called the tools and techniques (or methods) of the creative process. The creative flow in any design should be restricted as little as possible. A bad choice of tool or method can hinder progress, resulting in a worse solution.

4.5.3 Media for Development and Recording Creativity

Apart from the mind, the pen and paper approach is the least restrictive medium for an individual to quickly develop and store an idea, and is also convenient for communication. A4 sized paper is familiar, and a convenient size to start with. Bigger sizes will be needed for the overall model, or for sub-models of large or complicated systems. The path to a decision or an idea is important, so a record should be kept of the progress made. Examination of the recording may reveal an error in its creation. Errors are inevitable where creative development techniques require the absence of criticism of ideas.

A tool to aid the creative process within a group would be a video recording of the proceedings. The camera would capture the visual development evolving on the paper, at the same time as recording the verbal dialogue. Video recordings are time dependent, and can more easily show progress or evolution of an idea, than can a paper only technique. A computer system could record progress and replay it in real time. This could be accompanied by an audio recording, or combined in a multi-media computer. A computerised 'pad and pen' could act as the medium for development, storage and communication, and could execute the methodology for software development.

4.5.4 Modelling Methods

Methods imply a convention and a right and wrong way of representing a model. Models held and developed in the mind conform to this least. Using natural language, e.g. English, to describe the ideas and their inter-relationships is still very flexible, but some ideas are not easily described using current vocabulary. 'Mind maps' use natural language to describe ideas and graphics to associate between them. Mathematics provides strictly defined rules to specify relationships between ideas, see section 3.2.1. Petri nets allow simple mathematical relationships to be defined in a graphical way.

4.5.5 Idea Communication

The idea is not only for the generator, but also for others interested in the system, so communication of the idea must be possible independent of the generator, so the methodology (facilitated by the medium) must be comprehensible and enable the production of understandable ideas.

Validation, refer to section 3.2.2.1, can only be accomplished by communicating the idea to the customer. The medium for communication must be convenient and allow modification and replication.

4.5.6 The First Creative Steps

When an individual wants to develop an idea, unstructured impressions, or feelings, about the ideas are made. At some point, and for some reason (development, storage, communication), a modelling method and an appropriate medium to set down the ideas are adopted. It is the author's belief that a natural language/ graphical method should be chosen and developed with a pen and paper, because these are the least restrictive and most flexible, familiar and convenient tools.

4.5.7 A Repeated Procedure

In contrast to creating an idea is repeating the same approach in a different way. The user might have choices of tools and techniques. Today, draughtsmen often prefer to adopt CAD packages when making design modifications. The author believes a similar selection will be taken if an equivalent suitable tool was available in this and in other domains.

4.5.8 Comprehensibility of Formal Methods

In Prof. [Norcliffe's 1995] inaugural lecture, many and various issues were discussed. "Computers solve problems by carrying out instructions", and software developers supply the instructions. They can adopt a formal method to help "build the right system" and "build the system right". Computer algebra packages can assist mathematicians by performing the tedious manipulations and simplifications. Super-computers can produce too much numerical output for humans to read, so visual output is important. "We can instantly comprehend pictures and spot structures and anomalous behaviour" and "visualising mathematics, in a dynamic context is a potent aid to learning and understanding".

Comprehensibility is of paramount important in any development process. However, to non-mathematicians, some formal methods are incomprehensible, so formal methods are not adopted in the

first place, refer to section 4.4.2, or specifications produced by them are not validated properly by the customer, see section 3.2.2.1.

Graphical formal methods are available, such as Petri nets, marked graphs and finite state machines, refer to section 3.3. Like formal methods in general, these do not successfully represent non-functional system characteristics. Where formal methods can not represent particular features, recommendations, e.g. Def Stan 00-55, are to complement formal methods with natural language descriptions, refer to sections 3.2.4.1 and 4.6.1.

4.5.9 Conclusions

Part of the development process is creative, and at some stage the ideas must be validated by the customer. A methodology must facilitate the generation and communication of ideas, more so when the system is novel, less so for modifications.

Pen and paper, or a computer assisted pen and paper, appear to be appropriate media for development, storage and communication of ideas. Graphical/natural language combinations seem to be the most natural means of representing and relating ideas.

4.6 Petri Net Graphs and Pseudo Code

4.6.1 Petri Net Graphs and Pseudo Code in Modelling and Design

4.6.1.1 Model Description or Representation

The main and supporting modelling tools, Petri nets and pseudo code, are described here by means of a comparison.

There are many major differences between Petri nets and pseudo code. Pseudo code is a convention, and Petri nets are a formalism. Conventions are not considered absolute or universal. They are freely modified to suit the needs of the user. Formalisms are made up of mathematical rules relating sets, and violation of the rules would render their usage mathematically incorrect, unless the change in rule was proved to be correct.

Pseudo code is a textual description, while Petri net graphs are a mathematical and pictorial representation of interacting concurrent processes.

Pseudo code can describe anything that can be described in natural language - English. Pseudo code has three defined structures symbolised in English. They are: loops (WHILE, REPEAT), conditions (IF THEN ELSE) and blocks (modules and procedures). Anything else has only to be defined in pseudo code before use. Pseudo code is not simply natural language, but is much more formal. English uses metaphor and allegory to help describe, which are additional conventions in themselves.

Petri nets were developed to describe the interaction between concurrent processes. The basic Petri net is a place- transition Petri net [Peterson 1981], and is made up of five primitives: places, transitions, input arcs, output arcs and a marking, refer to section 3.3.

Transitions can symbolise interactions, conditions and events, and places can symbolise processes, statuses and activities. Arcs describe the association and direction of interaction between concurrent processes. Tokens indicate the marking of a place and can set statuses true or false, render processes and activities active, and represent buffers and queues.

4.6.1.2 Description Detail Versus Visualisation

A direct comparison can be made between Petri net graphs and pseudo code in terms of detail of description and visualisation. Pseudo code allows every and any detail of description anywhere in the code. In fact, pages can be written describing processes. Place- transition Petri nets allow places to be described, but far less is written in the circles. This is limited further in computer aided Petri net tools. Visualisation is at the other end of the scale to 'detail of description', though both are part of comprehensibility. It is the ease and effectiveness of absorbing, communicating and understanding the idea or model as a whole, which leads to the comprehension of the system it represents.

Petri net graphs allow the whole of a fairly complicated system (e.g. FMC) to be visualised on one large piece of paper, refer to the overall Petri net graph and the 'labyrinth' Petri net graph . The interaction between processes is clearly seen as directed arcs or arrows between places to transitions and transitions to places.

Pseudo code allows the description of interaction, which can be imagined, but is not as clear and 'fixed' as a picture. It is generally written on A4 sized paper, and a very simple system can be modelled on one sheet, but a fairly complicated system (e.g. the FMC) needed four pieces of paper to contain the description, refer to the appendices.

Redundancy is a distraction in visualisation, and by definition does not add to the description. In Petri nets, transitions must be followed by places and not more transitions. If a sequence of processes is represented in a Petri net, then the transitions do not contribute to the description, but arcs indicating relation remain important. There is no such redundancy in pseudo code. However, this distraction can be overcome in Petri nets by combining a sequence in a single transition or a single place. Petri net graphs have three distinct visualisation advantages over pseudo code for modelling concurrent systems:

- compactness
- graphical representations are absorbed more easily by the reader, than is text.
- interactions between processes are more obvious.

However, pseudo code can be used to describe anything describable in English, whereas Petri nets have a specific domain of application, i.e. concurrent systems.

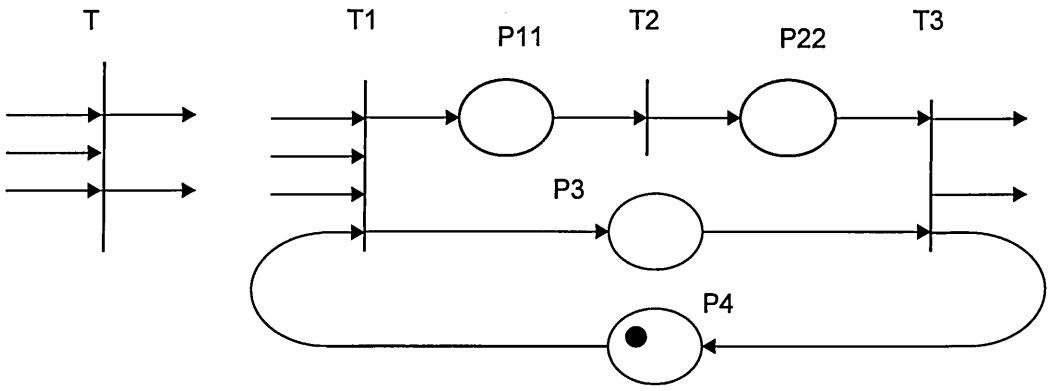


Figure 4-2 A transition decomposed into a subnet - preserving synchronisation

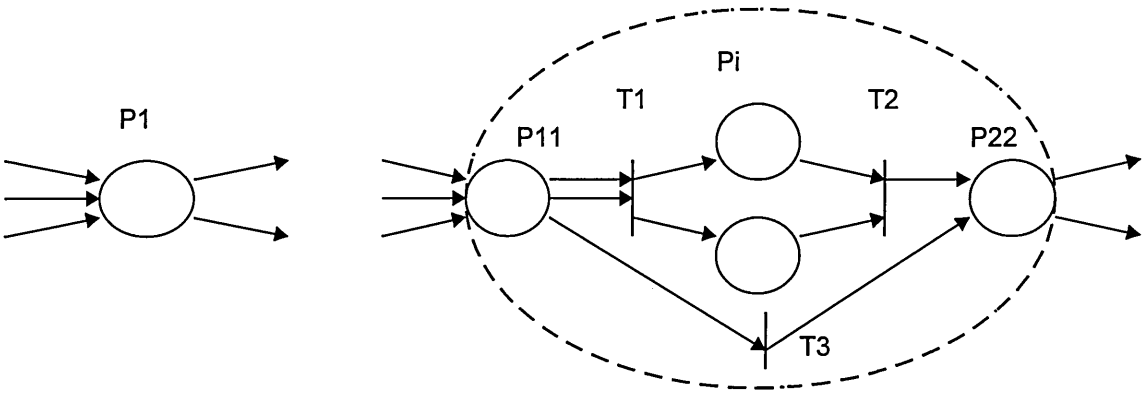


Figure 4-3 A place decomposed into a subnet - preserving marking

4.6.1.3 System Development

The PC based FMC control was originally designed using pseudo code, refer to the appendices. It was later modelled in Petri nets for subsequent transputerisation, but unfortunately for the reader, not in a structured way, refer to the 'labyrinth' Petri net graph.

Although Petri nets are a design and analysis tool, the literature presents most applications of Petri nets for analysis, and very rarely for design. This may be because most systems have been designed using other design tools, and have later been modelled for subsequent Petri net analysis.

Conducting a comparison of pseudo code and Petri nets as development techniques is difficult for the following reasons:

- as just stated, examples of Petri net usage in design are rare
- pseudo code has been undervalued as a technique due to its simplicity, so examples of it are again scarce
- pseudo code is mainly adopted in sequential systems, while Petri nets are used in parallel systems

Pseudo code development mostly progresses in a 'top- down' 'stepwise decomposition' hierarchical manner. The top level in the hierarchy describes the major tasks in the system. Each task is broken down, i.e. decomposed, into smaller tasks in the next level in the hierarchy. This step is repeated until the tasks describe the system in sufficient detail capable of being coded.

Petri nets are usually drawn in their entirety. Techniques can be used to represent sub-nets as places or transitions, refer to Figure 4-2 and Figure 4-3. Parameterised Petri nets are high level Petri nets, refer to section 3.3.2, that are formalised to enable hierarchies of Petri nets to be constructed and analysed. Colour Petri nets, another high level Petri net, can reduce the space needed compared with a place-transition Petri net, but with diminished visualisation.

4.6.1.4 Granularity

Concurrent systems are parallel at some level. Systems intended for repetition of calculations can 'vectorise' [Hockney 1988] the algorithm within the loops. This is considered as small grained granularity, i.e. small dependent modules. An example of large grain granularity is where much larger, virtually independent, sub-systems exist concurrently.

Pseudo code can describe any granularity at any point within a system. The concurrency can fit in the hierarchy of top- down development. The description is, however, the description of the hierarchy of the modules and not of the system. The interaction between concurrent modules is not obvious.

Parameterised (hierarchical) Petri nets, refer to section 3.3.2.3 allow places or transitions to be broken down into further detail. This detail is hidden at high levels of abstraction. The description power of parameterised Petri nets is closer to that of pseudo code than place-transition Petri nets, but at the cost of visualisation. For example, dropping down two levels in the hierarchy will reduce comprehensibility by losing visualisation.

4.6.2 Analysis Capabilities

Apart from the visual contrast, the marked difference between Petri nets and pseudo code is in their formality. Mathematically based methods, such as Petri nets, enable analysis, but informal pseudo code provides no such assistance.

Analyses available in Petri nets (refer to section 3.5.1) can reduce the doubt expressed in Table 4-1 of section 4.1, but are not adopted in the methodology.

4.6.3 Conclusion

The formal structure of Petri net graphs has to be known and understood by 'users' before comprehending any Petri net graph. Once known, this structure does not have to be explained in a Petri net graph and thus saves space. However the structure does sometimes hinder the design, and subsequent reading, of the Petri net graph, thus narrowing its applicability. Pseudo code is not as effective in modelling concurrent systems as Petri nets.

Pseudo code can be extended for improved modelling of concurrent systems. This can be done by making modules describe only sequential processes or describe process interaction. This adopts the fundamentals of CSP (communicating sequential processes), but in a less mathematical form, but would end up looking like occam.

Together Petri nets and pseudo code can be a formidable means of representing and describing concurrent systems.

4.7 Development with Petri Nets with Occam

This section serves three purposes: to discuss the state of the art, to set the research methodology in context, and to identify the gap in current work and hence the novelty in the research methodology. There is a variety of research involving Petri nets and occam in the current literature. The examples are categorised into: Performance, CASE Tools, Designing Petri Nets for Occam, Tools for Designing Petri Nets and Occam, Designing Safety Systems with Petri Nets and Occam and Non Petri Net Graphical Methodologies for Occam Code Production. Those examples with greater relevance or with similarities are discussed at more depth.

4.7.1 Modelling Occam in Petri Nets

4.7.1.1 Carpenter

[Carpenter 1987] All occam processes (SKIP, STOP, assignment, input, output, loop, conditional, sequence, parallel, alternation) are modelled in unmarked Petri net graphs. An example is given showing how deadlock is caused by the conflict of an alternation. Its resolution ensures that each process guarded by an alternative contains the channel inputs of the other alternatives.

4.7.1.2 Xu

[Xu 1991]A similar approach is taken to Carpenter's, but a subset of occam is modelled. The subset consists of SKIP, assignment, input, output, loop, conditional, sequence and parallel, and are modelled in Petri nets to analyse the code's determinacy, termination and liveness.

4.7.1.3 Steinmetz

[Steinmetz 1987]A comparison of two concurrent languages, Chill and occam, is made as to their potential for Petri net modelling. Chill requires the extra modelling power of high level Petri nets to model its code, while occam code can be modelled using restricted ordinary place-transition Petri nets, and moreover the reverse, produce occam code from the model. The method prevents the non-deterministic modelling of Boolean expressions in guards of ALT and IF constructions.

4.7.1.4 Best

[Best 1991]DEMON is the acronym for design methods based on nets of ESPRIT 3148. Its objective is to produce a formal design and verification framework for sophisticated concurrent systems. Its bases are Petri nets and partial order models to represent causality and concurrency in algebraic structures. Primarily, it uses net theory in composition, refinement and abstraction; equivalence and implementation; formal proof and analysis. One facet of the work involves translating an occam subset (data, blocks, procedures and priorities) into base level algebra, which is translated into high level Petri nets and onto low level Petri nets.

4.7.2 Performance

4.7.2.1 Balbo

[Balbo 1992]The goal is for correctness and performance in parallel programming. This is achieved through the construction and analysis of generalised stochastic Petri nets (GSPNs) from the CSP like code under examination. Like Carpenter, Balbo provides GSPN equivalencies for occam processes, but non-determinism and timing can be catered for with this statistical approach. Once the GSPN model is constructed from the code, and static analysis and performance evaluation is done, then communication graphs are used to determine how best to map applications to parallel architectures. The resulting structure is modelled in GSPNs, and is subjected to the same analyses.

4.7.3 CASE Tools

4.7.3.1 MARS

[MARS] is a modelling, analysis and prototyping environment for parallel systems. It allows object-oriented, VHDL and high level Petri net specifications of protocol, real time and distributed applications using structural, time, behavioural and quantitative analyses to produce Ada, c++ and occam prototypes.

The environment allows the use to enter a colour Petri net model in the graphics editor, and, after performing the various analyses and making the appropriate corrections, generates the prototype code for the chosen architecture.

4.7.3.2 Breant

[Breant 1991]A rapid prototyping tool for producing efficient parallel code is discussed, that can generate occam code from a Petri net specification in three steps:

Step 1 - analysis/translation
Once the Petri net model is entered, then it is analysed to find a set of interacting processes (sequential, synchronous interaction and asynchronous interaction (buffers)). Two nets are built from this: a sub-class Petri net graph consisting of sequential places, buffer places and synchronous transitions; and an interaction net, which seems to be a less detailed version of the first net.

Step 2 - object location
The non-functional requirements are integrated within the Petri net, such as hardware characteristics, process mapping, load balancing and communication management. A 'mapping net' is made up of a set of processes and a set of process connections, which are generated from the sub-class Petri net graph according to a set of rules.

Step 3 - occam code generation
The mapping net is translated into occam code. The three types of occam processes produced reflect the three interacting processes of step 1. The code produced terminates, avoids deadlock and allows for producer-consumer, buffered and point-to-point synchronisation.

4.7.4 Designing Petri Nets for Occam

4.7.4.1 Kerridge

[Kerridge 1984]Three parallel languages, occam, Pascal-Plus and Edison, are compared as to their applicability to the control of a robot arm. A program is written for each language from the same corrected Petri net graph specification. However, an intermediate step between specification and code is produced, because each language has a different model of communication, for example "Pascal-Plus is monitor based", "Edison is based on conditional critical regions" and "occam relies on the concepts of CSP". The intermediate step in the occam solution is a communication diagram, which shows the channels, their names and the message direction.

The indicates how much more succinct and realistic the occam code is compared with the other solutions.

Discussion
It is worth noting that, the Petri net graph specification is relatively simple (9 places and 8 transitions), and the text states that there are "essentially four processes which may be activated concurrently", but they are not obvious from the Petri net graph.

4.7.4.2 Gorton

[Gorton 1993]Petri nets are used as general purpose design, analysis and modelling tools for parallel programs. Coloured Petri nets are produced and then transformed into occam code. This is done by modelling each colour set in its own procedure. Each procedure is made up of parallel processes for each activity, and every colour Petri net transition is represented by an occam channel. The overall or top level code runs all procedures in parallel. Most procedures are replicated, because colour sets act like variables.

The paper does not present a methodology, and no Petri net /occam equivalencies are given, but provides a solution to an example, which includes a non-deterministic multiplexer.

The coupling of colour Petri nets and occam seems an attractive approach where replication is needed.

4.7.5 Tools for Designing Petri Nets and Occam

4.7.5.1 Lau

[Lau 1993]A software design tool for parallel programming is presented, which helps "define, refine and verify occam programs".

The paper begins by examining graphical tools to aid the design of occam code, and cites the use of tree diagrams, state transition diagrams and data flow diagrams, but criticises their lack of deadlock and liveness checking.

Occam is praised for its "inherent concurrency and choice", formal CSP basis and the ability to model "complex problems as a collection of concurrent processes that communicate with each other through explicit channels". Petri nets are commended as "formal mathematical and graphical methods to check for liveness and safeness" and their ability to "describe systems that are concurrent, distributed, deterministic and or stochastic with either synchronous or asynchronous communication". However, "Petri nets and occam have similar attributes that make designing of occam by Petri nets simple and natural".

Petri net models are produced for occam constructs, and are used as building blocks for constructing Petri net models for subsequent analysis and translation into occam code, similarly to [Balbo 1992].

Analysis, via matrices, determines whether the model produced will terminate and or deadlock, and whether its is bounded or safe. This is described in detail with examples. Translation into occam, however, is given only cursory mention. The Petri net graph and occam code for a simple producer-consumer example are given in full.

Two ways of using the tools are top-down, which highlights communication, and bottom-up, which provides extra detail. [Lau 1993] concludes that

- Petri nets have always been used as a concurrent design and analysis tool, and are better than data flow diagrams

- occam and deterministic ordinary Petri nets have many attributes in common, and together produce occam code for use on transputers
- a Petri net model "can be large and creates problems of its own, there are now CASE tools for Petri nets that can assist at least in the area of Petri net modelling"

Discussion

The aim is to produce occam code, which will not deadlock, via a Petri net design. However, rules to translate the Petri net model to occam code, and rules guiding away from deadlock are lacking. It is true that large Petri net models present problems, i.e. vast mathematical computation and unreadable Petri net graphs. CASE tools can facilitate computations, but cannot make Petri net graphs more readable.

4.7.6 Designing Safety Systems with Petri Nets and Occam

4.7.6.1 Birkinshaw

[Birkinshaw 1994]The thrust of the work is safety in parallel systems engineered via formal methods, analysis and CASE tools. Issues identified in safe parallel systems are: deadlock, livelock, complexity, performance and fault tolerance. Complexity of parallel systems "is the main barrier to understanding", and the "necessity to reduce complexity is of paramount importance".

Design considerations include:

- "knowing a design to be correct requires an understanding of its behaviour"
- "design faults which lead to an incorrect behaviour can be identified if the design is described in a formal way that lends itself easily to mathematical analysis"

Petri nets are used as the modelling tool, because they "can be viewed as a model of the causal relationship between states of communicating sequential processes", and "allow us to model the whole system: not just the hardware and software which is under computer control, but also the indeterministic environment in which the system operates". Coloured Petri nets are preferred to place transition Petri nets, because of their compactness.

'Safe state analysis' identifies unsafe or hazardous states. Petri net places and transitions can represent conditions and events of safe state analysis, respectively, and reachability tree analysis can determine whether a hazardous state can be reached from a given Petri net model. Petri net reachability analysis is also used to identify deadlock, but livelock requires additional temporal analysis provided by timed Petri nets, temporal logic or PAISLey, and reference is made to Gorton (section 4.7.4.2) and work originating from Carpenter (section 4.7.1.1).

Recommended treatment of time dependent systems is by specification and execution in PAISLey, and simulation in occam run on a transputer. A refinement process, between PAISLey and occam, results in a full specification ready for validation. PAISLey is a textual formal language with its own environment, that can specify both software and hardware timing, perform consistency checks and enable test data to

be applied to the specification, but "the format of the language is cumbersome, and does not adapt well to graphical representation".

Two CASE tools are examined: 'Software through Pictures' and 'Design/CPN'. The advantages of CASE tools are that they: allow "automation of laborious analysis, and reduce the likelihood of introducing human error".

Software through Pictures controls the integration of tools such as PAISLey, Petri nets, the Inmos toolset and formal proof checkers, thus "enable the user to design, extract, test, debug and document parallel code in one integrated environment".

Design/CPN is based on the design and analysis capabilities of coloured Petri nets with hierarchical extensions, and is controlled by an underlying functional language. Thus large models can be composed bottom-up or decomposed top-down, while "hiding the mathematics behind a graphical front end". An occurrence graph analyser produces a state space of the model to deduce deadlock and assist in timing and livelock analysis.

The ideal safety related CASE tool will integrate a flexible formal modelling tool, such as coloured Petri nets, with other semi-automatic tools, but without compromising the integrity of the model.

The paper concludes:

- "current software engineering methods and tools do not adequately support the requirements of the parallel systems developer"
- safety related parallel systems must ensure liveness in addition to safeness
- Petri nets are sufficient to detect deadlock, but livelock requires temporal analysis which is possible in PAISLey
- "hiding formal methods behind automated tools is possibly the best way to increase one's confidence in designs"
- Design/CPN is preferred to Software through Pictures, because of its formal modelling tool, and the flexibility of Software through Pictures allows the ad hoc incorporation of tools of unknown integrity
- the use of CASE tools, and in particular Design/CPN "appears to be the most promising way forward for engineering dependable parallel systems"

Discussion

Some of the issues discussed in safe parallel systems engineering: deadlock, correctness, understanding, CSP, occam, Petri nets and reachability analysis are issues in the research methodology. A contrast between the two can be made in the way that they are implemented: Birkinshaw's solution is parallel rather than distributed; a CASE tool rather than a methodology; more fault eliminating than fault avoiding; and understanding is devolved more to the tool rather than to the user.

The major criticisms are in the unstructured layout of the Petri net graphs, and relatedly the potential pitfall for devolving comprehension to the tool. The main differences are dealing with faults, by avoidance or elimination.

4.7.7 Non Petri Net Graphical Methodologies for Occam Code Production

4.7.7.1 Manson

[Manson 1994]Parallel Communicating Sequential Code (PCSC) is a methodology and CASE tool for designing and coding parallel software. Its core philosophy is "consistency between design, program code, configuration and documentation", and is achieved by a central database. It aims to be independent of language and architecture, thus be portable.

The methodology is based on CSP and makes use of data flow, data structure and communicating state diagrams and annotation. It aims to have the following steps: requirements analysis, system specification, design, verification, code generation, process mapping and execution.

The design stage consists of the following:

- determine the processes that are required
- determine the information that can be passed down the channels
- create a data flow diagram (DFD) for this design
- create a communicating state diagram (CSD) for each leaf process
- create a data structure diagram (DSD) for the data structures required in the CSD
- create a DSD for each piece of non-communicating sequential code and any required data structure
- add appropriate process and link annotation

The DFDs depict how the sequential processes operate in parallel and communicate, showing control, data storage and information (data and/or control) communication. Rectangles and circles indicate processes external and internal to the system, and arcs represent flows of information. Each process can be decomposed, enabling a top-down design approach and a conceptual hierarchical tree structure ending in leaves. DSDs define the protocol of the information communication shown in the DFD. Communication is unbuffered, synchronised and point-to-point message passing. CSDs model the behaviour of leaf processes with respect to their communication.

The DFDs, DSDs and CSDs and the verified design are used to generate c, occam and other concurrent code automatically by the PVM code generator. The code is derived from the diagrams and their associated annotation. The occam protocols are taken from the DSDs, the top level and intermediary processes are derived from the DFDs and the lowest level code, containing the communication, are taken from the CFDs. The generated code, the DFDs and the target hardware specification are used to map the processes to processors.

Use of the methodology is facilitated by a CASE tool called 'Software through Pictures'. It has a database management system which helps in consistency checking, and allows the coupling of a graphical user interface, verification, code generation and other tools.

Discussion

The rules on the construction of the diagrams are not explicit. This is necessary for the following reasons:

- the rules should ensure that the diagrams are built according to the underlying formalism

- the methodology or CASE tool should provide as much guidance as possible

The communication is confined to the CSDs. It seems unusual to have communication at the leaf or the bottom level in the hierarchy.

4.7.7.2 Jelly

[Jelly 1993]PARallel Software Engineering (PARSE) is a methodology for the "design of reliable and reusable parallel systems". It involves a top-down hierarchical approach, process graphs, Petri nets and transformation into occam or other parallel languages.

Petri nets are used to "specify the dynamic behaviour of the software" and "allow detailed executable design specifications to be formulated, and offer the potential for design verification using reachability analysis". They are applied "on a small scale at the lower levels of the design", because "Petri nets lack the structuring and scalability required for large systems", which is achieved by process graphs.

Process graphs "depict the process partitioning and communications relationships between processes, together with the role of each process in the system". Various symbols are used to represent process, communication path and protocol types and communication path constructors:

Process

- function server (normally passive)
- data server (passive)
- control process (active)

Communication protocol

- asynchronous (buffered)
- synchronous (unbuffered)
- broadcast (one to many- may need multiplexing hardware)
- bi-directional synchronous

Communication path constructors (input handling)

- undefined
- concurrent
- non-deterministic (random and fair)
- deterministic (selected or prioritised)

The top level process graph shows the problem decomposed into its major components and their interaction with the outside world. These are represented as combinations of the process, communication path and protocol types and communication path constructors. A series of top-down hierarchical decomposition follows, and ends when a process graph contains no internal asynchronous communication paths nor concurrent path constructors. The process, communication path and protocol types and communication path constructors in a process graph determine some of those in the lower level process graphs.

Transforming the process graphs into occam is as follows:

- derive occam processes from process graph processes

- derive the channels from the communication paths
- derive these channels' protocols from the message protocols associated with each communication path

Bi-directional paths need two channels and broadcast communication needs special protocol and channel consideration. However, "there is not always a one-to-one mapping of process graph communication paths to occam channels".

The type of communication protocol can determine the structure of the code. For example, where a simple function server receives a synchronous input and sends a buffered output, then the code could consist of an iterative channel input and a simple channel output. Where communication path constructors handle more than one input, then the occam constructions will be coded as PAR, SEQ or the non-deterministic ALT.

Where progress graphs cannot describe interaction of concurrent processes effectively, then Petri nets are employed. It is possible to transform all of the process graphs into Petri nets graphs as follows: "synchronous communications can be represented by a single transition shared by two processes, asynchronous and broadcast require a buffer place to hold messages not yet accepted, and bi-directional synchronous is represented by two indivisible synchronous communications. Further, non-deterministic and concurrent communication path constructors can be modelled using Petri net concepts of conflict and concurrency respectively". Petri net analysis is then possible, and suitable for transformation into occam.

Discussion

There does not appear to be a formal basis to the way in which process graphs are employed, so overall, the methodology seems to sacrifice integrity for convenience and flexibility. Guidance in transforming the model into occam is lacking, and analysis (including animation) is only possible when the process graphs are converted into Petri nets. If the methodology enables 'programming in the large', then a more extensive language, like Ada rather than occam, would be more applicable.

4.7.7.3 Schafers

[Schafers 1993]TRAPPER is a graphical programming environment for MIMD computers, and consists of design, mapping, visualisation and optimisation modules. The tools are written in c++ and run on the host computer, except the monitor which is written in the Inmos ANSI c Toolset and is run on the target transputer network.

Of greatest relevance is the design tool. It enables the parallel structure to be graphically presented using process graphs and the sequential components to be described textually. The process graphs consist of square nodes, representing processes, and arcs connecting nodes, representing channels. Input, output and bi-directional ports indicate where nodes interface with channels. Process graphs can be decomposed hierarchically into more detailed or lower level process graphs in a top-down manner, or in a bottom-up fashion where sequential processes can be parallelised.

Apart from sequential processes, all process graphs have double frames to indicate that they contain further process graphs. Sequential processes are described textually as Inmos ANSI c code. The design tool automatically provides the code's name, channels and header files.

The configuration tool maps software to target hardware. Firstly, the target hardware is specified by the programmer via process graphs. Next, manual or automatic partitioning and mapping is done. Lastly, software and hardware events, e.g. communication or load balancing, are entered into the monitoring system.

The visualisation tool analyses the design and provides run time characteristics (including deadlock detection), off-line and on-line animation. The optimisation tool shows loading characteristics and scheduling information.

Discussion

TRAPPER does not claim to be a methodology, but only a graphical programming environment, so no criticism is made. However, some of the concepts are similar to this research methodology:

- process graph flow appears from left to right
- concentric squares represent decomposed sub-processes
- a generic code format is generated from the process graphs
- process graphs and their process code have the same name

4.7.8 Observations and Conclusions from the Examples

By visual inspection of many of the examples presented here, it is often easier to comprehend the occam code than the Petri net graph. At least the top or overall occam procedure indicates which processes are concurrent.

Generally, in the hierarchy of occam code, communication takes place at a level below where the concurrency is specified, and above where the bulk of the work is done. Communication and processes can be shown at various levels in the code's hierarchy, but are generally depicted on the same level in non-hierarchical Petri net graphs. Some exceptions, sections 4.7.7.1 and 4.7.7.2, involve hierarchical decomposition Petri nets. In section 4.7.7.1, CFDs confine communication to the 'leaves' at the lowest levels.

Apart from hierarchical systems and hierarchical Petri nets, most systems appear more closely represented in Petri nets than occam.

Visualisation is not exploited in the use of Petri net graphs. Where systems are simple or small, e.g. section 4.7.4.1, the user might choose to draw the model manually. The user often trivialises the importance of the Petri net graph, because a well set-out graph aids understanding and idea communication, and facilitates visual consistency checks. Where systems are complex, the user will probably want to resort to a CASE tool. In doing so, the user is devolving responsibility of understanding to the tool, because generally CASE tools produce Petri net graphs which are difficult to follow. However, such tools often provide animation, so users can follow the progress of tokens around the net.

There are occasions when a model must represent actual communication or synchronisation, i.e. at the 'micro level', but mostly the description of communication is irrelevant when modelling DCSs. Figure 3-13 provides Petri net graph models of equivalent occam processes, such as ALT and PAR. Although useful at the micro level, they would complicate the essence of a macro level model.

Petri nets model concurrent systems. Parallel processing is often adopted to improve performance or reliability by using more than one processor. Distributed systems, with remote processing requirements, and parallel processing are often modelled using Petri net graphs, but Petri nets do not provide inherent partitioning to explicitly define which processes and their communications execute on which processors and their communication links.

Gorton's use of coloured Petri nets and occam could provide a useful extension to the research methodology. In the FMC example, the pallets are modelled in the Petri net graph of the status handler, refer to Figure 6-13 to Figure 6-15. The pallets might better be modelled in coloured Petri nets, where each attribute of the pallet could be a colour set or each pallet could be a set of attributes. This is left for further work.

Methodologies often augment tools and techniques, such as extending Petri nets to facilitate representation of common attributes of the system, e.g. section 4.7.7.2. This methodology has tried to minimise this, because it makes the models less comprehensible and the modelling tool less transferable and compatible. The three additions are the double concentric circles of complex places, refer to section 5.7.3, the mapping of places from exit places of controllers to entry places of other controllers, refer to sections 5.8 and 5.9.4, and other non standard Petri net representations which indicate the need for pseudo-code. Where some methodologies add too much that it is not obvious if the model is a Petri net graph.

Most examples aim to make good use of the system specification, rather than arrive at a good system specification, thus the code can be based on an uncertain foundation.

Petri net analysis techniques can show that errors exists, but do not show how to repair the faults. In the absence of any guidance, correct models are produced by a repetitive cycle of analysis and modification.

The models of the example applications fall into two categories: comprehensible and analysable. Schafers, Manson and Jelly present comprehensible hierarchies of process graphs or data flow diagrams, so, with the exception of Jelly which allows the use of fragmented Petri nets at the lowest levels, are not in a form to analyse. The other applications use Petri nets as analysis rather than graphical tools, because they are hard to read, but are complete and analysable.

There is no work that is directly comparable to the methodology described in chapter 6. The work in sections 4.7.6.1 and 4.7.7.1 discuss several common issues, but their emphasis on CASE tools departs from the comprehensibility, Petri net/occam equivalence and pro-activity which are core goals of the methodology.

4.8 Occam Development Examples

A number of occam programming reference books ([Kerridge 1987], [Pountain 1987], [Barns 1988], [Galletly 1990] and [De Carlini 1991]) were examined in order to compare recommended tools and techniques for developing occam systems.

4.8.1 Data Flow Diagrams

[Kerridge 1987] suggests a graphical aid to occam program design. Data flow diagrams (DFDs) are borrowed from SSADM to show processes and communication between processes. Two examples are given to explain its usage: a process control and a data processing application.

The use of DFDs is strictly adhered to. 'Mini-specifications' are produced from the DFDs, and are written in a similar way to high level pseudo code. In the examples each processing node of the DFD has its own mini-specification, which consist of a series of operations.

The relationship between processing nodes, shown graphically in the DFDs, and mini-specifications of the processing nodes are used to create high level programming structures. These structures determine how the program should control the data, but do not indicate how the code is to be written.

[Kerridge 1987] continues using the process control and data processing examples to illustrate how occam code can be produced from the DFDs and mini-specifications. No rules are given, but many useful occam programming tips are provided.

4.8.2 Programming Style

[Pountain 1987] suggests nine points about programming style:

- Factorise programs by using procedures, and factorise major procedures with further procedures.
- Group declarations above procedure and function specifications and adopt a convention for ordering the specification, such as channels, then timers, abbreviations and variables.
- The occam compiler is able to take advantage of abbreviations written in the source code, and produces object code that is more efficient.
- Occam lends itself to relatively readable source code. Names can be any length and contain full stops and different case characters (e.g. load.lathe and LoadLathe).
- Position specifications as local as possible.
- Library building is easy in occam. Suitable factorisation of code allows procedures saved as separate files to be included in new programs. Channels are ideal in building filters, pipelines and multiplexers.
- Occam enables the development and simulation of software and hardware. Software 'stubs', such as `SKIP, STOP and WRITE.TO.SCREEN("module 1")` can be substituted later for the real module. Procedures can simulate hardware before configuration is considered, so postpone configuration and priorities until the end.
- There is little to help writers of parallel programs in terms of techniques.
- Occam is a specification language, and allows the programmer to be explicit.

Pountain also argues for use of the folding editor. A fold is of more value than a procedure, because each fold is entered (unfolded) via its description, and a fold can contain a number of procedures preceded by a common specification.

4.8.3 Occam Transformations

[Barns 1988] does not discuss the folding editor, but promotes the modular benefits of procedures. Libraries of separately compiled procedures can be inserted into a hierarchy of procedures, which make up an occam program. Procedures are categorised into those that are part of an occam PAR construct and contain channels, and those that are made up of sequential statements and do not contain channels. The use of stubs, such as `SKIP, STOP and WRITE.TO.SCREEN("module 1")`, are advocated for program development.

A useful chapter shows how occam equivalencies enable transformation of occam code:

- to change a clear but inefficient program into an efficient but perhaps obscure one
- to change a sequential program to exploit parallel hardware
- to change a concurrent program into a sequential one for more efficient execution on a single processor
- to change a physically unfeasible program into a physically feasible one

22 'laws' of transformation are categorised into 7 headings: associativity, symmetry, replacing SEQ by PAR, declaration, loop rearrangement, replicated SEQ and distributivity. An example of distributivity and replacing SEQ by PAR is:

```
PAR                          SEQ                          SEQ
  SEQ                          PAR
    chan ! x                     chan ! x                   y:=x
    proc1(x)        ⇔            chan ? y        ⇔
  SEQ                          PAR                        PAR
    chan ? y                     proc1(x)                   proc1(x)
    proc2(y)                     proc2(y)                   proc2(y)
```

Such transformations are possible because occam is mathematically based.

Barns compares occam and Ada: "Occam provides a much simpler collection of language features, when compared with Ada; it does not support shared variables, aborts, timed or conditional message sends, or an extended rendezvous. Inevitably this results in a language that is less controversial, has sounder formal base and is more efficiently implementable." " And yet the main distinction between the languages is not what communication features are supported but their differing views as to the nature and notion of a process".

4.8.4 The Design Phase of Software Development

The design phase starts after analysis and specification of requirements [De Carlini 1991], so the choice of design technique depends on the requirements such as:

- performance- sub task completion times
- reliability- operating with part system failure
- growth- expansion

In parallel software design, modules and data are partitioned into processes, which are allocated to processors, and is made up of three or four stages:

- functional decomposition- breaking down the system by function (discussed below)
- partitioning- map modules to processes
- allocation- assign processes to processors

and sometimes

- scaling- find the optimal number of processors (discussed below)

Decisions taken at any one of the first three processes will affect the others, and so the overall design. However, it is not very clear how to fully exploit hardware and software characteristics of the system.

In functional decomposition, the functions of the application are decomposed at two logically different levels:

- very high level- identify the different tasks required to process the input data
- low level- relating logical modules of code and the data needed to carry out individual tasks

Very high level functional decomposition can be accomplished using conventional design methods, but parallel systems with their small grain structure are better served by low level functional decomposition.

Different systems lend themselves to different approaches to functional decomposition. There are systems that have an obvious geometry or symmetry in physical or behavioural structure, systems that are orientated around data, and systems that do not follow either. Where geometrical structure is important, then it is usual to examine partitioning before decomposition takes place. Due to the restriction imposed on the decomposition phase by the geometrical structure, then domain or data structure decomposition techniques are used.

Scaling is done to find the optimum size of the system once the processes are allocated to processors. This phase is not necessary if the grain size and the interconnection of the processes had been fixed at the partition phase. Applications can benefit from multiplying processes on to additional processors to 'speed up' or 'scale up' in performance.

- Speed up reduces the time taken to carry out the application.
- Scale up executes a larger problem in the same time.

The communication overhead inevitably increases with more processors, so performance does not have a linear relationship with the number of processors. Where the size of an application is fixed, any scaling will reduce the actual size of the process, and increases their number, and so increases the communication overhead. Plotting performance against cost/ performance will indicate the cut off point for the number of additional processors. If reliability is more important than performance, then choose fewer processors, but if performance is preferred to reliability, then use as many processors as needed. [Or adopting a performance related design tool, e.g. stochastic Petri nets.]

4.8.5 Models of Parallelism

[Galletly 1990] has a chapter discussing "approaches to writing parallel programs in occam", which mainly consists of models of parallelism.

Granularity is a measure of parallelism, and indicates the number of parallel processes. High granularity incurs a high communication overhead, but low granularity can lead to processor workload inefficiencies. Parallel program structures can be categorised into algorithmic, geometric and processor farming.

Algorithmic or data flow decomposition - algorithms are created or modified to include parallelism. For example, a solution that would naturally have work done in a loop or iteratively, and has values passed into the next iteration. The parallelised form would have the iterations running concurrently and passes the values between them, e.g. a pipeline. This is efficient when many of the same problem need processing, but because the processes are different, then there will be workload inefficiencies.

Geometric or data structure decomposition - the regular spatial geometry or structure inherent in the problem can be identified, and the process replicated onto processors. This approach is highly granular and processes communicate frequently with their neighbours.

Processor farming - independent replicated processes work on different data, which is provided and collected by a farmer, so granularity is very low.

[Galletly 1990] proposes the use of data flow diagrams, Petri net graphs and CSP for expressing the design of occam programs, but does not specify how.

4.8.6 Discussion

These tutorials and introductions to occam programming do not provide suggestions indicating how to achieve correct code. Most mention deadlock, but techniques to solve the problem are left to specialist work, such as those discussed in section 4.10.

4.9 Mutual Exclusion

In mutual exclusion, processes share a resource, but while one process is using the resource then no other process can access the resource. Petri nets model mutual exclusion well, and occam provides the non-deterministic ALT construct to enable safe resource sharing. In the example of Figure 4-4, processes A and B can claim the printer, but if A is ready before B is ready, then the printer can print A. The mutually exclusive regions are print A and print B.

4.10 Deadlock

4.10.1 Conditions for Deadlock

There are four conditions necessary for deadlock to happen in concurrent processes [Banaszak 1990]:

- mutual exclusion- processes require the exclusive use of resources (refer to section 4.9)

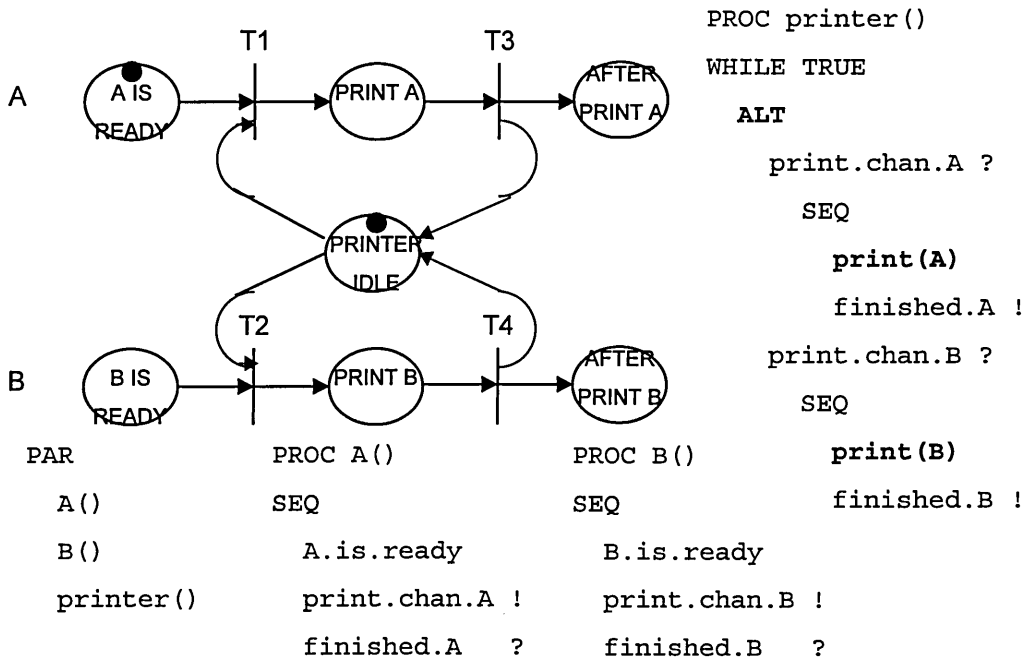


Figure 4-4 Petri net and occam models of mutual exclusion

- hold while wait- processes hold onto resources while waiting for additional required resources to become available
- no pre-emption- processes holding resources determine when they are released
- **circular wait-** closed chain of processes in which each process is waiting for a resource held by the next process in the chain

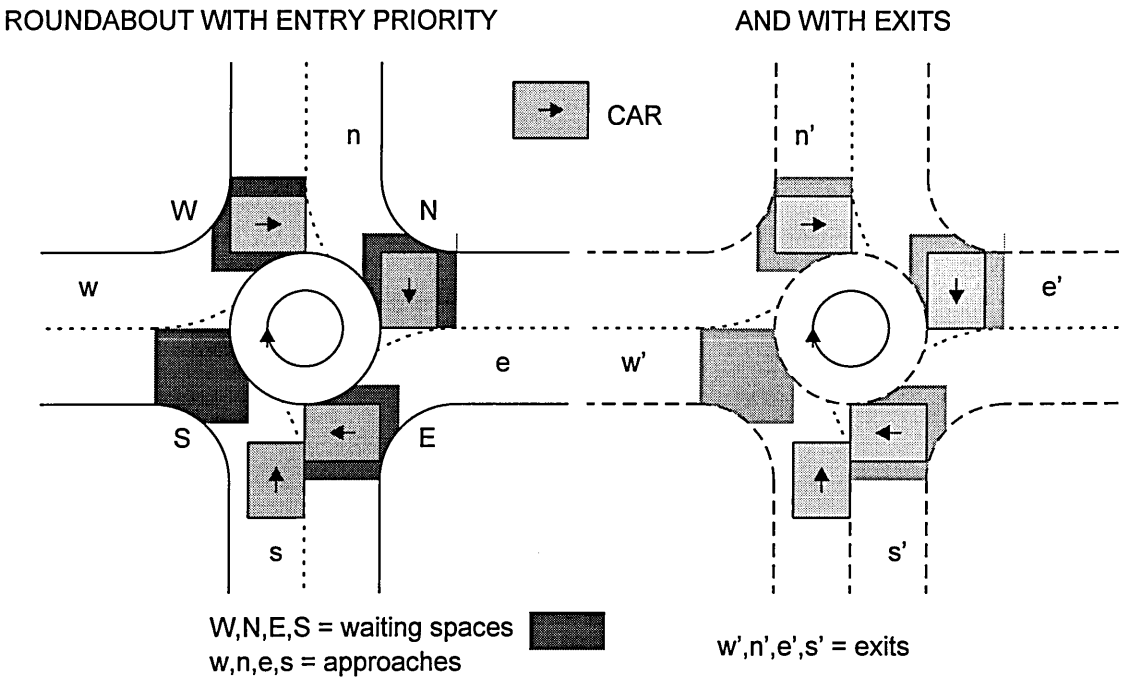
Deadlock can be prevented if any one of these conditions is avoided.

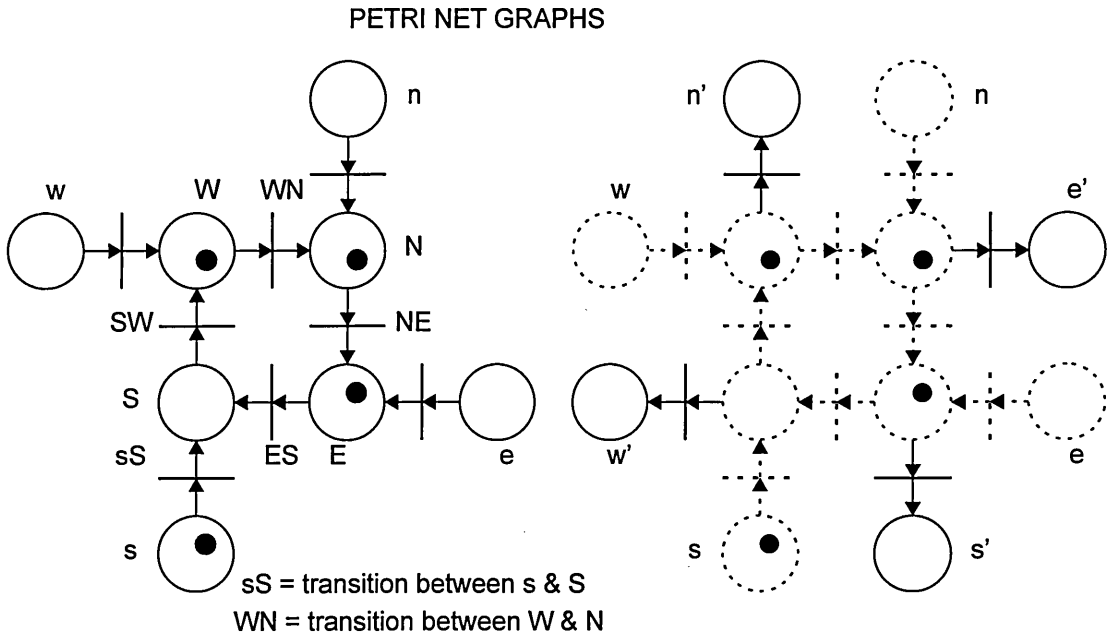
For example, the potential deadlock inherent in a traffic roundabout which gives priority to cars entering, refer to Figure 4-5. Deadlock occurs when all cars want to turn right, and the roundabout is full. If one car wants to leave the roundabout at the next exit and another car wishes to enter at the previous entrance, then the exiting car must wait for the entering car. Here, cars are processes, and spaces are resources.

There is a space in roundabouts that must be crossed by cars entering and exiting, and is shown by the four dark shaded areas. For roundabouts with entry priority, the space can be occupied by a waiting car that had recently arrived. For roundabouts with exit priority the space is not occupied (the resource is available to no car), and is only crossed.

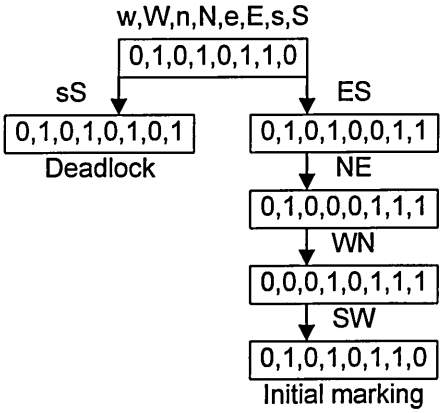
Relating the example to the four conditions for deadlock, while any car occupies its current space:

- it has exclusive use of that space (mutual exclusion)
- it waits for the space in front to vacate before it releases its current space (hold while wait)
- its space cannot be taken until it no longer has use for it (no pre-emption)
- it cannot proceed until it has access to the space in front, which is true for the car in front, which in turn depends on the first car (circular wait)





REACHABILITY TREE



All cars want to turn right, and a car approaches from the south.

IF the car at s arrives before the car at space E proceeds (transition sS fires)
THEN deadlock
ELSE the car at space E moves to the space at S (ES fires). Next the car at N moves to E (NE fires), W moves to N (WN fires) and the car at space S moves to space W (SW fires) which returns to the initial marking.
ENDIF
Modelling cars leaving leads to a much larger reachability tree.

Figure 4-5 An example of deadlock showing Petri net model and reachability tree

Deadlock can be prevented if any one of these conditions is avoided (e.g. in roundabouts):

- (mutual exclusion- the ability for each process to exclude other processes from use of its current resources, means that this cannot apply to all resources simultaneously)
- hold while wait- processes must request resources simultaneously and none can proceed until resource determination is complete (e.g. cars must arrive in a pre-determined way - precluding random arrivals)
- no pre-emption- where processes have been denied a request while holding resources, then the resources must be released before the request can be granted (e.g. a car must vacate its current space before it requests the space in front)
- circular wait- processes can only request resources which are linear, i.e. r1, r2 then r3 (e.g. roundabouts must not be round, i.e. spaces should be linear as in traffic lights)

4.10.2 Deadlock Detection in Occam

This section abridges a paper [Joosen 1989], which describes a static analysis tool for deadlock detection in occam.

A definition of deadlock is given: "several processes may compete for a finite number of resources. A process requests resources, and if the resources are not available, the process enters a waiting state. It may happen that the processes never leave this waiting state because resources they have requested are held by other waiting processes."

Two classes of resources are defined:

- reusable- a fixed number of pooled resources, e.g. database records or printers
- consumable- produced and consumed by processes, e.g. messages

In occam, messages are the only consumable resource, so in occam "deadlock is the set of processes that either compete for reusable resources or communicate with each other". Also in occam, "reusable resources are embedded in active processes", therefore "the only kind of deadlock that can occur is a communication deadlock", which simplifies the problem. Deadlock can be detected at the implementation, the testing or the normal execution stages in the system life cycle, and static analysis, run time detection and dynamic run time detection techniques are adopted respectively.

The approach taken in the paper, static analysis, has the advantage of being independent of the run time system, and is effective earlier in the development life cycle.

The analysis tool starts with the previously written code. Process graphs are drawn from the code, where a process starts as 'passive', becomes 'active' at the beginning of the program, at a PARbegin or after a PARend, and terminates at the end of sequence of actions. An 'execution sequence' ends successfully with no processes active.

Complete analysis of a program would normally require all of its execution sequences to be investigated. However, the tool precludes this need by identifying and examining 'equivalent sequences'.

'Action graphs' are decomposed from the process graphs, by breaking them down at PARbegin, PARend, channel input, channel output and ALT, so sequences, assignments, conditions and timing delays are ignored.

The tool runs an algorithm, which searches for 'infinite wait' situations (including deadlock). For each active process: it simulates communication; and as a result two processes evolve into another action or terminate, and consequently might cause other processes to become active due to a PARbegin or PARend respectively. If all branches and all processes are examined and two processes can not evolve from the communication simulation, then an infinite wait is encountered.

Limitations of static analyses are mainly in expression evaluation, such as conditionally guarded ALTs, IFs, CASEs, discriminated channel protocols and arrays of channels. Other difficulties given, irrelevant to occam, are in pointers, dynamic task creation and recursion.

The paper ends by discussing work related to communication deadlocks. Very little work has involved asynchronous communication deadlocks, and of synchronous deadlocks only one other describes an algorithm suitable for systems without a kernel or operating system, such as transputers. The difference between the two are that [Joosen 1989] takes a 'space-time', rather than an 'interleaving', view of parallel programs, where terms are taken from [Alford 1982].

4.10.3 Deadlock Elimination in Occam

This section summaries a paper [Crowe 1989], which describes a CASE tool for designing deadlock free occam programs. The tool assists at the design, refinement and verification stages of software development. It is based on the process algebra CSP, is written in prolog and c and incorporates the use of the Inmos TDS at the implementation stage.

Process graphs are used at the design stage to depict the "number of sequential processes running concurrently and communicating by point-to-point channels". They consist of nodes and arcs, which represent processes and channels. A graphical hierarchy of process graphs is produced by decomposition at process nodes, and associated with each process graph is CSP notation consisting of "raw CSP, channel protocols and library calls".

"Network refinement" involves building a hierarchical tree. The tree represents the decomposition, and ends with "atomic processes at the leaves of the tree". The tool enables the "verifying of proof obligations" for every refinement step, and by following the branches of the tree, checks whether deadlock or other design errors exist. The cycle of 'modify the design and re-analyse', continues until correct.

Translating the corrected CSP notation into occam code is straight forward, and takes advantage of the folding editor and debugger in the TDS.

Discussion

Occam is an implementation of CSP, so they are very similar, but relevant differences are:

- CSP allows recursion, but occam (and place-transition Petri nets) do not
- occam has timing and priority characteristics, but CSP (and place-transition Petri nets) do not

The tool does not provide guidance at the design stage, but allows the user to produce ad hoc process graphs. Users who rely on making random changes in response to the errors found by analysis may never fully comprehend the specifications.

Information entry to the tool is via a process graph interface. This removes the need for structuring the CSP and occam, but at present the users must understand the CSP and occam languages to complete the design and implementation stages.

The three major attractions to this tool are:

- the use of a formal notation (CSP) which has very strong links with the implementation language (occam)
- early testing by analysis at the refinement stage
- the re-use of corrected user-defined libraries

4.10.4 Deadlock Avoidance in Occam

This section is a synopsis of a paper [Welch 1993], which adopts a 'pro-active' approach to achieving deadlock and livelock free communication in the client-server model of communication.

It argues that verifying the absence of deadlock is better than checking the global state of the network. An example of state checking is given: "The number of states in a multi-process network is bounded by the product of the states in each component process. Even for quite modest systems (e.g. a 10 process network, where each process has 10 states), we can have the order of a billion states to check out! For almost all practical systems, exhaustive testing by generating each possible state is impossible."

The client-server model of communication is "based on the notion of 'synchronisation classes' for processes that are closed under certain forms of parallel composition". Deadlock and livelock are avoided in client-server communication by ensuring acyclic communication between each client-server pair, and finite message acceptance times between each client-server pair, by imposing the following:

- one process becomes the client by initiating the transaction, so the other becomes the server
- the initial communication is accepted within a finite time, when synchronisation takes place, or the transaction is aborted
- messages are transmitted in either direction through a channel between client and server, but the client may not enter into communication with any other process, so must wait
- when the answer is transmitted from server to client, then the transaction terminates, and the processes are neither client nor server, Figure 4-6a
- while the server is serving the client, and the server needs information from another process, then is allowed to become the client in a separate client-server transaction, Figure 4-6b. This will delay the answer to the first client-server transaction

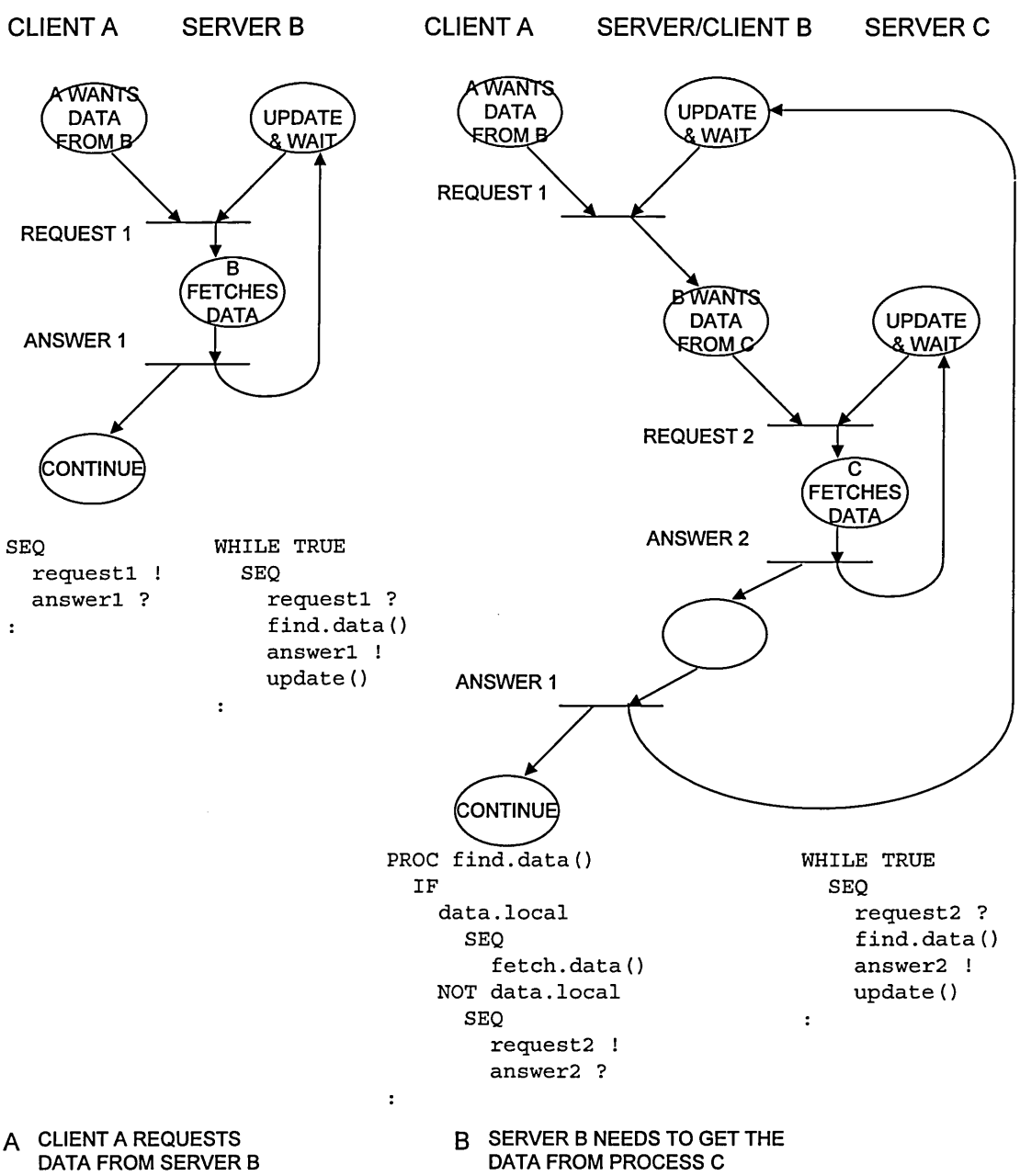


Figure 4-6 Client-server communication, in Petri net graphs and occam code

4.10.5 Discussion

The three methods for achieving deadlock free occam code are significantly different. The client-server approach, section 4.10.4, avoids the introduction of deadlock by imposing a restriction ensuring acyclic communication between each client-server pair. The other two are examples of elimination, and both allow ad hoc design, via decomposition of process graphs, followed by testing and redesign, using CASE tools. In section 4.10.2, a static analyser runs an algorithm to identify 'infinite wait' states. In section 4.10.3, CSP is produced for each node in the process graph, and the network is checked for deadlock. When the basic CSP is correct, then a process of formal network refinement ensures the detailed CSP, and therefore occam code, is correct.

Deadlock will prevent a system operating as required. It must be identified and measures taken to prevent or eliminate it. Prevention restricts the design process, but elimination is a tedious process needing computer assistance.

4.11 A Comparison with SIFT

Formal methods have been developed for more than one stage in the software life cycle, refer to Figure 3-1. Formal tools that integrate more than one formal method are being developed, such as SIFT, this research methodology and some of the examples in section 4.7.

[Moser 1990] describes the development of SIFT (Software Implemented Fault Tolerant operating system for aircraft flight control). Moser has outlined five levels of abstraction for the development of safety critical computer systems. Each level, except for the top level, builds on the previous level. The levels of abstraction are top level requirements (functional and safety requirements), design, program code, object code and execution. Techniques to validate one level against the one above are given in Table 4-2.

LEVEL	VERIFICATION TECHNIQUE
Top level requirements	Human inspection
Design	Design Verification
Program code	Program Verification
Object code	Verification of compiler
Execution	Hardware logic, micro-code verification

Table 4-2Verification techniques

The top level requirements are validated by human inspection, the design is validated against the top level requirements by design verification, the program code is validated against the design/model by program verification, the object code is validated via the verification of the compiler, and the execution is validated via hardware logic, micro-code verification and CAD tools. Moser stresses the need for using other techniques, such as testing and fault tolerance, in tandem to provide reliability necessary in safety critical systems.

The research methodology is concerned with all of the five levels of Table 4-2. The methodology uses Petri nets at the requirements and design stages and occam at the program code stage. Occam and the transputer were chosen for their mathematical basis and transputer reliability. Their object code and execution are examined in sections 3.4, 3.5.5 and 3.5.5.1

The sets of rules integrating and exploiting Petri nets and occam that make up the research methodology are given in chapter 5.

The major parallels between the methods of Moser and the research methodology are in the levels. Both models recognise the hierarchy of requirements, design, program code, object code and execution. Tools and techniques are also part of both methods to bridge the gap between the levels in the hierarchy. The bridges being human inspection, design verification, program verification, verification of compiler, and verification of processor(s).

The major differences are in the verification. Moser describes the construction of a design and a verification that the design meets the requirements, the construction of code and a verification that the code meets the design. The research methodology uses Petri nets in helping produce requirements, and in design, but in a restricted way in order to meet the needs of occam and prevent deadlock. So rather than Moser verifying that level 2 represents level 1, the research method makes sure that level 2 is built from the needs of level 1. The methodology forces the production of distributed occam code which works 'first time', while SIFT allows more choice in modelling and programming tools and techniques.

4.12 Conclusions

This is the final section before the methodology is described, and makes conclusions from this and previous chapters. It argues the need for a novel methodology in four stages: the needs of a transputer based FMC, the needs of an occam based methodology, the use of Petri nets with occam, and criticism of current use of Petri nets and occam. It ends by presenting techniques concluded as potentially useful to the methodology.

4.12.1 The Needs of a Transputer Based FMC

These are conclusions from chapters 2 and 3 that relate to the control, the communication of the control and the development of FMCs.

1 FMCs require flexible, safe, reliable and correct distributed control

The requirements of section 2.7 determine that the control of the FMC should be flexible, safe, reliable, correct and distributed.

2 Modularisation can separate the manufacturing from the distribution of control of an FMC, leaving a simpler DCS problem

The manufacturing control hierarchy of Figure 2-8a described in section 2.4 shows a structure where each element in a level controls one or more elements in the level below. The communication hierarchy,

shown in Figure 2-8b has a similar structure, which together enable elements to be modular. Therefore workstation modules take instructions from a single source above (a cell module), and issue control to machine and work handling modules below. A workstation module consists of the process, its transfer device and its local work handling equipment, as illustrated in Figure 2-4. Section 2.3.4 describes the modules for robot, lathe and miller workstations. By such modularisation, the manufacturing control problem can be simplified into a distributed control problem.

3 Transputers and occam are an effective DCS for use in an FMC

Section 3.5.5 determined the suitability of transputers and occam as the hardware and software in the control and distribution of that control in an FMC. They enable flexible and swift modification and expansion, and are relatively correct, reliable and safe.

4 There is a need for a methodology to develop flexible and dependable occam code

Section 4.8 reviews various recommended occam development examples, but none of these provides sufficient guidance in the development of flexible and dependable occam code. The review presented in section 4.2.1 concludes that the needs of a methodology (see conclusion 5) are not adequately satisfied.

These conclusions demonstrate that the development of a transputer based FMC requires flexible, safe, reliable, correct, distributed and modularised control. Occam is shown to be a suitable language for control, but better occam development techniques are needed.

4.12.2 The Needs of an Occam Based Methodology

The requirements of the methodology to develop occam software are taken from the needs of a transputer based FMC, above, and from the reviews of chapter 4.

5 Methodologies for dependable distributed control must consider safeness, liveness, reliability, portability and scalability

The major conclusion from the review of software engineering for parallel systems, refer to section 4.2.1 states that there is "a need for techniques and tools for the construction of reliable, portable and scaleable software systems". Conclusions from section 4.7.6.1 agree, and add that safety related parallel systems must ensure liveness in addition to safeness.

6 Visualisation is necessary in distributed control

The review in section 4.2.1 advocates the use of graphical techniques to manage the complexities required in design methodologies, and recommends Petri nets. Sections 4.5 and 4.6 discuss the benefits of visualisation in general and of Petri nets in particular.

These conclusions demonstrate that the needs of occam software development are not sufficiently served, but visual tools, such as Petri nets, could play a significant role in a methodology. The needs of a

methodology for safety related software engineering for parallel systems are: safeness, liveness, reliability, portability and scalability.

4.12.3 The Use of Petri Nets with Occam

These conclusions illustrate some desirable properties of occam and Petri nets to meet the needs of an occam based methodology.

7 Petri nets have many properties which are desirable in methodologies

Petri nets, see section 3.3, were created to model and analyse interacting concurrent systems, and their mathematical bases allow the system to be formally specified and analysed, see conclusion 10. The process of creating a Petri net model, within imposed constraints, is Petri net design. Petri nets can be represented in matrix and graphical form. Petri net graphs enable the depiction of concurrency, synchronisation and decisions, and thus aid visualisation. The Petri net graph models the structure of the system, while the marked Petri net graph enables the behaviour of the system to be 'animated'.

8 Petri nets are applicable to distributed systems

All parallel systems are logically the same, but differences between distributed and concurrent systems are the communication bandwidth and the communication latency, refer to section 4.3. Static techniques, such as deadlock analysis, are equally applicable to both, but dynamic analysis of distributed processing requires temporal techniques.

9 Petri nets have a good reputation in manufacturing, DCSs and FMCs

The review, in section 4.2.1, extols the virtues of graphical techniques, and in particular those of Petri nets. There are copious examples of the application of Petri nets in modelling most aspects of manufacturing, including communication, distributed shop-floor control and FMSs in particular, e.g. [Silva 1989]. The type of Petri net employed ranges from subclasses to high level Petri nets. Many Petri net modifications have been developed to facilitate modelling manufacturing features.

10 Petri nets and occam will improve confidence in a design due to their mathematical bases

Petri nets, section 3.3, are a set of specification languages based on 'bag theory', which allows multiple items in a set, and Petri net graphs are 'bipartite directed multi-graphs'. Occam, section 3.4, is an implementation of the process algebra communicating sequential processes (CSP), and the transputer is a hardware implementation of occam.

11 There are applications which make use of Petri nets with occam

Section 4.7 provides examples in the literature where Petri net and occam are used together. Their combinations are for modelling, analysis, performance, design and CASE tools. The example in section 4.7.6.1 relates to safety, but it does not fully exploit the visualisation potential of Petri nets, which leads to devolving the responsibility for understanding the design to the CASE tool.

These conclusions show the effectiveness of Petri nets in general and applied to FMCs, and shows that together Petri nets and occam meet the needs of a methodology for safety related parallel software. There are examples of a number of tools and techniques for developing occam code with Petri nets, but none provide adequate guidance and visualisation to enable the development of dependable occam code.

4.12.4 Criticism of the Current Use of Petri Nets and Occam

These conclusions relate to the inherent problems which can lead to the detrimental use of Petri net and occam.

12 Petri net graphs can be drawn that are incomprehensible

All but the simplest Petri net models, be they general or high level Petri nets, are difficult to read, and thus understand. The example that motivated the research, the 'labyrinth' Petri net graph, is so called because it is drawn in an unstructured way and thus is difficult to follow. All but the simplest Petri net graphs given in the literature are also hard to understand, refer to section 4.7.

13 High level Petri net graphs are less understandable than place transition ones

Although general Petri net graphs require more space than high level Petri net graphs to represent replications or variables, the information held in place transition Petri net graphs is often easier to absorb, because there are no arc functions to read.

14 Petri net refinement is tedious

The conventional approach of developing Petri net models seems to be to produce a model and analyse it. When the model is found to be incorrect, e.g. due to deadlock, it is then modified and re-analysed, see section 4.7.8. This step might be repeated several times before a satisfactory result is achieved.

15 Occam refinement is tedious

Similarly, the conventional way to produce occam code, refer to section 4.8, seems to be to provide a basic structure (using data flow diagrams or Petri nets), produce the occam code, then try to compile it. When it does not compile, it is then modified and recompiled. This step is repeated until successful. Next execution is attempted, but if found to deadlock, a further repetitive modification and re-execution step is endured.

16 Formal tools are largely unwanted in industry

Adopting formal or rigorous design approaches, as described in section 4.4.1, will help produce the necessary safety integrity highlighted in sections 3.2.4.2 and 3.2.4.3. The formal development life cycle, of section 4.4.3, describes a tedious and highly mathematical refinement process. The survey referred to in section 4.4.2 indicates that industry is reluctant to adopt such a refinement process. Section 4.7.6.1 suggests "hiding formal methods behind automated tools is possibly the best way to increase one's confidence in designs"

These conclusions demonstrate that formal tools are complicated, do not show the user how to get the correct solution, and therefore deter their use. There is a need for a novel methodology containing, and making better use of, Petri nets and occam.

4.12.5 Techniques Useful to a Petri Net Occam Based Methodology

These conclusions highlight techniques which can be beneficial to a Petri net occam based methodology.

17 Pseudo code complements Petri net graphs

Some Petri net graphs are labelled with inadequate descriptions of places and transitions. Also a small feature may not be able to be modelled effectively in Petri nets. In these circumstances, augmenting places and transitions with fuller descriptions in pseudo code, or explaining the feature in pseudo code, will assist in the understanding of the model. Section 4.6.1 discusses how Petri net graphs and pseudo code can be applied.

18 An equivalence can be made between Petri nets and occam

Figure 3-12 and Figure 3-13 show examples of Petri net and occam equivalences. Section 3.6 shows that the Petri net 'output OR' cannot be translated directly into occam, so measures to restrict this will provide structural equivalence.

19 Deadlock avoidance is advantageous

Deadlock is manifest in four ways as described in section 4.10.1. By preventing the introduction of one or more of these, then deadlock is avoided.

20 Staged development of parallel systems is desirable

It is common practice to postpone the mapping of processes to hardware until after the code is complete, refer to section 4.2.1 and 4.8.3. This allows the development of parallel systems to be done in stages.

Thus there is a need for a novel methodology to help develop flexible and dependable distributed control, and that Petri nets and occam are suitable constituents. It is also evident that a combination of the tools with prescribed use can improve confidence in the software produced. The conclusions have presented terms of reference from which the aims, techniques and the description of the methodology can be based.

5. Methodology for Design and Implementation of a DCS

5.1 Introduction

Previous chapters have reviewed tools and techniques currently available in the literature regarding the development of flexible and dependable distributed control. Chapter 4 ends by making conclusions from them, which act as a summary and terms of reference from which to base the methodology.

This chapter describes a novel methodology for developing dependable distributed control software, and begins by specifying the aims of the methodology. It discusses techniques and considerations used by the methodology to achieve the aims, and summarises them as core goals and strategies. The methodology guides the user towards a solution in four steps, each step consisting of tasks and examples. Chapter 6 discusses the steps and tasks through example and then by comparison with the aims, goals and strategies.

5.2 Aims of the Methodology

In order to alleviate many of the problems and exploit the advantages highlighted in section 4.12, the methodology aims to:

- help create the requirements of the DCS, from the manufacturing requirements, to produce the system specification of the FMC
- produce a comprehensible system specification, in a significantly graphical but formal way, which will facilitate understanding during all stages of the development life-cycle of the FMC (system specification, design, coding, mapping, installation and maintenance)
- exploit the structural similarities of Petri nets and occam to obtain an equivalence between the model and the code, thus obviating the need for formal refinement, which is attractive to industry
- prevent deadlock by reducing the specification to a single, but controlled, closed loop of communication
- produce dependable occam code which could run on a network of transputers to control and communicate that control to the manufacturing facilities of the FMC

The objective of the work described in this thesis is to produce a Petri net, occam and transputer based methodology for the comprehensible development of dependable and flexible distributed control applied to a flexible manufacturing cell.

5.3 Techniques and Considerations in the Methodology

5.3.1 Pro-activity

The formal refinement process described in sections 4.4.3.3 and 4.4.3.4, and the verification between levels of SIFT in section 4.11, are tedious and prone to errors, if done manually. Computer assistance is available to refinement, however, [McDermid 1991] refinement tools work well for only a small

application domain, are immature and were designed using small examples which do not scale up to industrial needs. He concludes that "none of these problems seem likely to prove insurmountable", and that "refinement has a realistic chance of assuring correctness". These imply that effective general purpose refinement is for the future. Also, section 4.10.4 argues that verifying the absence of faults is better than testing for all possibilities. Pro-activity aims to 'get-it-right-first-time' in four ways: comprehensibility, Petri net/occam equivalence and deadlock avoidance, see below.

5.3.2 Comprehensibility

This has many facets, including: the ability to be understood, read, visualised, and the ease of information absorption, and relates to the methodology, the system specification and the code produced. The methodology aims to make full use of the graphical benefits of Petri net formalisms, in order to allow the user to understand and absorb the whole meaning of the system specification and code, refer to section 4.5.

Where Petri nets cannot or do not enable satisfactory description, then pseudo code should be used to clarify the situation. Pseudo code, refer to section 4.6, can also be used to represent the sequential code behind a Petri net place.

Comprehensibility will lead to fewer errors being introduced, so is pro-active, and will facilitate testing and future maintenance.

5.3.3 Petri Net/Occam Equivalence

By restricting Petri nets to be equivalent to occam, refer to section 3.6, then the system specification and implementation are equivalent, thus precluding the need for formal refinement. The restriction is largely achieved by the output-work-backwards technique.

5.3.4 Concurrent and Sequential Actions

At one stage in parallel software development actions must be categorised as concurrent or sequential. Generally it is a difficult and often a crucial decision, with only a few models of parallelism to help, refer to section 4.8.5.

The requirements of the system determine the sequential actions and the overall parallelism, and the methodology begins by helping to produce DCS requirements from the system requirements. The ways in which the Petri net graphs and the occam code are structured and modularised are discussed in section 5.3.8.

5.3.5 Deadlock Avoidance

Of the four manifestations of deadlock of section 4.10.1, the methodology avoids circular wait. Output-work-backwards leads to a single circular flow of information between controllers which is presented in the overall Petri net graph. The cycle is broken, but the 'closed loop' is maintained, by controlling the communication between status handler and cell controller. This communication enables the cell controller

to be updated with the state of the cell, and the decision when to update is determined by the cell controller.

When to update is an important decision, and is dependent on the consequences of the previous decision taken by the cell controller. For example, the consequence of the decision to start the lathe, in the lathe conditions of the cell controller, only changes the lathe status in the status handler from 'lathe not started' to 'lathe turning'. The only instance of 'lathe not started' in the cell controller is in the decision to start the lathe. However, where there are more instances in the cell controller, such as 'lathe work handling idle' in the index conditions and in the lathe conditions, then the consequences of loading or unloading the lathe will change the status in the lathe statuses in the status handler, and this should be made known to the cell controller before the indexing conditions are determined. Manufacturing requirements define that indexing cannot happen while loading or unloading.

5.3.6 Output-work-backwards

Output-work-backwards focuses on and satisfies output requirements. Much of the manufacturing requirements are controlled by the workstation controllers. Outputs (instructions) from the workstation controllers manage their equipment controllers, and their inputs are from the cell controller. The outputs (instructions) from the cell controller are the inputs to the workstation controllers, and its input is from the status handler. The status handler's output is to the cell controller (the update), and its inputs are from the cell controller and the workstation controllers. Further outputs from the workstation controllers are the inputs to the status handler (feedback), and these outputs originate from inputs from the equipment controllers. Further outputs from the cell controller are the inputs to the status handler (the request). Refer to the example in section 6.3.5.

Output-work-backwards produces a 'closed loop control' which is made up of a predominant uni-directional flow of information and contains the update from the status handler to the cell controller. The single cycle with controlled feedback prevents deadlock, refer to section 4.10.1.

Output-work-backwards is a powerful technique, and can be analogous with satisfying a market need and the 'market pull' (as opposed to the technology push).

5.3.7 Steps and Tasks

It is common practice to postpone the mapping of processes to hardware until after the code is complete, refer to section 4.2.1 and 4.8.3. This allows the methodology to be constructed in a modular way, by enabling subsequent stages, e.g. mapping software to hardware, to be defined later, refer to further work.

Users will find it easier to follow a development methodology if it is in stages and steps, and if the rules are defined in detail for each step.

5.3.8 Modularise and Structuralise

Information is most easily understood in chunks and packets. Modules should be small enough to understand without loosing the identity or meaning of the information. For example, the lathe controller is shown in its entirety, and is easily understood, but the status handler is too big to comprehend, so is broken into meaningful chunks, such as the machine and pallet statuses.

Considerations of structure in the methodology relate to Petri net graphs more than to the occam code. The occam code follows the structure of the Petri net graphs, except in: the generic template, modules that hide extraneous code and some details. The structure of the Petri net graphs reflects the modularity and the concurrent and sequential actions. For example, the overall Petri net graph shows six controllers which operate concurrently. The cell controller and the status handler are drawn to the left and the right and the workstation controllers in-between. This layout highlights the uni-directional flow of information, and the single controlled feedback from the status handler to the cell controller, which helps prevent deadlock. Flexibility of expansion is facilitated by simply adding the controller Petri net graph and appending relevant conditions or statuses in the cell controller and status handler.

Reading controller Petri net graphs downwards mostly indicates a sequence. For example, Figure 6-8 shows how the lathe status can be idle, then loading, not started, machining and unloading.

5.3.9 The Need for a Cell Controller and Status Handler

The hierarchical nature of manufacturing control, refer to section 2.5, and the master/slave model of communication often used in transputer networks, see section 3.5.2, suggests the need for hierarchical cell supervisor or master. The status handler, like the rest of the cell, is slave to the cell controller. The status handler is involved in deadlock prevention by gathering feedback from the workstation controllers, acting as a buffer between cell controller and the rest of the cell, and by controlled communication between cell controller and status handler.

5.3.10 Petri Net Entry Places Become Occam Alternatives

There is an equivalence between certain structures in Petri nets and occam, as highlighted in section 3.6. The behaviour of the PARbegin and PARend of the occam PAR and the Petri net outputAND and inputAND constructs of Figure 3-13a and b are the same [Carpenter, Lau, Balbo and Xu]. Figure 3-13c shows the Petri net outputOR, which is a conflict or part of a decision and has no occam equivalence, and Figure 3-13d shows the Petri net inputOR, which is similar to the occam ALT. A mathematical proof of such equivalence is left for further work, but the relationships, called equivalence in the thesis, and their application to the methodology are examined here.

Figure 5-2a shows the pseudo code of four occam processes which run in parallel. All processes have communication within a loop. Processes A, C and D output, and process B accepts their corresponding inputs within an alternation. Process D is needed in the explanation of the actual unfair ALT implementation, but is omitted in the discussion of the fair ALT for clarity.

Fair ALT implementation

```
B
WHILE TRUE
  SEQ
    b1
    ALT -- fair
      a2b ? --cannot
        b2  --starve
      c2b ? --cannot
        b3  --starve
    b4
```

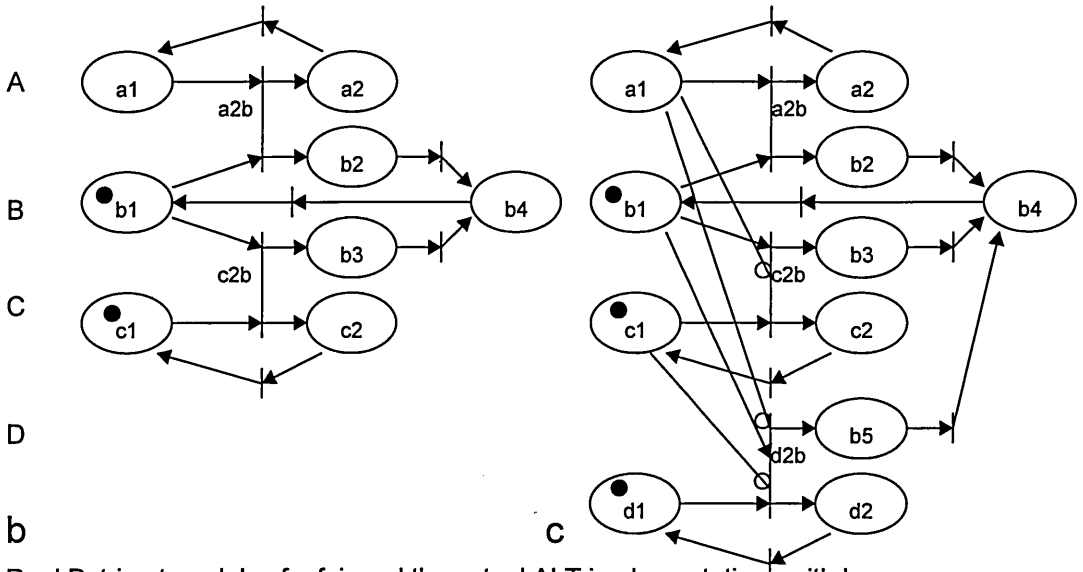
```
A
WHILE TRUE
  SEQ
    a1
    a2b !
    a2

C
WHILE TRUE
  SEQ
    c1
    c2b !
    c2
```

Actual ALT implementation

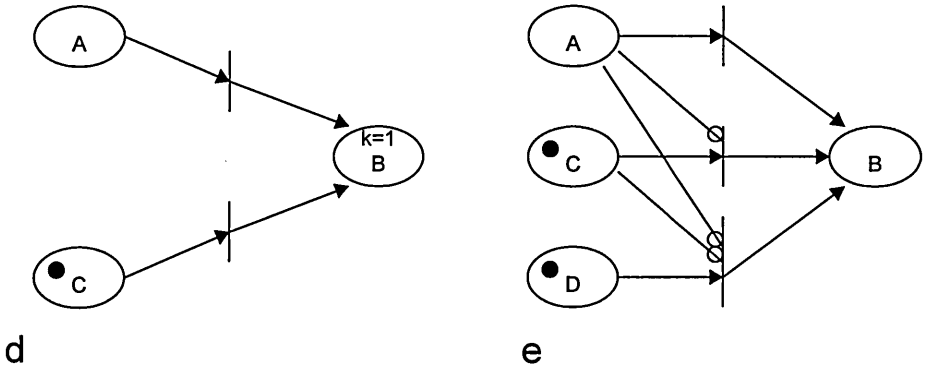
```
D
WHILE TRUE
  SEQ
    d1
    d2b !
    d2
```

```
B
WHILE TRUE
  SEQ
    b1
    ALT -- unfair
      a2b ? --cannot
        b2  --starve
      c2b ? --can
        b3  --starve
      d2b ? --can
        b5  --starve
    b4
```

a Occam models of a fair and the actual ALT implementations with loops



Real Petri net models of a fair and the actual ALT implementations with loops



Abstract Petri net models of a fair and the actual ALT implementations

The fair representation shows only processes A and C communicating with ALT process B, for clarity. The actual representation shows processes A,C and D communicating with ALT process B.

Figure 5-1 Occam (a) and abstract (b,c) and real (d,e) Petri net models of a fair and the actual ALT implementation

```
--#INCLUDE ".inc"
PROTOCOL PROT.IN
  CASE
    chan.in.tag; BOOL
    stop.in.tag; BOOL
:
PROTOCOL PROT.OUT
  CASE
    chan.out.tag; BOOL
:
PROC controller.name(CHAN OF PROT.IN chan.in,
                     CHAN OF PROT.OUT chan.out)
  BOOL Var.in,Var.out,Loop,Stop.loop:
  SEQ
    Loop:=TRUE
    WHILE Loop
      ALT
        chan.in ? CASE
          chan.in.tag; Var.in
            IF
              Var.in
                SEQ
                  chan.out ! chan.out.tag; Var.out
              TRUE
                SKIP
          stop.in.tag; Stop.loop
            IF
              Stop.loop
                Loop:=FALSE
              TRUE
                SKIP
:
```

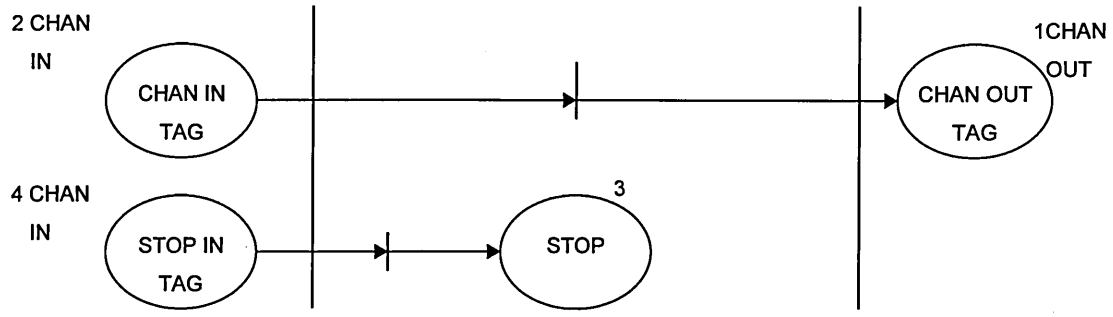


Figure 5-2 Occam template or generic controller procedure, and its Petri net equivalent

The most likely case in the FMC is when process B waits until the first of A or C is ready to output, but A or C can wait for B to communicate, or A and C can wait for B. However, if A and C are waiting for B, then B should choose A or C fairly, but the iTools implementation takes the ones written first. If this were always the case, then process A would always communicate and process C would never, and therefore starve, refer to section 3.4.4. Petri net models of the fair and actual code are given in Figure 5-1b and c, where Figure 5-1b is taken from [Carpenter]. Figure 5-1d and e are abstractions of Figure 5-1b and c, and are much more comprehensible. Place B, representing the non-deterministic ALT in Figure 5-1d, is bounded to one to prevent both transitions firing, when places A and C are marked simultaneously. The inhibitor arcs, in Figure 5-1e, ensure places drawn higher up the page have priority over those lower down.

The occam alternation construct can have zero or more alternatives [Inmos 1988], but behaves like STOP if there are none, and like SEQ if there is one. A variant protocol enables a single channel to send messages of different protocols. This is done by defining the protocol with tags, which indicate to the receiving end which data-types to expect.

The nature of FMCs determines the use of non-deterministic inputs to controllers. Single channels between pairs of controllers can be used more than once, e.g. the channel between cell controller and lathe controller (cc2lc) can be used to instruct the lathe controller to load, start, unload and terminate (stop). For clarity and because instructions can include different data, e.g. component codes, then variant protocols are used, e.g.

```
PROTOCOL CC2LC
  CASE
    load.lathe.cc2lc;    BOOL; INT::[]BYTE
    start.lathe.cc2lc;   BOOL
    unload.lathe.cc2lc;  BOOL
    stop.lathe.cc2lc;    BOOL
:
```

The status handler must be able to accept feedback and information from the workstation and the cell controllers, thus includes many channels and tags. The communication between controllers is not drawn in the combined Petri net graph, but the entry and exit places correspond to the output and input ends of the tagged channels. Unique occam channels and tag names must correspond to the Petri net place names and reference codes.

The generic template of Figure 5-2 shows the starting construction of all controllers. It shows a single tagged channel with two tags as alternatives in an ALT. The alternation is within a loop, which is terminated via the stop in tag. The template code compiles with no errors or warnings. In the methodology, the Petri net equivalent of the occam template is given at the bottom of Figure 5-2.

5.3.11 Simple and Complex Places

Safeness is a Petri net property, described in section 3.3.1.4, where places are bounded to one token per place. It is therefore desirable that all places, in a Petri net model of a DCS, to represent Boolean statuses. Much of a DCS can be modelled in such 'simple places'. 'Complex places' represent all other data-types and data structures, such as an integer, an array, a parts list, a component code or a sequential protocol.

Complex places must be distinguished from simple places in the Petri net graph to alert the users to the anomaly. Refer to section 6.3.4.

5.3.12 Pseudo Code and Place Descriptions

Petri net places should represent familiar or obvious ideas to the users of the specification, and place names should be meaningful abbreviations of the ideas. The logic represented by the Petri net structure can enhance the meaning, or allow more succinct abbreviations, of place names.

Where there is doubt that a place name represents a familiar or obvious idea then it should be accompanied by a further description. Where there is doubt that the Petri net structure does not easily express the logic required, then pseudo-code should supplement the Petri net graph.

Users must be informed where the Petri net graph is augmented. Refer to section 6.3.4.

5.3.13 The Human Computer Interface

In sequential programs, or a single process, communicating via the screen and keyboard offers few problems when compared with distributed message-passing programs. This is because, each concurrent process competes for the channel to the screen and the possibly the channel to the keyboard. A multiplexer can resolve the competition, but the channels between the process and the multiplexer remain. Such channels are often ignored at the design stage, and communication with the user can alter a design significantly.

In the methodology, the communication with the screen and keyboard is confined to the status handler. The status handler is designed like a multiplexer, and acts as a storage of information and as a buffer between the cell controller and the rest of the cell.

5.3.14 Testing

The methodology addresses testing at a number of levels. The levels are reviews, consistency checking, integrated test strategy, formal verification and validation. No integrated test strategy has been developed and is left for further work.

Reviews

A review would reconsider the implementation of the rules of the methodology, like examining the grammar and layout of this thesis. The simple rules of the methodology, core goals, strategies and the graphical design tool facilitate the production of comprehensible designs and code, which are consequently easy to review.

Consistency Checking

A deeper understanding is needed to examine the logic or control law, like checking the contents of this thesis and its cross references. The methodology adopts the rules of the mathematically based tool, Petri nets, and provides other rules to enable consistency checking (e.g. entry and exit place reference codes). Entry and exit places in controller interfaces and their replication (refer to section 6.3.4) are obvious danger areas. Consistency checking will fail if different ideas are represented in the same way or given the same label, or if one idea is represented by different labels.

Formal Verification

The formal refinement process is criticised in section 4.12.4. The goal of pro-activity and Petri net/occam equivalence precludes the need for formal verification, as discussed in sections 5.3.1 and 5.3.3.

Validation

Validation by animation is a property of Petri net graphs, but the transformation from Petri net graphs into occam code is easier in the methodology than to produce an animation tool, refer to section 3.3.1.2, so the validation is done as an occam simulation. This has the added advantage of including temporal properties and priorities.

5.3.15 Timing

The operation of an FMC is in real time, refer to section 2.7. Timing is not addressed fully in the methodology, but time delays are applied in the occam simulation. A detailed examination of timing and its integration in the methodology is left for further work.

One major criticism of the PC/LAN based DCS, refer to section 2.6.1, is in its timing uncertainty, which leads to uncertainty in sequencing of operations. Such uncertainty, intolerable in safety systems, is not a problem in transputers, which have real time clocks and deterministic execution. The control law of the manufacturing logic in the example FMC precludes the need for timing at the design stage, and relies on synchronisation of pre-determinable concurrent operations.

In an occam simulation, delays represent durations of machining and work handling operations. In real time control, the end of the machining and work handling operations is relayed to the appropriate level 2 controllers via the level 1 PLCs, which monitor the operations.

5.4 Core Goals and Strategies

On examination of the aims and techniques of sections 5.2 and 5.3, it is possible to define core goals which run through the methodology, and define strategies which help realise the goals. The methodology, refer to chapter 6, defines four steps which make use of the goals and strategies. Three of the aims are significant and apply to more than one stage in the development life cycle. These 'core goals' can be summarised as:

Goal 1 Petri net/occam equivalence

A restricted use of Petri nets enables an equivalence with occam constructs, and is enabled by Strategy 1 output work backwards.

Goal 2 comprehensibility

This aims to enable understanding of the methodology, readability of the Petri net graphs and occam code to arrive at a better solution. It is manifest in Strategy 3 structuralise and modularise, Strategy 2 concurrent and sequential actions and in the simplicity enabled by the narrow problem and solution domain.

Goal 3 pro-activity

This aims to pre-empt issues in DCS development by 'getting it right first time', thus precludes the need for formal refinement. It is manifest in Goal 1 Petri net/occam equivalence and Strategy 4 deadlock avoidance.

The strategies to achieve the goals of the methodology are:

Strategy 1 output work backwards

Output is of primary concern, so conditions must be determined to satisfy the requirements of the output. Working backwards, the inputs to the condition become of concern, and must be satisfied in turn. It assists in Goal 1 Petri net/occam equivalence and Strategy 4 deadlock avoidance

Strategy 2 concurrent and sequential actions

The actions required of the DCS are separated into those best done concurrently and those best done sequentially. This assists in Strategy 4 deadlock avoidance and is implemented in rule 1 of step 1.
The controller outputs are largely arranged sequentially downwards in the Petri net graphs. This is implemented in rules 4 to 7 in step 2 and step 3, but is best explained in section 6.3.5.
Both of these are manifest in Strategy 3 structuralise and modularise and assist in Goal 2 comprehensibility

Strategy 3 structuralise and modularise

Defining Petri net graph boundaries, and input and output communication though boundaries, enable modularisation. Structural arrangements between and within controller Petri net graphs improve Goal 2 comprehensibility, Strategy 2 concurrent and sequential actions and Strategy 4 deadlock avoidance.

Strategy 4 deadlock avoidance

The predominant uni-directional flow of communication between cell and workstation controllers, workstation and status handler controllers and controlled communication between cell and status handler controllers avoids deadlock. Strategy 3 structuralise and modularise facilitates its comprehension.

5.5 The Description of the Methodology

The three core goals of the methodology are Petri net/occam equivalence, comprehensibility and pro-activity, and are evident throughout the steps. The methodology is restricted to facilitate the development of occam code for the purpose of running on a transputer based DCS, therefore the methodology starts when the manufacturing requirements are complete. The example requirements are that of the PC based FMC, refer to chapter 2.

The methodology consists of four steps:

- Identify concurrent and sequential operations.
- Produce a Petri net graph for each controller.
- Combine the controller Petri net graphs.
- Translate the Petri net graphs into occam code.

Each step consists of one or more tasks. The tasks are defined, and are explained with examples (in italics) and with reasons for their inclusion. The examples mostly involve the cell controller's instruction for the lathe to be loaded with a component, or the lathe controller itself. Further discussion relating to the methodology and the implementation of its steps is given in chapter 6. Font differences relate to Petri nets or occam.

Step 1 is concerned with the analysis of the overall concurrency of the system, to identify the concurrent tasks in FMCs. The cell controller and status handler are imposed.

Step 2 examines the requirements of controllers of the concurrent operations. Petri net graphs are produced, for each controller, that explicitly state the inputs to the controllers, the logical operation of the controllers and the outputs from the controllers. They also state from where such inputs come and to where outputs go.

Step 3 is concerned with the synthesis of the controllers and with producing a combined Petri net graph, which is the DCS specification. Managing the communication between the closed loop of control between controllers prevents deadlock.

Step 4 translates the Petri net graph into occam.

By following these steps, the Petri net graphs for the FMC, Figure 5-7 to Figure 5-11, Figure 6-1 to Figure 6-11 and the overall Petri net graph, and the occam code, refer to the appendices, were produced.

5.6 Step 1- Concurrent and Sequential Operations

5.6.1 Task 1 Identify concurrent and sequential operations

Identify those operations that can beneficially occur concurrently, and those that do not or must not happen concurrently.

Some operations are required to be concurrent from the requirements of the FMC, e.g. lathe machining and miller machining. For some operations it will be obvious that they can be concurrent because they are independent, e.g. lathe machining and conveyor indexing. Examples of those operations that should never be concurrent or are best kept sequential are mostly obvious: loading the lathe before lathe machining before unloading the lathe, and loading a part into a pallet before transporting the part to the lathe.

The reasons for identifying concurrent and sequential operations are to:

- make it apparent which operations can be controlled on separate transputers and which are best executed on the same transputer.
- make the concurrency clear and understandable during design, modification and checking.

5.6.2 Task 2 Create cell controller and status handler

Two concurrent operations defined by the methodology are the cell controller and status handler.

The cell controller makes decisions and issues instructions to the rest of the DCS. It determines the appropriate action to take based on the current state of the cell, which it requests from the status handler. The status handler receives feedback from the workstation controllers and updates the statuses of the cell, and, when requested by the cell controller, sends the cell statuses to refresh the cell controller.

The reason for the cell controller and the status handler is given in section 5.3.9

In the FMC example, six controllers are wanted: cell, status handler, robot, lathe, miller and conveyor.

5.7 Step 2- Produce Petri Net Graphs for each Controller

5.7.1 Task 1 Identify the outputs for the controller

Outputs are taken from the controller as defined in step 1.

Controller outputs can be inputs to other controllers, e.g. the cell controller has an output to the lathe controller. Outputs from level 1 controllers can also be inputs to operations, which can be found from the manuals of the manufacturing equipment, e.g. the lathe can be started, stopped and can have two user defined inputs.

The outputs from the controllers are instructions to manufacturing devices or other controllers and feedback to the status handler, and will be made into transputer links in the distributed implementation.

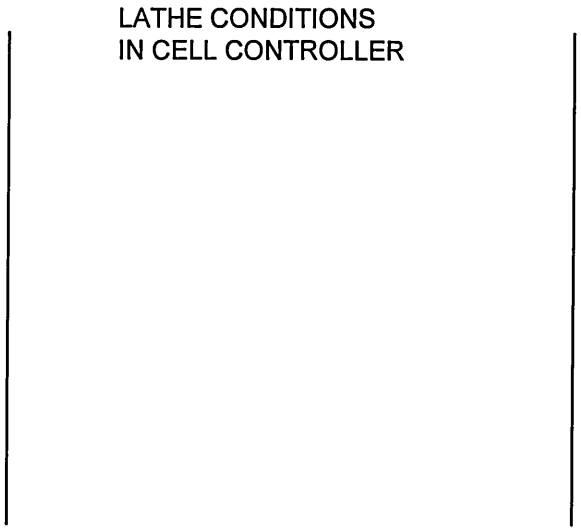


Figure 5-3 Rule 2 of step 2 - draw controller boundaries

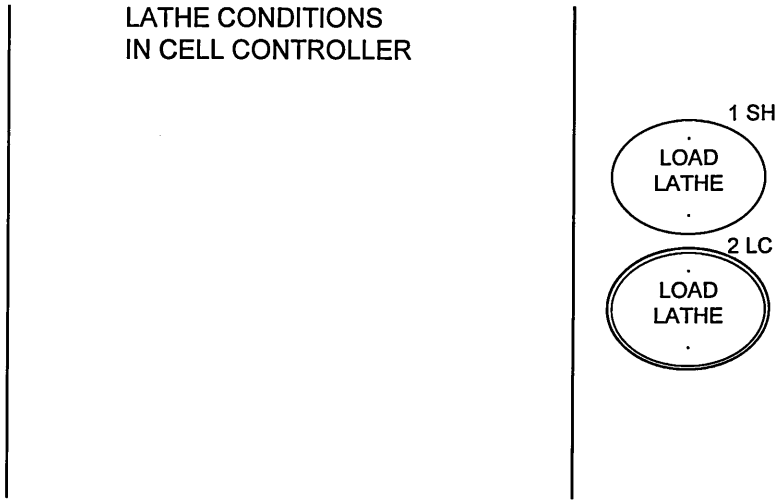


Figure 5-4 Rule 3 of step 2 - draw exit places

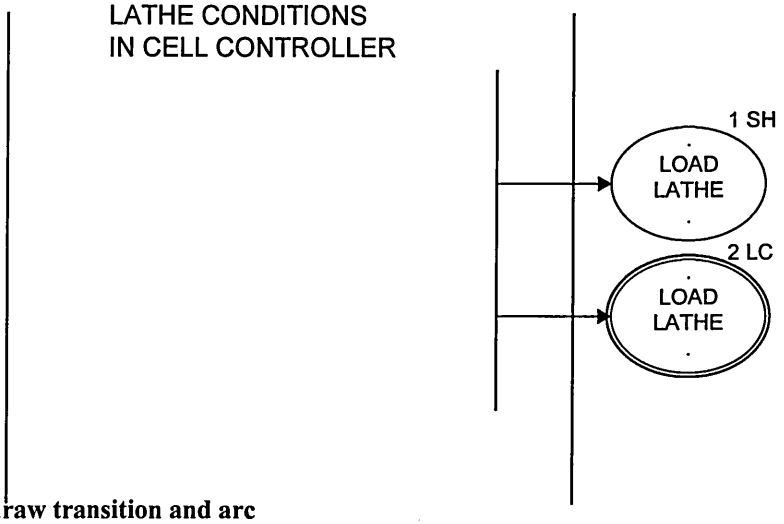


Figure 5-5 Rule 4 of step 2 - draw transition and arc

5.7.2 Task 2 Draw the controller's boundaries

Draw the controller's boundaries.

Position the vertical lines of the 'output boundary' to the right of the Petri net graph, and the 'input boundary' to the left, refer to Figure 5-3.

The lines are drawn to emphasise the extremity of control of the sub-system.

5.7.3 Task 3 Draw 'exit places'

Draw 'exit places' and their 'destination codes' and 'reference numbers' to the right of the output boundary, and name places explicitly. Places are either: simple - drawn with a single circle; consolidated (refer to task 8); or complex - other than simple and complex and drawn with double concentric circles.

Exit places are outputs from the controllers identified in task 1, and are positioned with liberal spacing in a vertical line, refer to Figure 5-4. Places should be drawn, from top to bottom, in the sequence that they are to be executed. Each destination code is written to the right of its exit place to show with which controller or device it is communicating and to which Petri net graph to refer, e.g. the destination code for exit place 'load lathe' is 'LC' for the lathe controller. Include reference numbers, with destination codes with the places in sequence as they are drawn.
Places are drawn with a single circle if they represent a simple place. If they represent other than a simple status, e.g. include a file as in 'load lathe', or an abstraction such as 'pallet at lathe' (which represents the statuses of Figure 6-13), then places are drawn with double concentric circles. Appropriate and succinct place names should be chosen, as discussed in section 6.3.4. load lathe is familiar to users of the FMC, but unexpected places names, such as pallet at lathe miller associated in the status handler of Figure 6-13, require further explanation, which is provided in the accompanying place name description (not done here).

The exit places highlight the communication requirements, and the places' names and destination codes represent the tag and the channel names in occam in step 4. The destination codes can help identify devices or controller Petri net graphs which have not yet been considered. The reference numbers help during step 2 (refer to task 6) and step 4, by enabling referencing and by providing a history of the design. Use of appropriate place names and of the discretionary accompanying place name descriptions are to improve comprehensibility. The description aims to preclude misunderstanding and inform users of unfamiliar terminology or ideas. Double and single circled places represent the data requirement. Single circle represents an occam Boolean. Double circles are either consolidated places (refer to task 8) which represent channels with sequential protocols, or complex places which represent channel protocols other than Boolean.

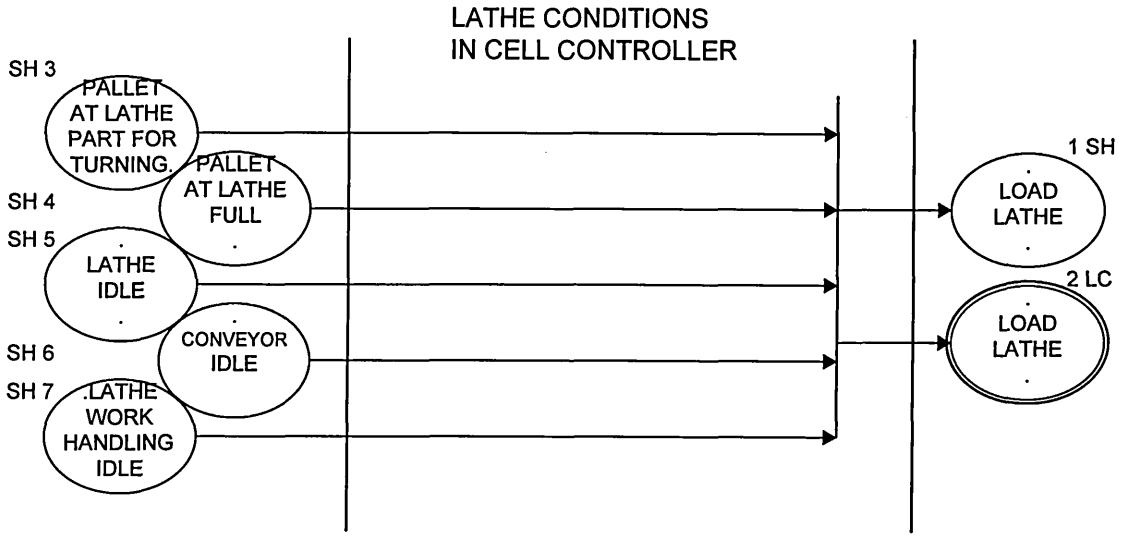


Figure 5-6 Rule 6 of step 2 - draw input places and arcs

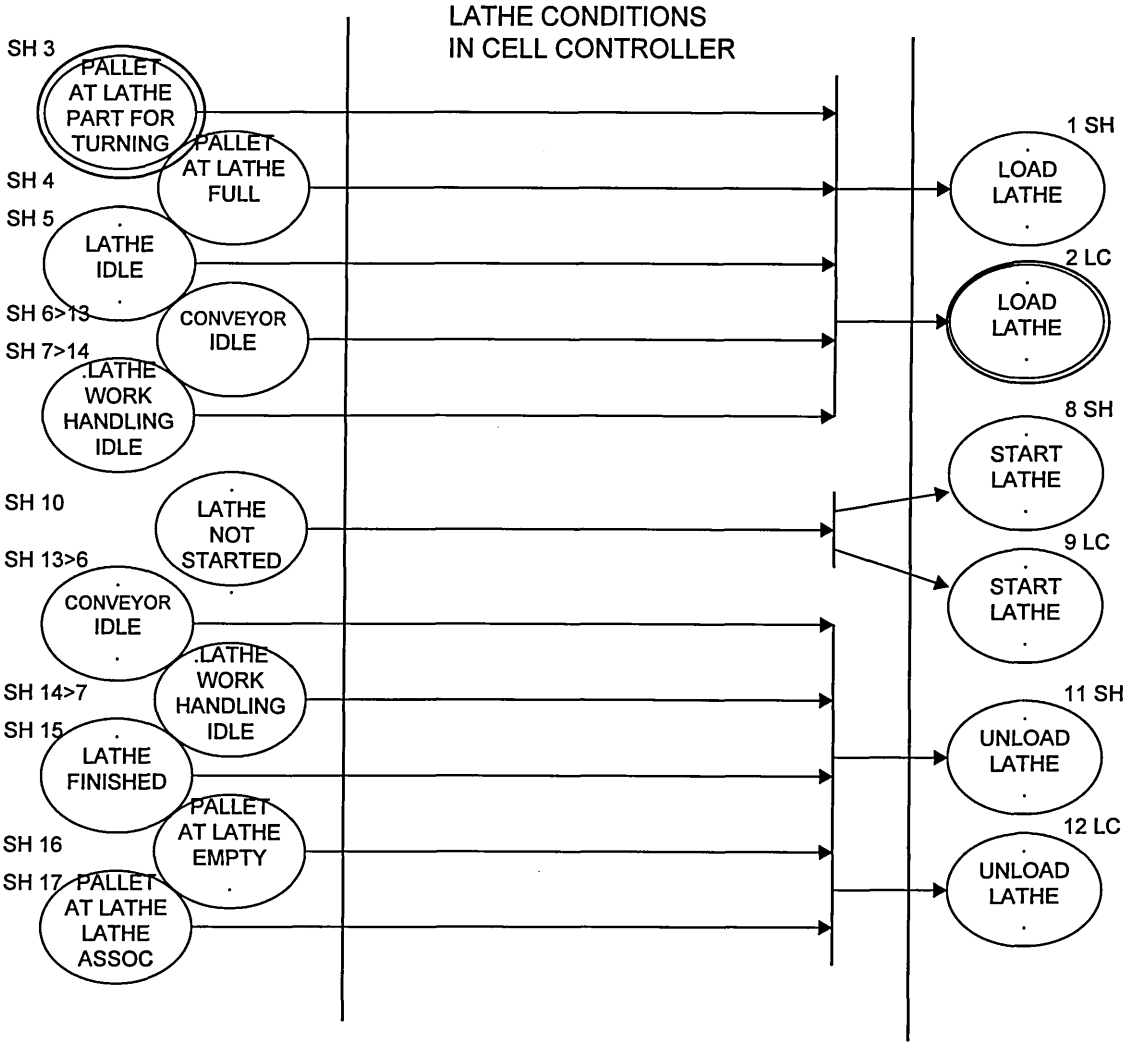


Figure 5-7 Conditions to load, start and unload the lathe, in the cell controller, with duplicate places

5.7.4 Task 4 Draw a transition and arc for each exit place

Draw a transition and arc for each exit place.

The arc is drawn horizontally, pointing left to right, and the transition is drawn vertically. Each arc will cross the output boundary of the controller, refer to Figure 5-5.

The arc crossing the output boundary indicates a link outside of the controller. Transitions can be numbered during step 2 and 4 to facilitate referencing and provide a history of the design, but this is not done in the thesis.

5.7.5 Task 5 Determine transition inputs

Determine the inputs to the transitions (of task 4) necessary before each exit place is enabled.

Identify all of the operations that must have happened or statuses that must hold immediately prior to the exit place being enabled. The condition may depend on operations or statuses from within or from outside the controller. For example, Figure 5-6, the condition that must be satisfied before the lathe can be loaded requires: a pallet waiting at the lathe, the pallet being full with a part, the part requiring turning, the lathe waiting and the local work handling (transferring component between lathe and conveyor) waiting.

The transitions become occam conditions, and are implemented as IF constructs, refer to task 5 of step 4.

5.7.6 Task 6 Draw the transitions' input places and arcs

Draw the places and arcs that input to the transitions of task 5.

Make each operation or status identified in task 5 into a place, and number it. Those places that only exist in the controller must be drawn within the boundaries of the Petri net graph of the controller, and those that come from outside the controller become 'entry places' to the controller, and are drawn to the left of the input boundary of the Petri net graph. 'Source codes' are written to the left of the entry places to indicate with which controller or device they are communicating and to which Petri net graph to refer. For example in Figure 5-6, places have been chosen to represent 'pallet at lathe full', 'pallet at lathe part for turning', 'lathe idle', 'lathe work handling idle' and 'conveyor idle'. They are all from the status handler which is outside the cell controller, so are drawn as entry places, and the source code is written as 'SH' for status handler.

It may be necessary to replicate places to reduce the frequency of arcs crossing each other, and thus improve visualisation. If done, the replicated places must reference each other. It is sometimes better not to replicate, as in resource sharing. For example in the cell controller of Figure 5-7, exit places 'load lathe' and 'unload lathe' output from transitions which share the entry place 'lathe work handling idle'. It is possible to duplicate this entry place and indicate that place 7 is replicated with place 13 and that place 13 is replicated with place 7. The same is true for place 6 and 14 'conveyor idle'. Compare this place replication with a solution with arcs crossing of Figure 5-8.

The data requirement, indicated by places with single or double circles, follows that of task 3. However, places shown within the controller's boundaries are obviously local to the controller, and are to be occam variables. Entry places, that enter from the input boundary, are to be occam channels, and subsequently transputer links. Entry places can be staggered to save space.

5.7.7 Task 7 Repeat tasks 4,5 and 6

Repeat tasks 4,5 and 6

Repeat tasks 4,5 and 6 until all places are either entry places (exit places of other controllers or devices) or are preceded by a transition, and until all conditions are satisfied. This repeating process is cyclical within a controller except at the entry and exit places.

This cycle ensures that places are preceded by output arcs, which are proceeded by transitions, which are proceeded by input arcs, which are proceeded by places again, which is a requirement of Petri nets.

5.7.8 Task 8 Consolidate entry and exit places

Examine the entry and exit places and their reference codes, and combine those which communicate with the same controllers and can communicate at the same time. Consolidate these entry and exit places into a single entry or exit place and draw it with double circles. Provide local places for these consolidated entry and exit places (now called entry places made local and exit places made local).

The update between the status handler and cell controller can be a mass transfer of information through the same communication. The exit places of the status handler are consolidated into the single exit place, update, which is given a double circle. All of the other exit places are made local, and are drawn just inside the output boundary. The entry places of the cell controller must also be consolidated into a double circled entry place, made into local places and drawn inside the input boundary.

Entry and exit places represent the input and output of an occam channel. Where source and destination codes reference the same controller, but communication is at different times, then they become tagged channels. Where codes reference the same controller, but communication is at the same time, then they can be consolidated into a single occam channel with a sequential protocol, and the newly made local places become the communicated occam variable (called Input and output variables). Local places corresponding to the entry and exit places can be drawn to represent the input and output occam variables sent or received through the non-consolidated channels, but will clutter the Petri net graphs and do not add to comprehensibility.

5.7.9 Task 9 Set initial conditions

Define the initial state of the controller by marking the Petri net graph with tokens, according to the manufacturing requirements.

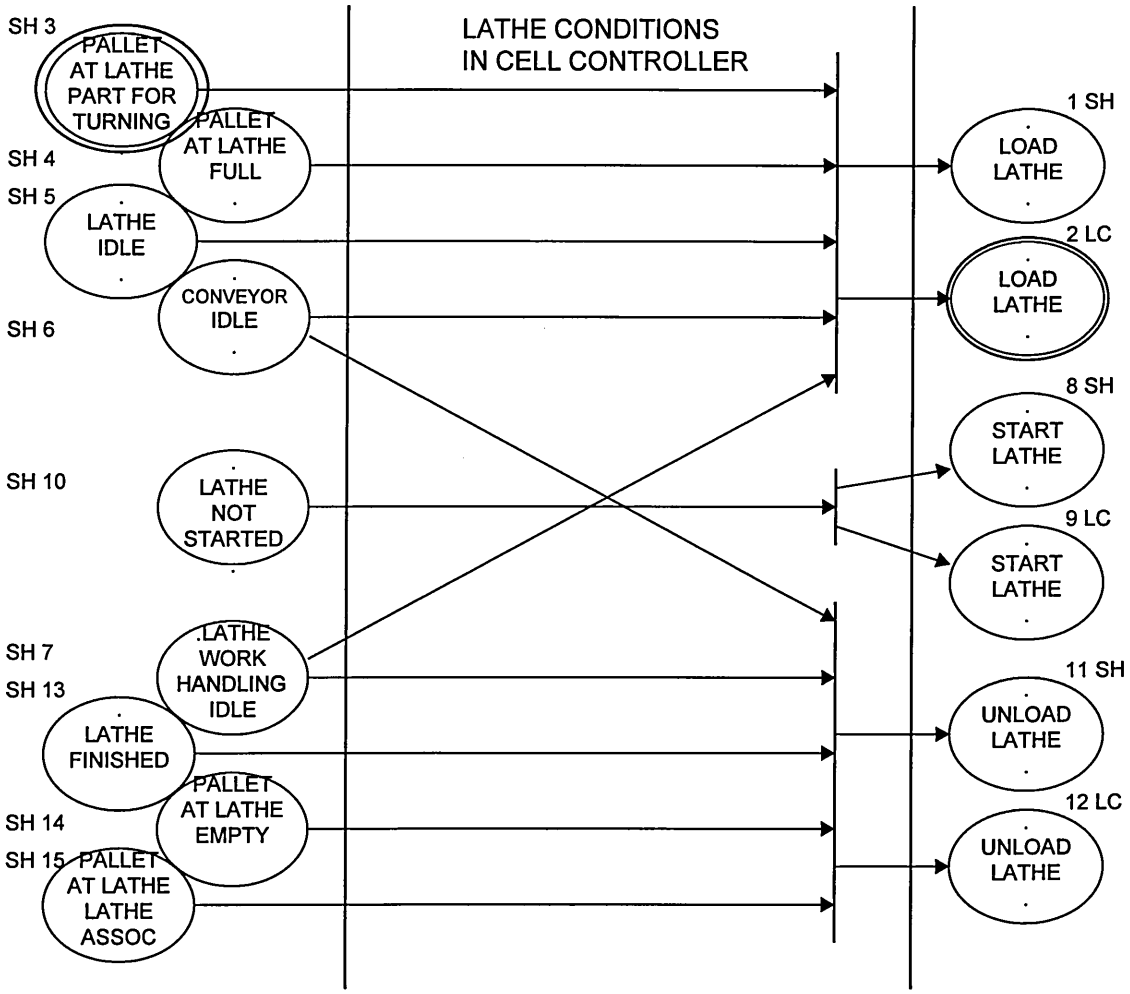


Figure 5-8 Conditions load, start and unload the lathe, in the cell controller, showing mutual exclusion rather than duplicate places

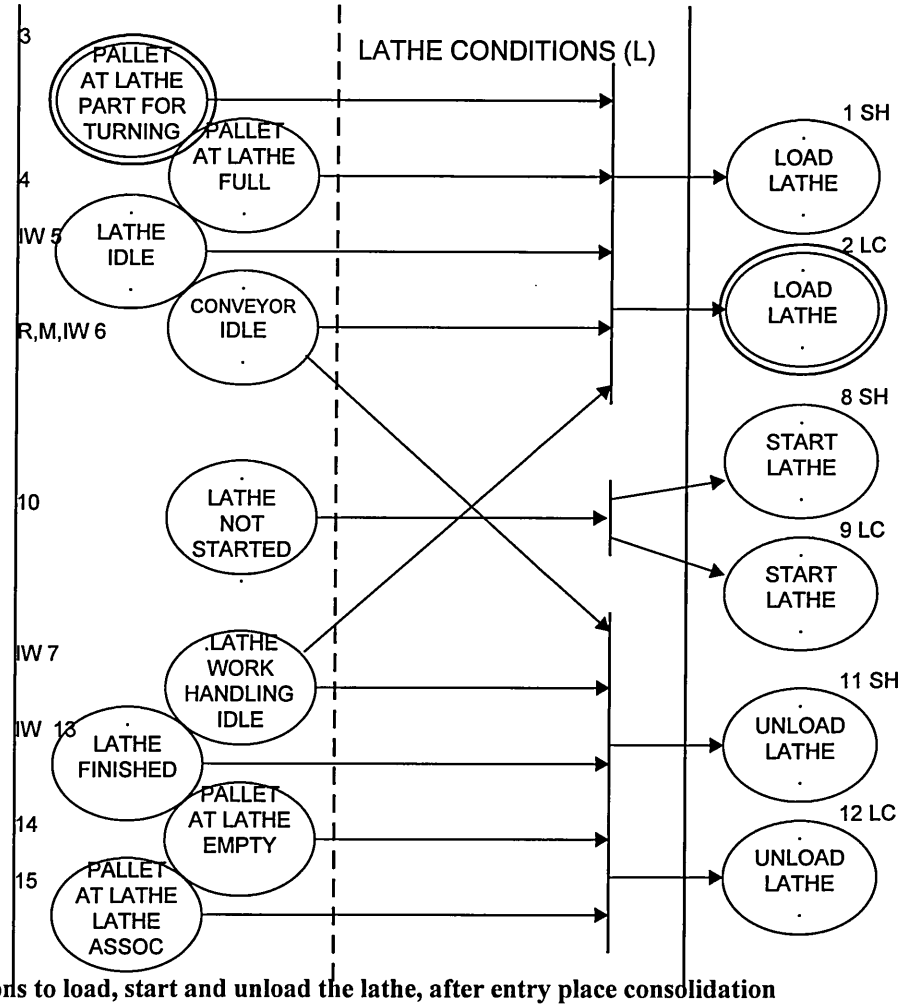


Figure 5-9 Conditions to load, start and unload the lathe, after entry place consolidation

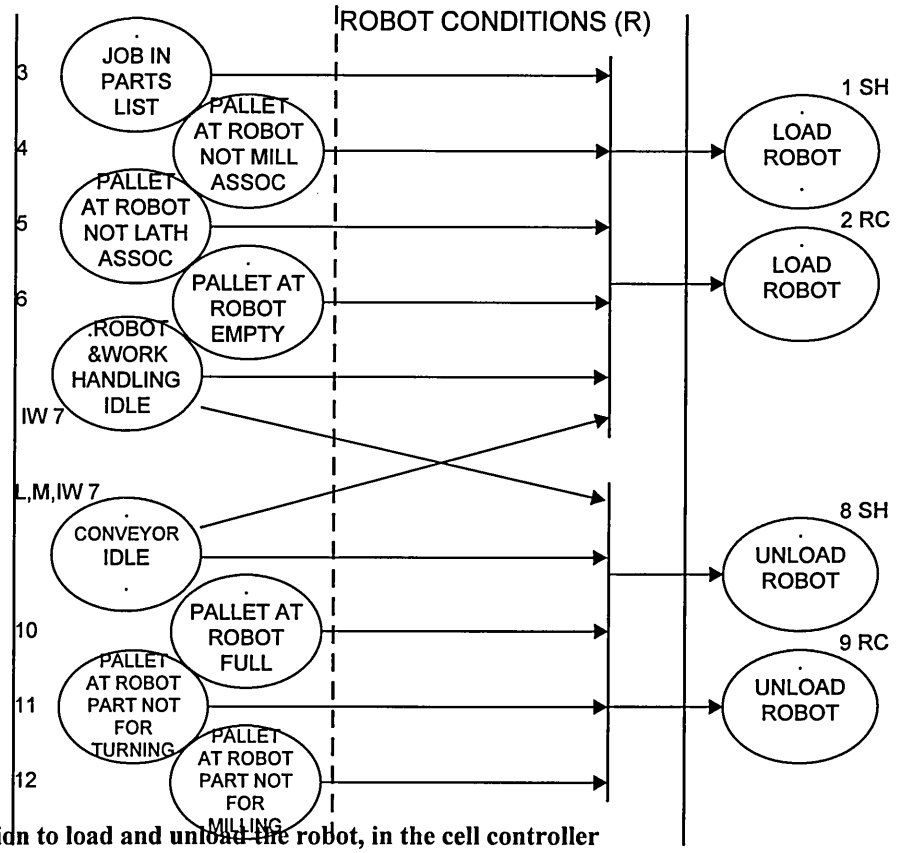


Figure 5-10 Condition to load and unload the robot, in the cell controller

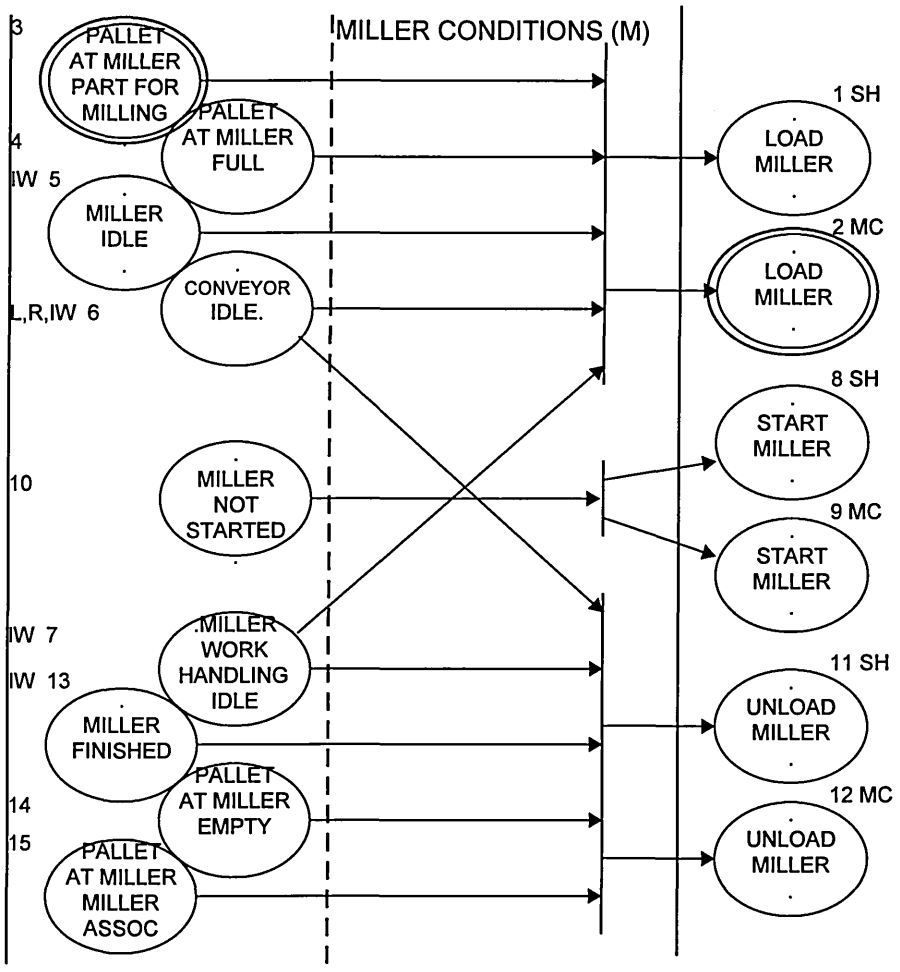


Figure 5-11 Conditions to load, start and unload the miller in the cell controller

In the example FMC, virtually all initial conditions are set in the status handler. The Petri net graphs for the robot and miller conditions are given in Figure 5-10 and Figure 5-11.

The initial marking in the graph becomes the initial conditions in the occam code.

5.8 Step 3- Combine Controller Petri Net Graphs

Design the Petri net graphs for the whole DCS by combining and positioning the controllers' Petri net graphs, produced in step 2. Where controllers are concurrent but do not communicate, then they can be positioned vertically. Where controllers are concurrent and communicate, then they can be positioned horizontally with the communication being predominantly left to right, according to the reference codes of the entry and exit places. It may be possible to alter a controller, to improve visualisation, by consolidating entry or exit places, changing the position of places or transitions, or by replicating places (refer to task 6 step 2). When satisfied with the synthesis of the Petri net graphs, the reference numbers can be re-ordered, or be left to denote the history to help in future maintenance.

The combined Petri net graph for the FMC example, refer to the overall Petri net graph, is laid out in three vertical portions. The middle portion consists of the Petri net graphs for the lathe, miller, robot and conveyor controllers. The reference codes of the controllers' entry places are either outputs from the level 1 controllers or exit places from the cell controller, so the cell controller is positioned to the left of the workstation controllers. The exit places of the workstation controllers refer to the inputs to the level 1 controllers or to the status handler, so the status handler is positioned to the right. Communication between controllers has all been defined from left to right except between status handler and cell controller, where the request by the cell controller of the status handler to send data is two way. This is a break in the uni-directional inter-controller communication and in the closed loop control, and it is the management of the break which prevents deadlock.

5.9 Step 4- Translate Petri Net Graphs in to Occam

5.9.1 Task 1 Preserve Names

The names in the Petri net graphs become the names in the occam code. Petri net module names become occam procedure names. Entry and exit place names and their reference codes become channels and tags, and local place names become variable names. Conserving names helps comprehensibility.

The reference codes of the entry and exit places build into tagged channel names. For example, the cell controller instruction 'load lathe' is referenced 'LC' and is received by entry place 'load lathe' with reference 'CC' of the lathe controller, so the channel name is '`cc2lc`' with tag '`load.cc2lc`', and is defined in file `latheHd.inc` in the appendices.

Variable names are taken from the local place names, including entry and exit places made local, and variables sent to or received from are given the same name as their tag.

Data types of channels and variables are determined by the places' circle and pseudo-code. A single circle signifies a simple status, because a place either holds or does not. A double concentric circle signifies other than a simple status, refer to task 3 of step 2. Occam has Boolean, byte, integer and real data and channel types, a timer type and array, file and user defined data structures. Simple places represent Boolean variables, and simple entry and exit places represent Boolean tagged channel protocols. A complex place, such as exit place load lathe will have a tagged protocol of `load.lathe.cc2lc; BOOL; INT::[]BYTE`, which enables the component code or part program to be sent. A consolidated place, such as update, will have a sequential protocol, refer to file `CCandSH.inc`.

5.9.2 Task 2 Specify the overall procedure

The combined Petri net graph becomes the main occam procedure, see file `top.occ` in appendices. The controllers, identified in step 1 and defined in task 3 below, are made to run in parallel, and are instanced (called) with actual arguments being the same as the formal arguments in task 3.

5.9.3 Task 3 Specifythe controller procedures

The template of Figure 5-2 is the starting point for all controller procedures and their header files. Each controller is defined as a separate procedure, and written in a separate ASCII file. The procedure name is the same as that in the Petri net graph, and its formal arguments are the tagged channels which are the entry and exit places of the controller.

For example, the lathe controller is defined in file `latheCon.occ` in appendices. Its formal arguments in the simulation code, indicating communication with the cell controller and status handler, are two tagged channels with names, tags and data types (refer to task 1) as follows:

- `cc2lc` with tags load, start and unload
- `lc2sh` with tags loaded, stopped and unloaded

Tagged channels allow communication to be grouped in to logically similar and more manageable channels. In the example, they are grouped by entry to and exit from the controller and communication with the cell controller, status handler and lathe controller.

5.9.4 Task 4 Entry and exit places become channels

For each controller, the entry places (refer to task 6 step 2) become tagged channel input alternatives in an ALTernation. The `ALT` construction is placed in a `WHILE` loop. The exit places (refer to task 3 step 2) become tagged channel outputs, and are implemented in the process part of the `IF` condition (refer to task 5). File `latheCon.occ` in the appendices shows that the tags of channel '`cc2lc`', defined in header file `latheHd.inc`, are alternatives `load.lathe.cc2lc, start.lathe.cc2lc and unload.lathe.cc2lc`, and represent the entry places 'load lathe', 'start lathe' and 'unload lathe'. These communicate with the corresponding exit places of the cell controller.

Where entry or exit places are drawn with double concentric circles (refer to task 1), then provision must be made for appropriate inputting or outputting of information. Protocols for the above tags are `BOOL;` `INT::[]BYTE, BOOL` and `BOOL` respectively.

5.9.5 Task 5 Transitions become IFs

All transitions become IF conditions. Transitions (refer to task 4 of step 2) are coded as either: a condition whose output process remains within the controller, or a condition whose output process includes a channel output (an exit place). Where entry places are inputs to transitions, such as 'load lathe' in the lathe controller of file `latheCon.occ` in appendices, then the `IF` construct must sequentially follow the `ALT`. However, where an instruction is merely passed on, as in the lathe controller 'start lathe', then it is simpler to ignore the `IF` and put the channel output as the process in the alternation. The IF is left in, and shows the `delay` (which simulates transfer or machining times) and output channel in the tested guarded process.

5.9.6 Task 6 Set initial conditions

Provide a procedure that defines the initial state of the FMC according to the marking specified in task 9 step 2.

The initial conditions of the example FMC are in a procedure held in an included file used by the status handler, refer to file `statHand.inc` in the appendices.

6. Discussion of the Methodology

6.1 Introduction

Four approaches are taken in the discussion of the methodology, and are:

- how and why the decisions were taken for each step during the development of the DCS of the example FMC, in sections 6.2 to 6.5
- what were the consequences of the goals, strategies and steps, and general observations, in section 6.7
- whether the methodology applies to other systems, in section 6.6
- how the methodology compares with the applications reviewed in section 4.7, in section 6.8

The chapter begins by discussing the decisions made during each of the four steps. These help explain the steps and provide examples for the overall discussion. Font differences relate to Petri nets or occam.

6.2 Step 1 - Analysis of the FMC

Properties of FMCs, given in section 2.2 and exemplified in section 2.3, are that operations which contribute to the manufacture of components can operate concurrently and can be made to co-operate. The manufacturing processes of turning and milling can occur at the same time as component transportation, decision making and monitoring. Other activities that can happen concurrently are materials handling, but these are heavily dependent on the processes which they serve. Materials handling operations for the lathe, miller and conveyor are: lathe loading and unloading, miller loading and unloading and loading and unloading the conveyor by the cell robot. The lathe and miller, each must be loaded with a component before the component is machined, and the machined component must be unloaded in readiness for the next component. A sequence also exists for transportation, where each pallet must be loaded with a component by material handling devices, before the pallet is transported to its destination and eventually unloaded. Machining durations vary to a much greater extent than transportation and materials handling transfer times. When the lathe and/or miller are not machining they are waiting to be loaded or unloaded (or started) with components which are transported by the conveyor. These are the manufacturing requirements.

The purpose of the DCS is to enable the manufacturing operations to co-operate, so the requirements of the DCS are to facilitate the manufacturing requirements. Synchronisation, a major function of the DCS, is required between the lathe transfer device and the lathe while loading, and between the lathe transfer device and the lathe while unloading. This is also true for the miller and its transfer device, between the conveyor and cell robot, between the conveyor and the lathe transfer device and between the conveyor and the miller transfer device.

Sequential and concurrent tasks and the synchronisation between them determine how best to separate and group operations. The lathe, miller and conveyor can operate concurrently with the cell controller and status handler. There is a sequence of load, operate, unload which is achieved via synchronisation

between processes (lathe and miller) and their transfer devices. The conveyor must synchronise with the cell robot, the lathe transfer device and the miller transfer device. The conveyor is also involved in the sequence of robot load, transportation, robot unload, but the conveyor and cell robot should be treated separately. The sequences should be grouped together and be managed by their own controllers, so there are the following concurrent controllers: lathe, miller, robot, conveyor, cell and status handler. The names of the controllers are chosen to be descriptions of their function to enhance comprehensibility.

6.3 Step 2 - Controller Petri Net Graphs

6.3.1 Starting Approaches

Of the conventional starting approaches, such as top-down, bottom-up, breadth-first and depth-first, only breadth-first seems plausible. However, the creation of the strategy 'output-work-backwards' was motivated from the need for a suitable starting approach, and this and breadth-first are discussed:

In a **breadth-first** approach, the choice of a particular operation can be made first, like loading, and an examination made of all loading operations in the lathe controller, miller controller, cell controller and status handler, before choosing another operation, such as unloading. Which ever operation is chosen, it is taken in isolation. This might suit sequential programming, but concurrent software development requires a holistic approach.

The approach recommended is '**output-work-backwards**'. Here you start with outputs from one controller and work backwards satisfying their needs, as described in section 5.3.6 and exemplified in section 6.3.5. It is best to begin with a workstation controller, because the methodology helps design the DCS from the manufacturing requirements.

Once the starting approach is chosen, then the choice of which controller to begin must be made. Users might begin with the controller which they feel most confident or knowledgeable. For example, this thesis makes reference mainly to the lathe workstation, because it was completed first, and consequently was the first to be modelled.

6.3.2 Communication

The methodology addresses communication between controllers in the Petri net specification in the same way as occam does. Rather than explicitly drawing the arcs between the controller Petri net graphs, communication is made by reference. Petri net exit places become occam input processes and entry places become entry places become occam input processes. This avoids many arcs crossing in an intricate overall graph. It also leaves open decisions regarding introducing partitioning and configuring into the methodology, which is for further work.

Occam provides three types of channel protocol: simple, tagged and sequential, and the methodology provides three types of exit and entry place: simple, complex and consolidated. Where the occam simple and tagged protocols can be any literal data type and are distinguished by the data structure, the simple and complex places model the simple and tagged protocols in the same way, but are distinguished by

their data types: a simple place is Boolean and a complex place is anything else. Consolidated exit and entry places represent the occam sequential protocol and, like the complex place, are drawn with double concentric places.

6.3.3 Petri Net and Occam Differences

The occam template at the top of Figure 5-2 is the basis for all controller modules, and its Petri net model is shown below it. The Petri net entry places are represented as tagged input channels within an ALT construct. The input variables are given the same name as the input channel, and are tested in a conditional IF construct. Output channels represent Petri net exit places, and send output variables from within the tested guard of the IF. The stop tagged channel is also part of the ALT construct, but does not have a corresponding output channel. Instead it changes the variable of the WHILE loop, which contains the ALT construct, from TRUE to FALSE.

6.3.4 Naming and Referencing Conventions

The naming of controllers, places and communication between controllers must be consistent in order to be as comprehensible as practicable, because the names will represent ideas in the Petri net graphs and in the occam code.

Naming

The controller names should represent the concurrent controllers identified in step 1 for the purpose of comprehensibility, and are lathe, miller, robot, conveyor, cell and status handler. Place labels or names should be designed to be understandable and meaningful. Instructions to controllers are labelled in the present tense and feedback to the status handler is in the past tense. For example, 'load lathe' and 'lathe loaded' in the lathe controller of Figure 6-2, where the instruction arrives from the cell controller and is passed to the level 1 controller, and when the operation is complete, then the feedback is received from the level 1 controller and sent to the status handler.

In the controller Petri net graphs, the Petri net structure and the Petri net place names are synergistic. For example, the lathe statuses of Figure 6-8 shows that when the lathe is loaded, then the lathe is no longer idle and it becomes not started. In isolation, not started could mean that the lathe could not be started that morning. However, within the Petri net graph it is obvious that not started is a state between being loaded and being started, so means that the part is ready to be machined and the lathe is waiting for the instruction to begin machining.

The synergy can breakdown where there is insufficient information in the structure and the place names. This happens when the structure is complicated or a place represents a difficult or unexpected idea. In these circumstances, the Petri net graphs are accompanied by pseudo-code and place name descriptions.

Referencing

Numbering places facilitates referencing in three ways: identification, history and replication. It is easier to locate a place if it is given a unique identification number, but it must be done logically. The approach

taken follows output-work-backwards described above. In this way, a history of development can be kept to help in error checking and in future maintenance, because places can be numbered in the order they are drawn.

It is possible to provide every place with a unique number. However, it has proved simpler to start referencing from 1 for each controller or module within a controller, and make reference to a place as if addressing a house, e.g. place 1 'load lathe' in lathe conditions of the cell controller. This is a more modular way, which enables easier modification. A very large system might require numbering of controllers in the same way as numbering places.

Where places appear more than once, i.e. replicated, then they must be cross-referenced. Places can be replicated within a module of a controller or between two or more modules of a controller. For example, during the development of lathe conditions in the cell controller, Figure 5-7 shows places 7 and 14 'lathe work handling idle' are mutually cross referenced, and this is indicated by '>'. In further development of the cell controller, Figure 5-8 shows in the lathe conditions appears renumbered as place 7 and is cross referenced with place 3 in index conditions. The referencing is denoted by 'IW', and correspondingly place 3 in index conditions is cross referenced 'L' for lathe conditions.

Petri net place reference codes appear with entry and exit places as source and destination codes, and with local Petri net places as place replication cross references. Entry and exit places are manifest as occam channels, and are not replicated. Local Petri net places become occam variables, but, unlike local places, variables are not replicated for comprehensibility.

Naming and Referencing

Communication between controllers is referenced and not joined by arcs, in order to reduce clutter. This communication is implemented as channels in step 4. Exit and entry places are given destination and source codes respectively. The codes are abbreviations for the controllers with which communication is intended, e.g. 'LC' is the alias of lathe controller. A combination of the codes is used in naming channels, see section 5.9.1, and is done for readability and to help in consistency checking.

Places can be drawn with different radii to accommodate voluminous place names, but place names should be abbreviations of understood ideas. It is preferable for the user to comprehend the idea, by learning the meaning of the accompanying description, rather than to read a Petri net graph with lengthy places names. Petri net place name descriptions supplement the names where there is doubt that the place is a sufficient or obvious representation. For example, in the pallet at lathe statuses of Figure 6-8, it is obvious when a full pallet arrives at the lathe and the lathe is loaded with a component, then the full pallet becomes empty. Unexpectedly however, as the pallet becomes empty, it also becomes lathe associated, and ceases to be not lathe associated. Here the respective accompanying descriptions would be 'the part leaves the pallet but only that part will return to that pallet due to the association between the pallet and the lathe' and 'complement to lathe associated'. Users are informed that there is an accompanying description by a '+d' in the place (not done).

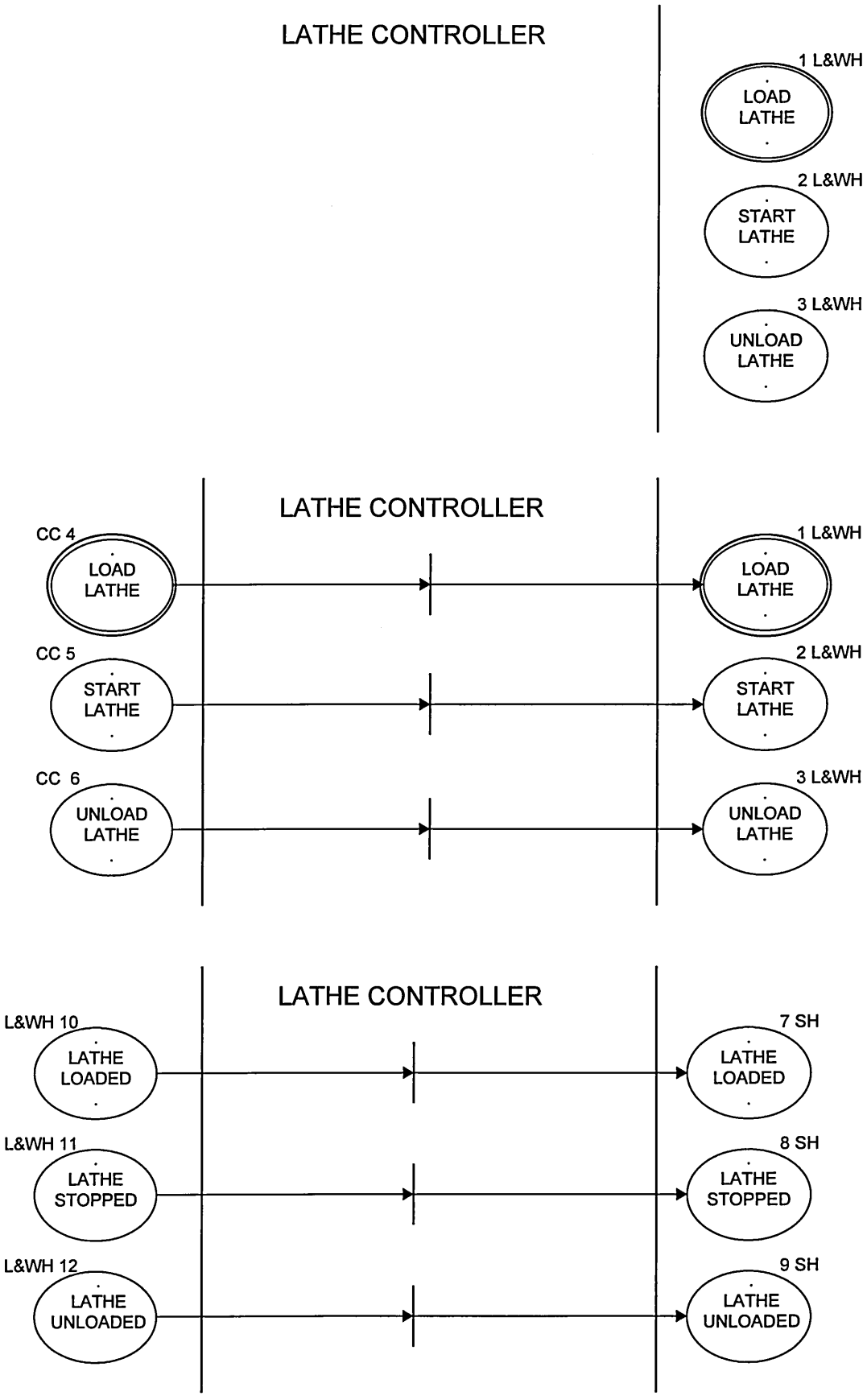


Figure 6-1 Lathe controller development of step 2: a) rules 1-3 and b) rules 4-6 outputting to level 1 controllers; c) rules 1-6 outputting to the status handler

The next sections pay particular attention to the decisions made for the different controllers. According to the output-work-backward strategy, the first to be considered are the workstation controllers, then the cell controller, then status handler, and the cycle completes with the workstation controllers again.

6.3.5 Workstation Controllers

The workstation controllers are similar, refer to Figure 2-4, and the lathe controller is chosen as the example, refer to Figure 6-1.

Task 1-3

The outputs from the lathe controller are sequential, from the manufacturing requirements, so are drawn vertically downward, the first being to load the lathe. The output 'load lathe' is drawn as an exit place to the right of the output boundary. It is given reference 'L&WH' to indicate to which Petri net graph it will communicate, a level 1 Petri net graph in this case, and is numbered 1. The next output in the sequence is to start the lathe, and is drawn as exit place 'start lathe'. It too is referenced 'L&WH', as is the final output in the sequence 'unload lathe', but are numbered 2 and 3 respectively, as shown in Figure 6-1a.

Tasks 4-6

The condition to satisfy exit place 'load lathe' is simply the instruction from the cell controller to load the lathe, which is named entry place 'load lathe'. The entry place is drawn to the left of the input boundary, referenced 'CC' indicating communication with the cell controller, and numbered 4. Exit places 'start lathe' and 'unload lathe' also simply relay instructions from the cell controller, so are treated in the same way as 'load lathe' but are numbered 5 and 6, refer to Figure 6-1b.

Task 7-9

In this instant no action is needed for either task 7, 8 or 9, because: there are no places local to the lathe controller; no exit place consolidation is possible; and the tokens indicating the initial marking are introduced from the cell controller.

The cycle outlined above indicates that some of the inputs to the status handler are outputs from the workstation controllers. Several entry places of the status handler are exit places of the lathe controller, and are 'lathe loaded', 'lathe stopped' and 'lathe unloaded'. These, like the other entry places, merely relay information from the input of the controller to its output, and are treated in a similar way, except the destination reference code is 'SH' to the status handler, and the source reference code is 'L&WH' from the lathe and lathe work handling, refer to Figure 6-1c.

The lathe controller is re-arranged to follow the strategy of sequences being read vertically downward, see Figure 6-2. Similar results are obtained for the miller, robot and conveyor controllers, refer to Figure 6-3, Figure 6-4 and Figure 6-5.

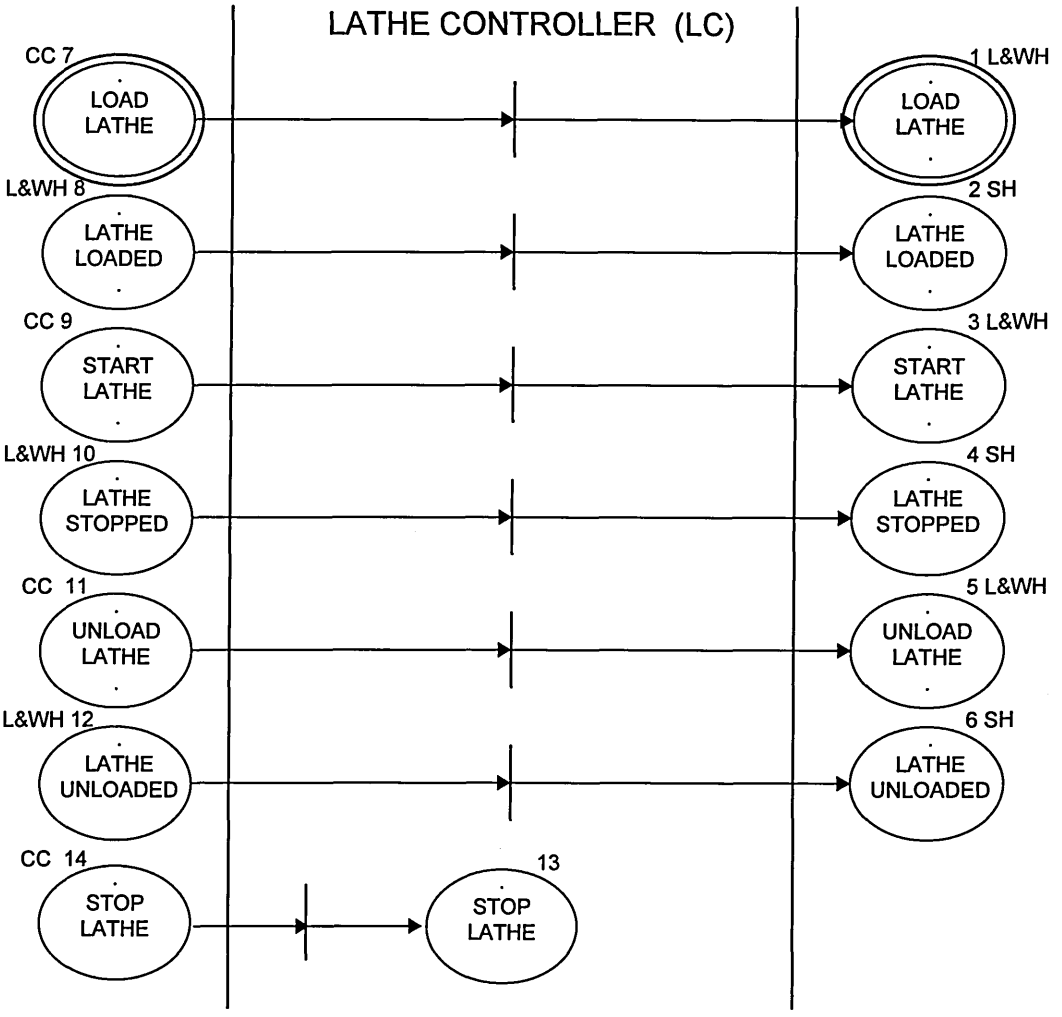


Figure 6-2 Combined lathe controller showing sequence (read vertically down), rearranged reference numbers and termination

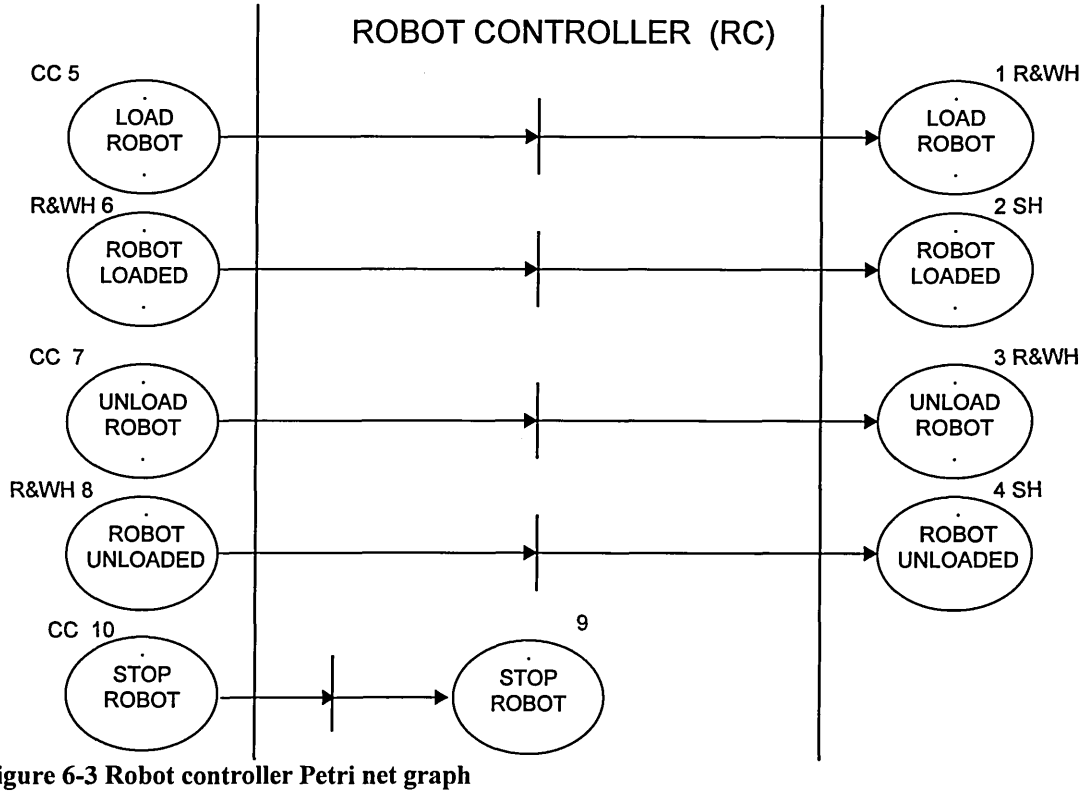


Figure 6-3 Robot controller Petri net graph

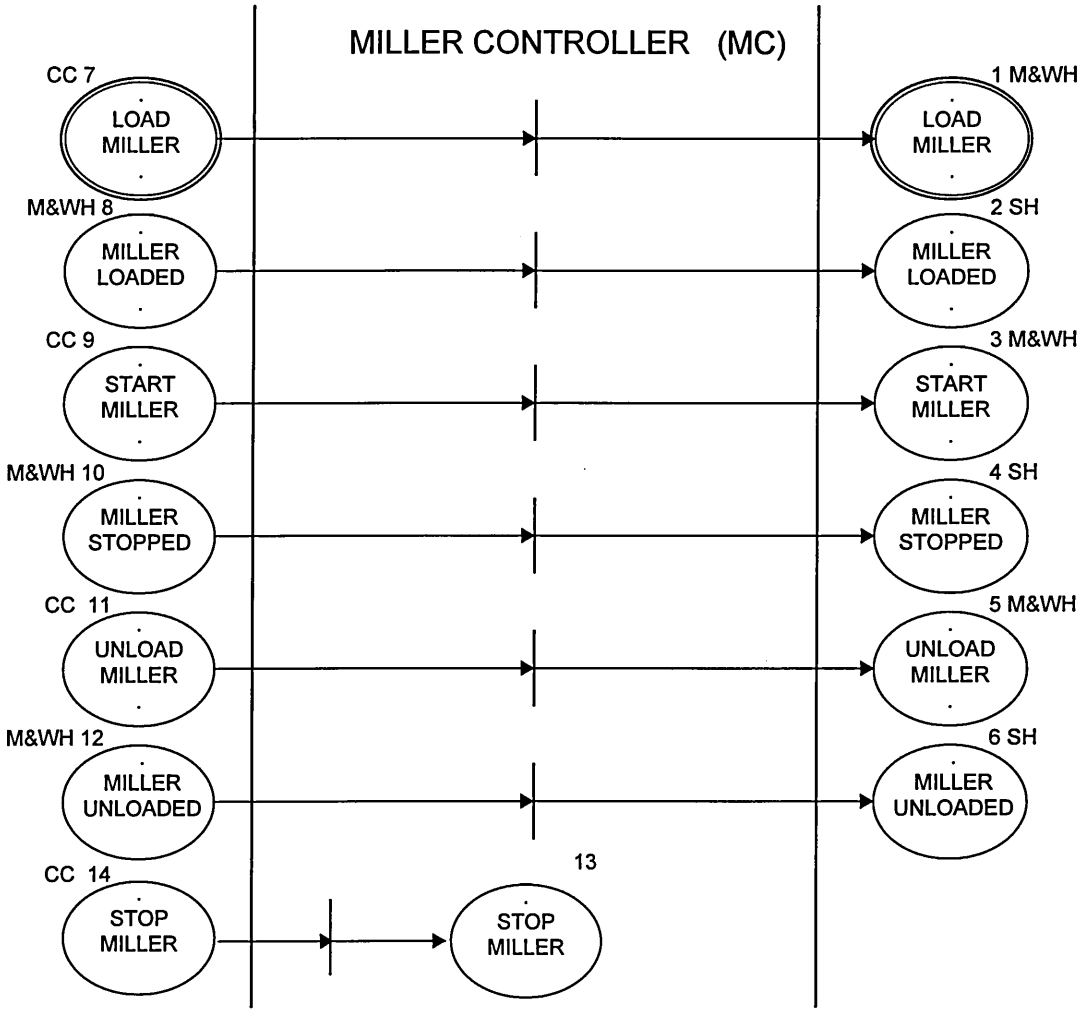


Figure 6-4 Miller controller Petri net graph

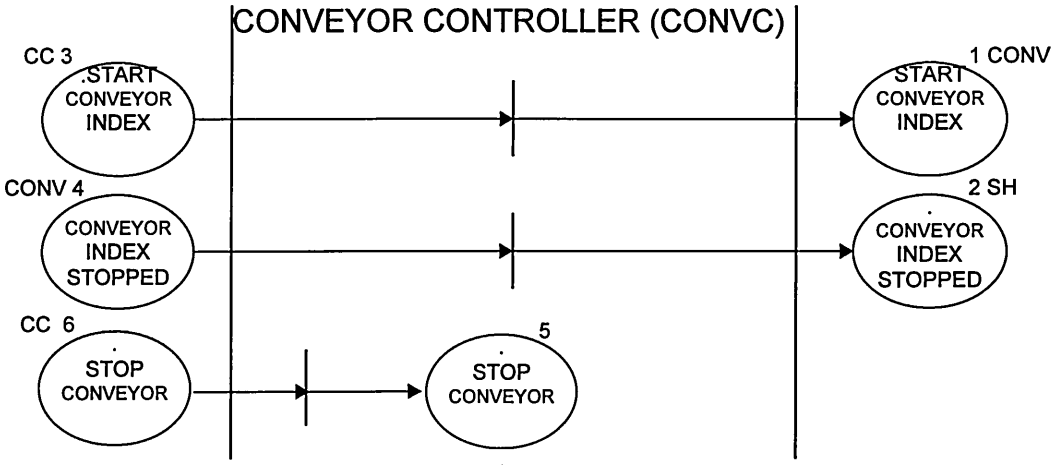


Figure 6-5 Conveyor controller Petri net graph

6.3.6 Cell Controller

The cell controller determines what and when things happen in the FMC. It regularly requests an update from the status handler, which responds by sending all of the statuses of the FMC. The decisions are made from this fresh update of statuses.

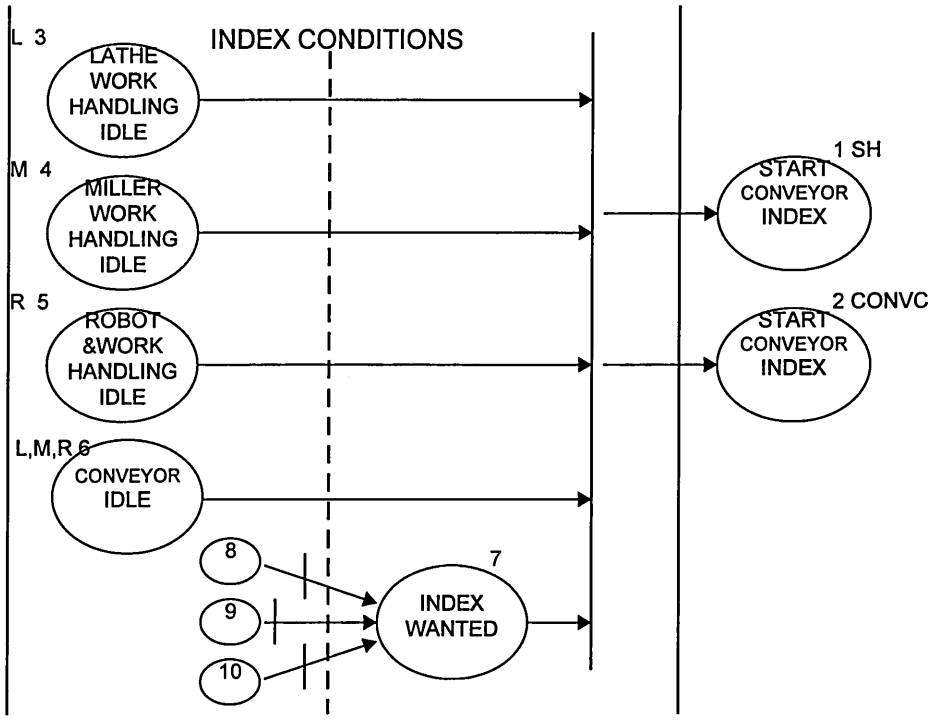
The statuses held in the status handler are combined and sent to the cell controller via the consolidated place update. The update from the status handler is the only input to the cell controller, and, as can be seen from the overall Petri net graph, it can happen on more than one occasion.

How often and for what reason to update is a very important consideration, because it is the managed update, in an otherwise uni-directional flow, and closed loop, of information, which prevents deadlock. The consideration is a balance between the need for fresh statuses, communication workload, comprehensibility and maintenance.

The major reason to update is the need for fresh statuses. It is necessary to update after an index, because the pallet information is altered after an index due to the new positions of the pallets. When to index is only one decision made by the cell controller. Others are when to load, unload and start the lathe, load, unload and start the miller and when to load and unload at the robot refer to Figure 5-8 to Figure 5-11. conveyor idle is replicated in the lathe, robot, miller and indexing conditions, lathe work handling idle is replicated in lathe and indexing conditions and robot & work handling idle is replicated in robot and indexing conditions, so a fresh update is needed between each of the conditions in the cell controller.

For the most part, the development of the Petri net graph for the cell controller is straight forward. On examination of the Petri net graphs relating to loading, unloading and starting the lathe and the miller, and loading and unloading at the robot, it is seen that all input places to transitions are entry places and require no local places. The only concern is whether to duplicate input places common to more than one transition, such as entry places: 'lathe idle', 'lathe work handling idle', 'miller idle', 'miller work handling idle', 'robot & work handling idle'. Loading and unloading the lathe are mutually exclusive. Not duplicating these entry places emphasises the mutual exclusion between the transitions to which they input, so there is a trade-off between replicating for clarity, and not replicating for showing mutual exclusion. Mutual exclusion is a manifestation of deadlock, refer to section 4.10.1, so should be made obvious. All entry places in the cell controller are made local and communication is consolidated into exit place update.

Due to its size, the Petri net graph regarding indexing is shown in Figure 6-6 and Figure 6-7. 'start conveyor index' depends on whether all of the transferring is complete and indexing is wanted. The strategy adopted, when indexing is wanted, is a 'pull' rather than a 'push' approach. Thus indexing is wanted when the pallet is wanted at the lathe, at the miller or at the robot. These, in turn, depend on whether the lathe, for example, is idle or has just finished turning and the pallet is not at the lathe (i.e. the pallet is at the miller or robot). Figure 6-6 shows 'start conveyor index', where places 7,8,9 and 10 appear in Figure 6-7.



Places 7 to 10 from Figure 6-7

Figure 6-6 Condition index allowed in the cell controller

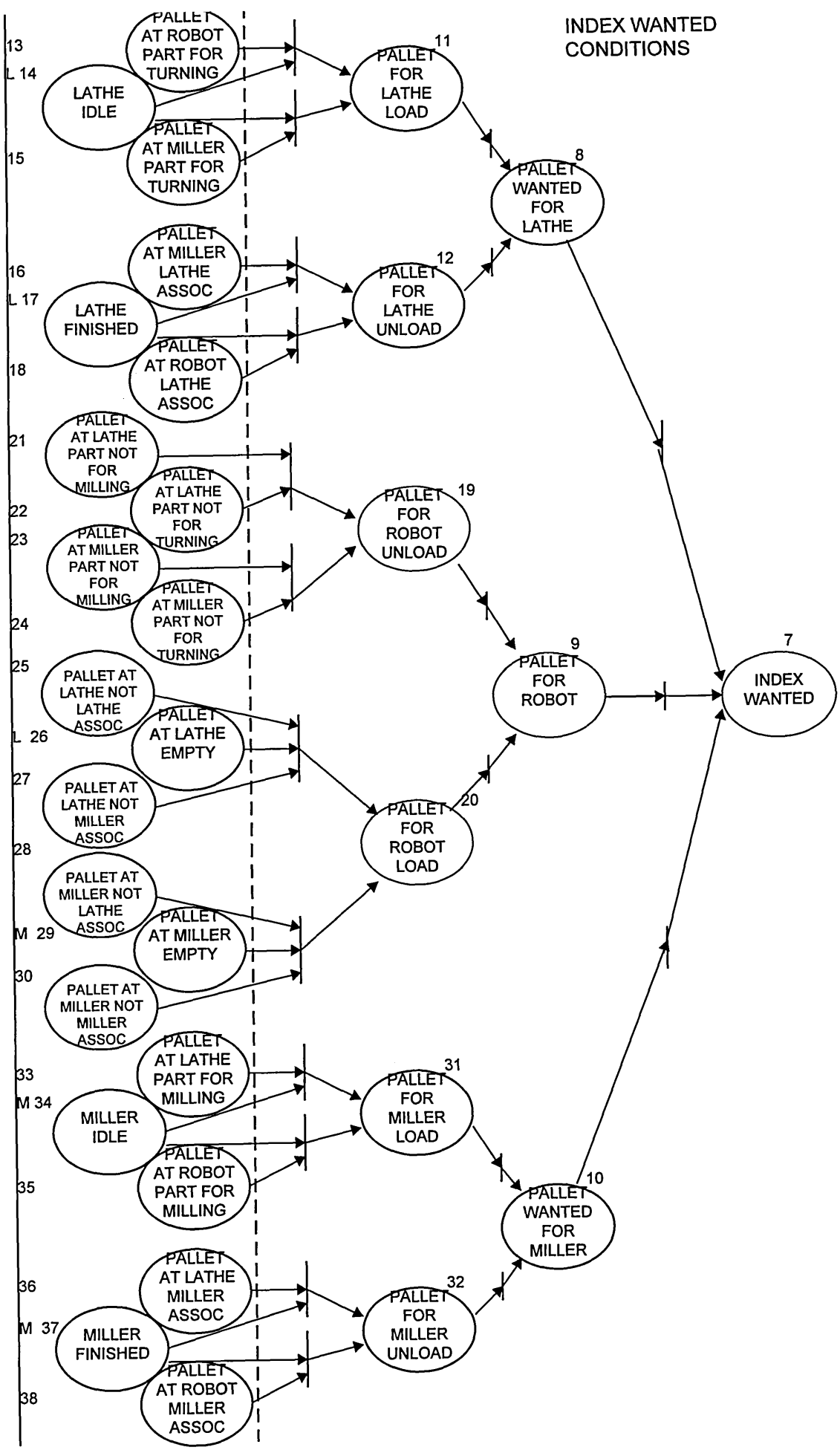


Figure 6-7 Condition index wanted in the cell controller

To aid comprehensibility and future maintenance, all the places local to the cell controller are preserved, even though all but place 'index wanted' are redundant. That is places 7 to 11 and 24 and 25 are not needed, because the transitions input by the entry places can all output to the place 'index wanted'.

When an update of fresh statuses is received, then all of the tokens in the cell controller are removed and relevant markings in the exit places made local are mirrored in the entry places made local of the cell controller. The only initial marking in the cell controller is in the first request to update.

6.3.7 Status Handler

At first sight, none of the Petri net graphs in the status handler follow the tasks in step 2 of the methodology. Places have arcs leading into them pointing left-to-right, but they also have arcs leaving them pointing right-to-left, refer to Figure 6-8 to Figure 6-11. Some places are drawn with no arcs at all, as in Figure 6-13 and Figure 6-14. The reasons, detailed below, are to improve comprehensibility by minimising replication of places while highlighting which statuses are output from the controller, and by organising statuses in convenient and appropriate modules.

Minimising the replication of places in the 'lathe statuses' Petri net graph can be seen in detail in Figure 6-12 a), b) and c). During development, by applying the tasks of step 2 to place 4 'lathe turning', the transition and its output arc are drawn leading to the exit place, in the usual way. The input places needed to satisfy the condition of the transition are determined, and are 'start lathe' from the cell controller and 'lathe not started' which is local (at this instant) to the status handler and should be drawn within the boundaries. Similarly, exit place 5 'lathe finished' is output place to a transition whose input places are 'lathe stopped' and 'lathe turning' which is the exit place just considered. These and all other exit places in the status handler are later consolidated into exit place update. This Petri net graph does preserve its sequential properties when read downwards.

Exit places in the status handler are consolidated into update, and are made local. Exit place update becomes an occam channel with a sequential protocol, and the exit places made local become occam output variables. The arcs pointing right-to-left remain within the controller, thus preludes communication deadlock.

Pallet statuses proved the most difficult to represent in an appropriate and understandable way. Considerations include: the FMC should be expandable, it has three pallets and three workstations, but could have more in future. It must be capable of representing any number of components which are conveyed between workstations by the pallets, and components could require turning only, milling only or turning and milling.

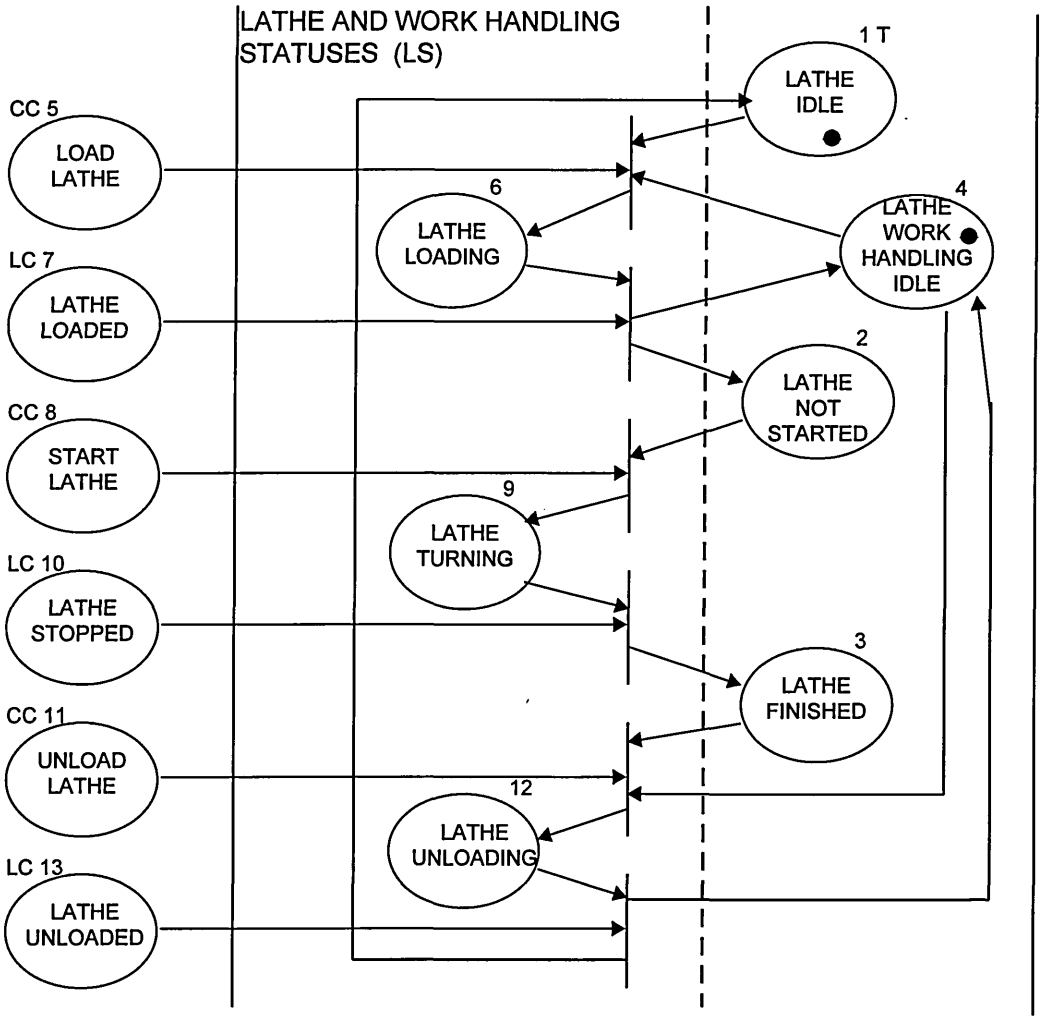


Figure 6-8 Lathe statuses in the status handler

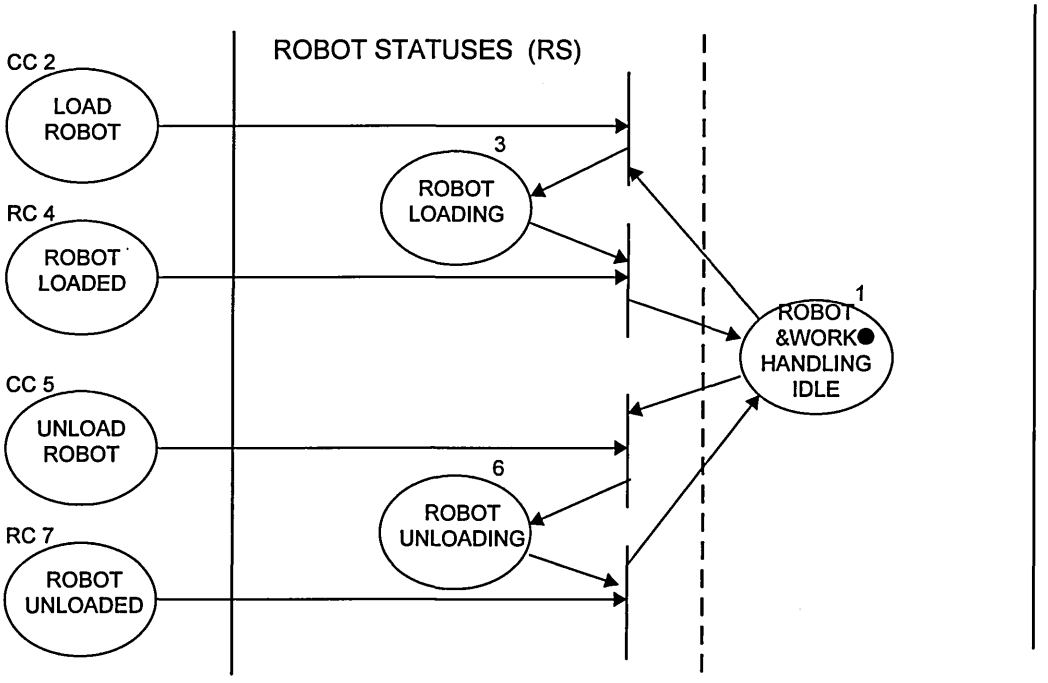


Figure 6-9 Robot statuses in the status handler

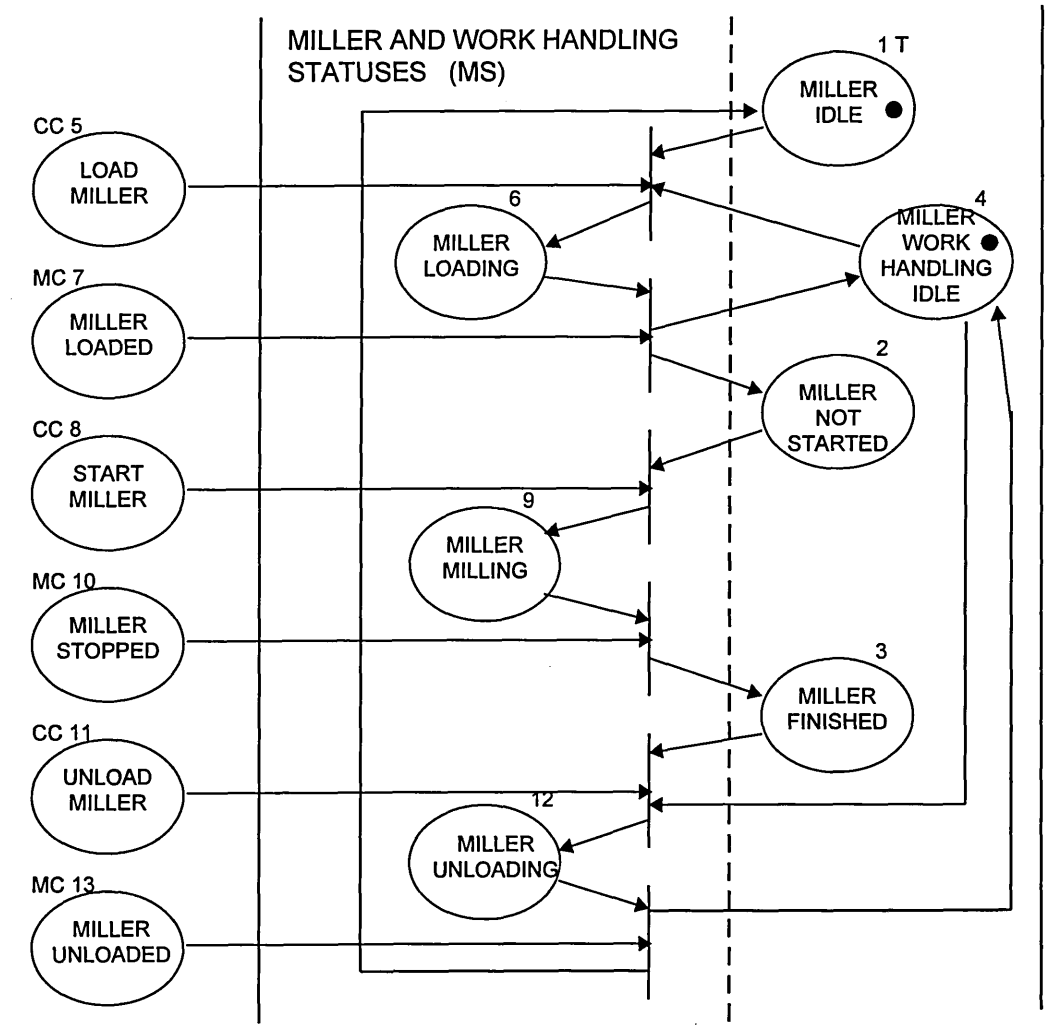


Figure 6-10 Miller statuses in the status handler

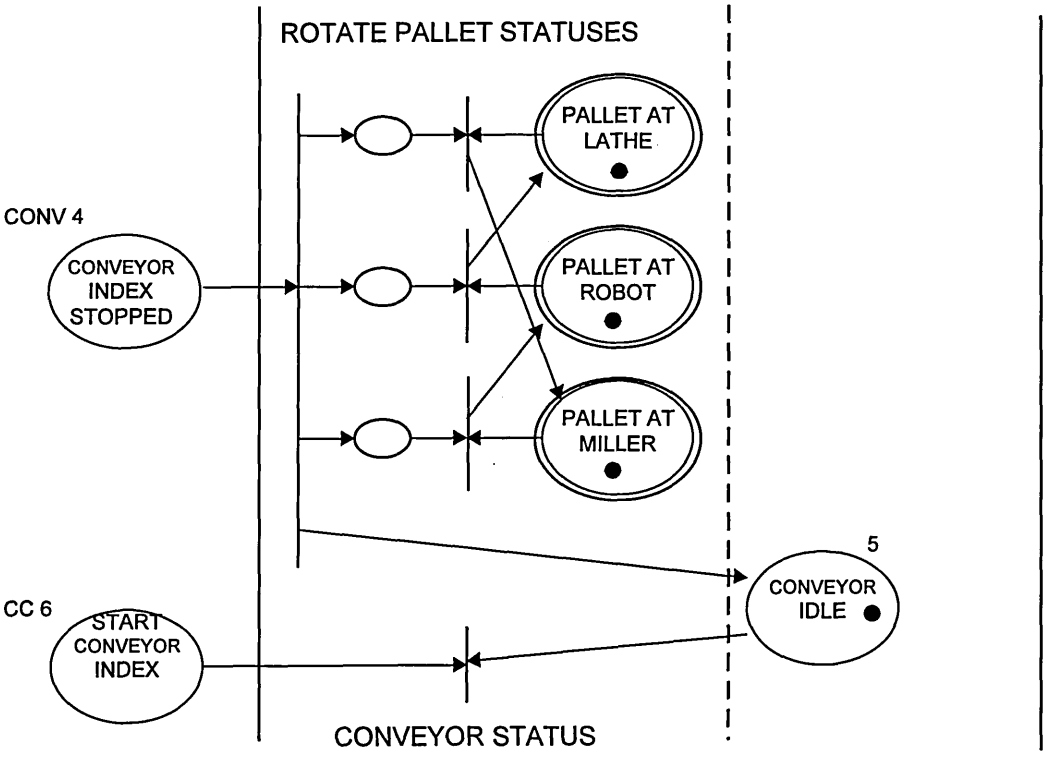


Figure 6-11 Rotate pallets statuses and conveyor status

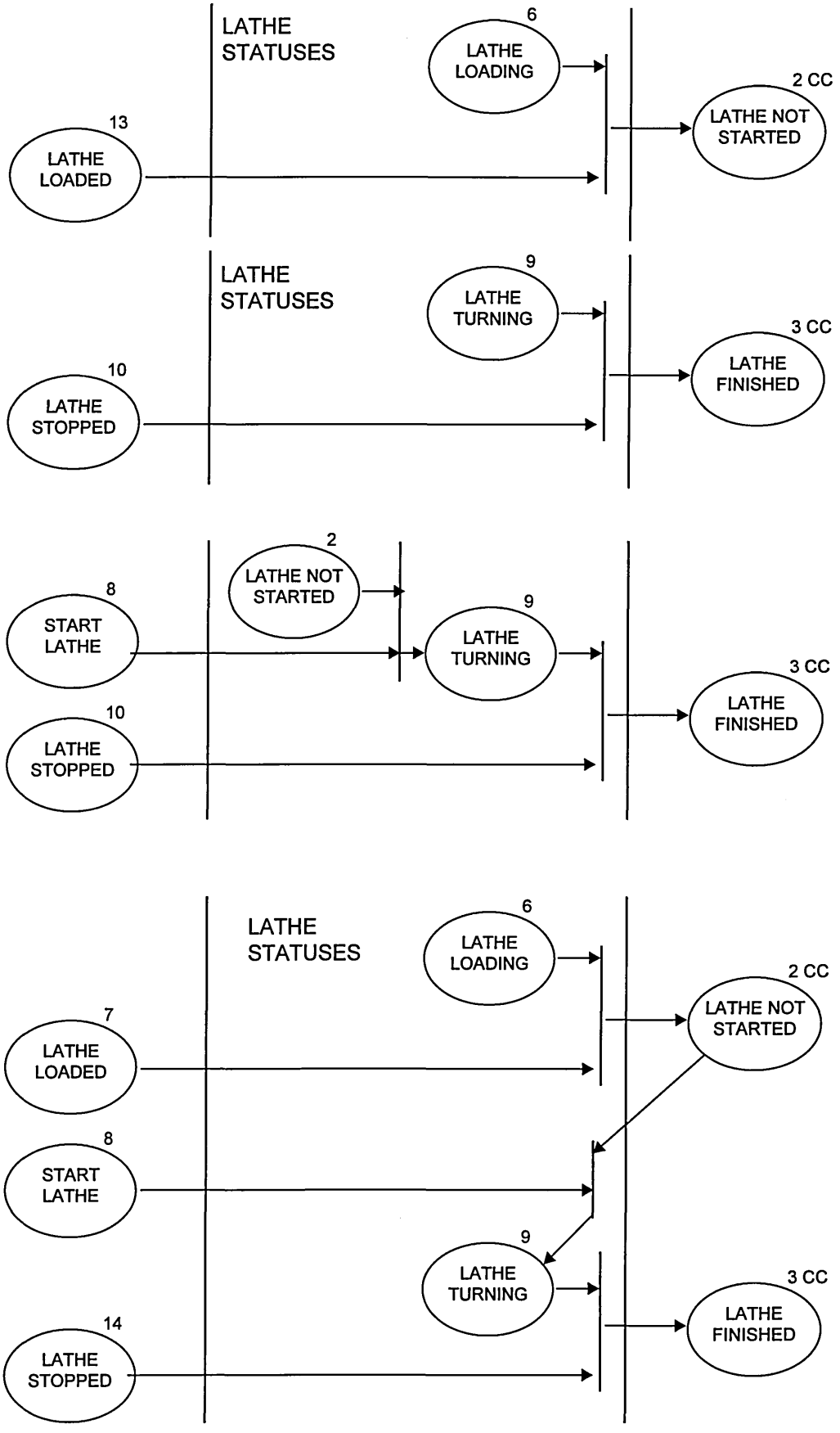


Figure 6-12 Reduction of excessive place replication in the development of Figure 6-8

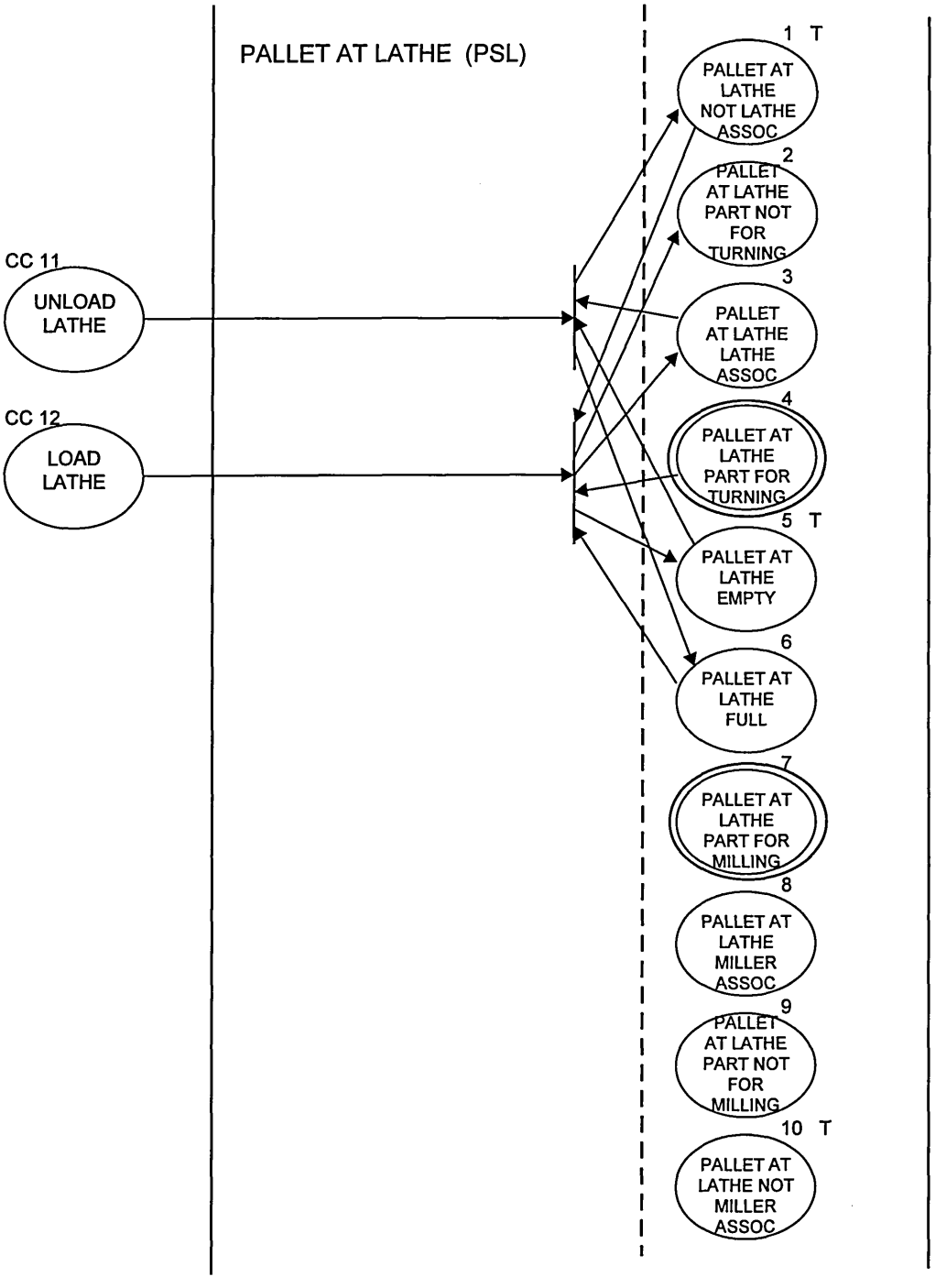


Figure 6-13 Pallet at lathe statuses in status handler

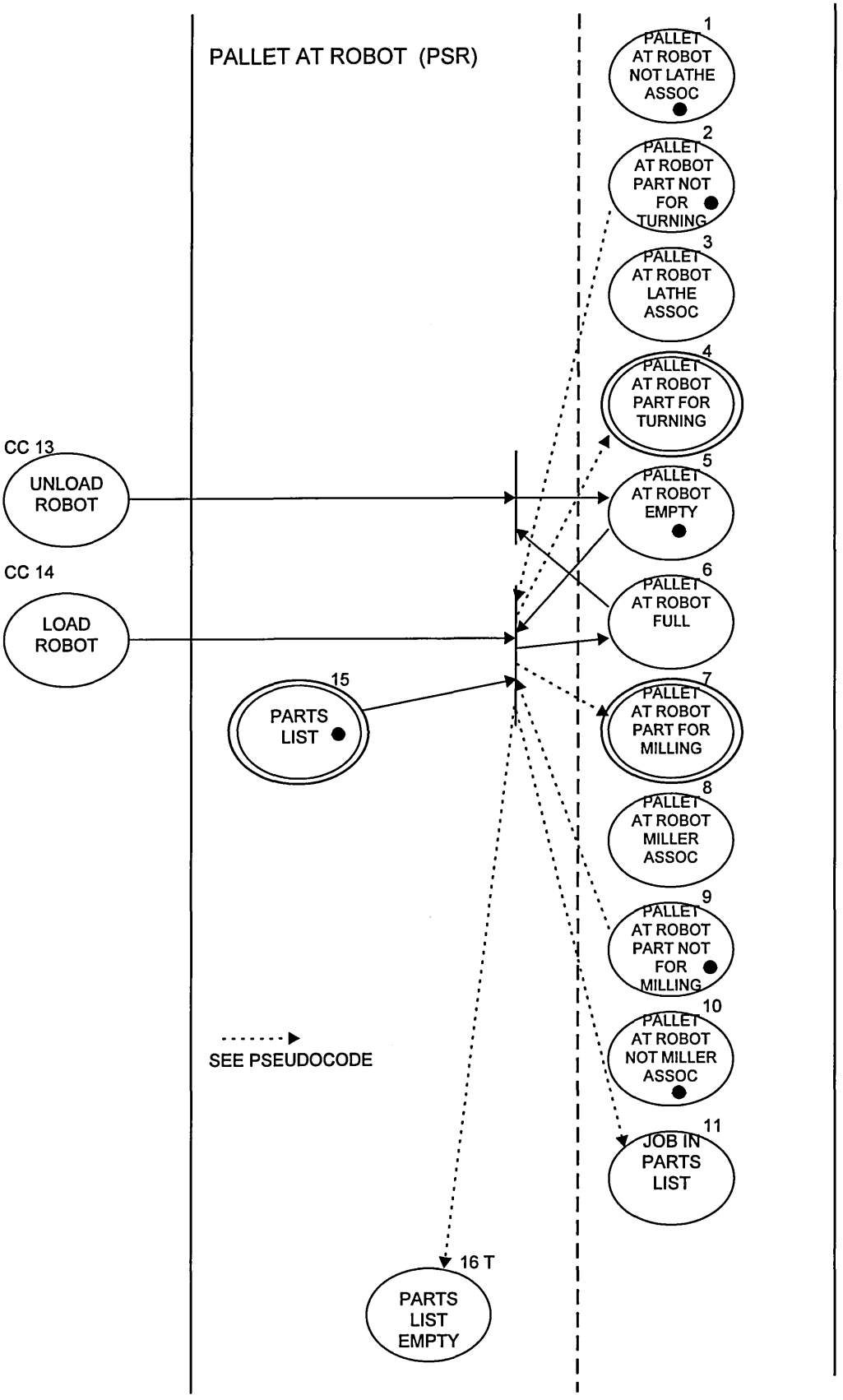


Figure 6-14 Pallet at robot statuses in status handler

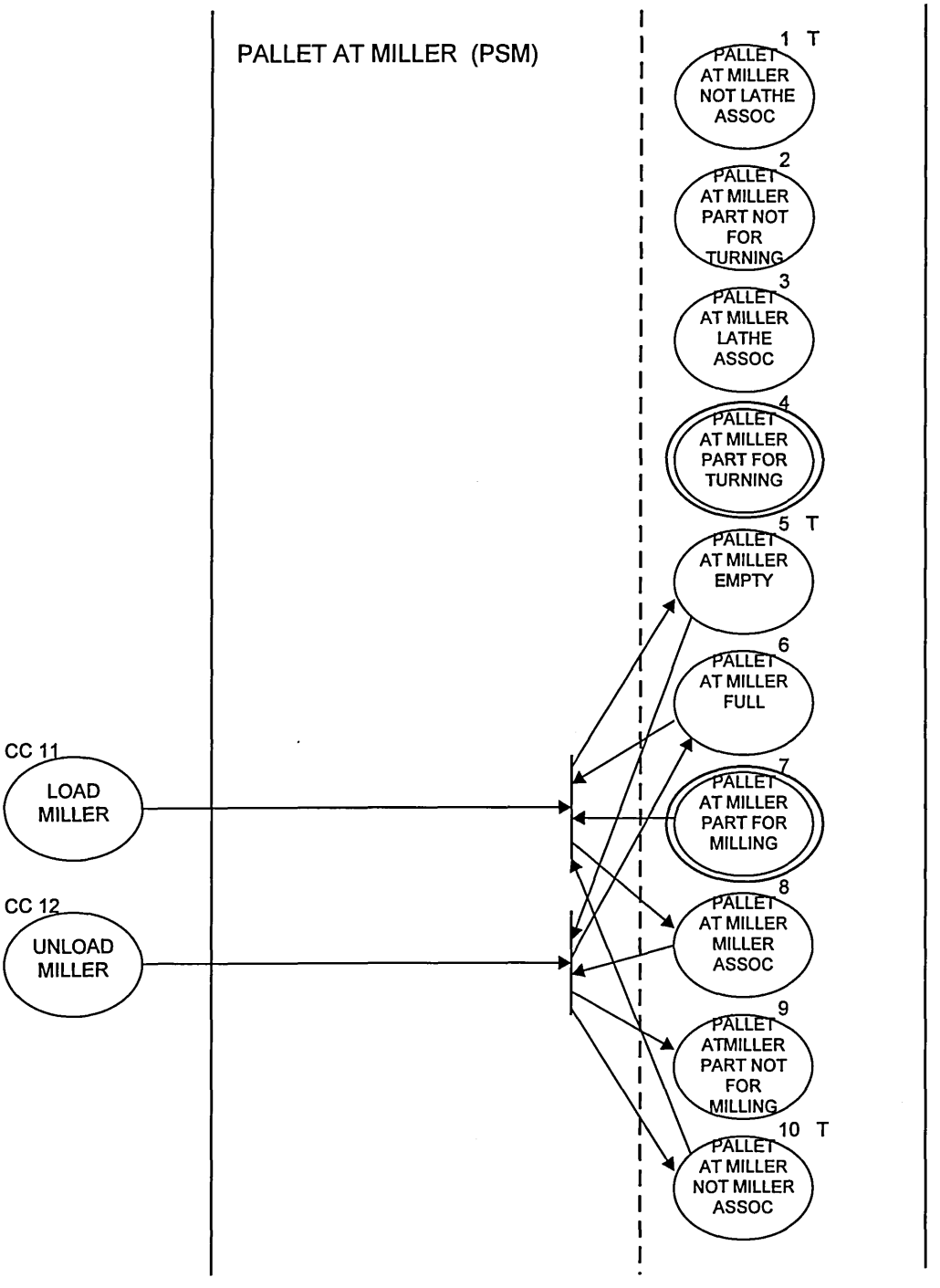


Figure 6-15 Pallet at miller statuses in status handler

6.4 Step 3 - Synthesis of the Controller Graphs

6.4.1 Graph Positioning

Step 3 specifies the interaction of the Petri net graphs of step 2, and models the entire FMC. The output-work-backward cycle described in section 6.3.1 is represented horizontally in the overall Petri net graph, and consequently is in three columns. To the right, the outputs of the status handler are readily available for the cell controller update, and the inputs of the status handler are continually receiving feedback from all of the other controllers. In the middle column, the workstation controllers output feedback to the status handler as a result of processing the instructions input from the cell controller. To the left, the cell controller outputs instructions to the workstation controllers, determined from the manufacturing conditions and the state of the cell, which is input from the status handler. The modules representing the 'pallet at lathe statuses' and 'pallet at miller statuses' of the status handler are drawn below the workstation controller Petri net graphs to save space.

6.4.2 Inter-Controller Communications

The deadlock prevention and Petri net/occam equivalence characteristics of the methodology preclude the need to consider implementation and the restriction imposed by finite software and physical hardware limitations. Communication between controllers will be implemented as channels if controllers run on the same transputer, or as links if controllers run on different transputers. The small amount of data communicated between controllers and the work done within controllers is minute. The reason why more than one transputer is used is to distribute the processing close to the workstations, to make sufficient links available to communicate with the workstation PLCs, and to simulate the case where communication and processing workloads become significant.

6.5 Step 4 - Conversion to Occam

6.5.1 Naming Conventions

The naming convention of section 6.3.4 is applied as follows:

- procedure, file (`.occ`), header and include file (`.inc`) names are taken from the names of the Petri net graphs of the controllers
- variables are taken from the names of places
- channels and links take their names from the combination of the destination and source codes referenced by exit and entry places of communication controllers. For example, in the instruction to load the lathe, the destination code of exit place 'load lathe' in the cell controller is 'LC' and the source code of entry place 'load lathe' in the lathe controller is 'CC', so the channel name becomes `'cc2lc'`

latheHd.inc

```
PROTOCOL CC2LC
  CASE
    load.lathe.cc2lc;   BOOL; INT::[]BYTE --for part program
    start.lathe.cc2lc;  BOOL
    unload.lathe.cc2lc; BOOL
    stop.cc2lc;         BOOL
:
PROTOCOL LC2SH
  CASE
    lathe.loaded.lc2sh;   BOOL
    lathe.stopped.lc2sh;  BOOL
    lathe.unloaded.lc2sh; BOOL
:
VAL Lathe.machining.time IS 229:
VAL Lathe.transfer.time IS 15:
```

latheCon.occ

```
#INCLUDE "latheHd.inc"

PROC lathe.controller(CHAN OF CC2LC cc2lc,
                      CHAN OF LC2SH lc2sh)
  #USE "delay"
  BOOL Load.lathe.cc2lc, Start.lathe.cc2lc, Unload.lathe.cc2lc:
  BOOL Lathe.loaded.lc2sh, Lathe.stopped.lc2sh, Lathe.unloaded.lc2sh:
  BOOL Loop, Stop.cc2lc:
  [7]BYTE Code.lathe:
  INT Code.size.lathe:
  SEQ
    Loop:=TRUE
    WHILE Loop
      ALT
        cc2lc ? CASE
          load.lathe.cc2lc; Load.lathe.cc2lc; Code.size.lathe :: Code.lathe
            IF
              Load.lathe.cc2lc
                SEQ
                  --find part program relating to Code.lathe
                  delay(Lathe.transfer.time)
                  lc2sh ! lathe.loaded.lc2sh; Lathe.loaded.lc2sh
          start.lathe.cc2lc; Start.lathe.cc2lc
            IF
              Start.lathe.cc2lc
                SEQ
                  delay(Lathe.machining.time)
                  lc2sh ! lathe.stopped.lc2sh; Lathe.stopped.lc2sh
          unload.lathe.cc2lc;  Unload.lathe.cc2lc
            IF
              Unload.lathe.cc2lc
                SEQ
                  delay(Lathe.transfer.time)
                  lc2sh ! lathe.unloaded.lc2sh; Lathe.unloaded.lc2sh
          stop.cc2lc; Stop.cc2lc
            IF
              Stop.cc2lc
                Loop:=FALSE
:
```

Figure 6-16 Occam code for the lathe controller

Many of the conventions of occam 2 [Inmos 1988] are adopted as follows:

- reserved words and channel protocols are in capital letters
- names are descriptive - words are concatenated by full stops

Conventions created in the methodology are:

- DOS file names can be mixed case, and describe the procedure names that they contain
- channel names and tags are in lower case, and their combinations are unique
- variable names begin with a capital letter, other letters are lower case
- statuses, such as pallet, lathe and lathe transfer, begin with two capitals for brevity, e.g. `LS.full` means the lathe status is full, and `LT.idle` means the lathe transfer status is idle

6.5.2 Procedure Formats

The generic format of procedures for controllers is given in the template of Figure 5-2. The format contains the foundations of any workstation controller, which are:

- channels definitions- to be put in the header file
- procedure name and arguments list
- used files, e.g. timer delay
- local definitions
- a `SEQ`
- a `WHILE` loop
- an `ALT` with one tagged channel input guard, which guards a process with
- a condition testing the input variable
- one tagged channel output

6.5.3 Workstation Controllers

All of the workstation controllers follow the generic procedure format and naming conventions, and their occam code appears in the appendices. Timing delays are used in place of transfers and machining operations to simulate their effects. Compare the lathe controller Petri net graph of Figure 6-2 with the occam code of Figure 6-16 (other occam code is in the appendices). The differences are discussed in section 6.3.3.

6.5.4 Cell Controller

The cell controller is split into 5 sub-procedures, which contain the decisions and reflect the Petri net graphs. It also includes a procedure which hides the task of requesting and receiving the status update (`update.statuses()`).

At the end of the file `cellCon.occ`, after the declarations and definitions, a loop contains the sequence of updates and decisions. The sequence 'request statuses, receive statuses and make decisions' is an important one, because it is this that helps prevents deadlock.

In the cell controller, the number of decisions made between the feedbacks, and the reason to update the cell controller, can be seen from the cell controller in the overall Petri net graph and explained in section 6.3.6. The Petri net graph shows the cell controller conditions can be grouped in to those associated with lathe conditions, robot conditions, miller conditions, and conditions determining whether an index is to be started and if an index is allowed, and these are reflected in the occam code.

Examine the lathe conditions, where places 6 and 7 'conveyor idle' and 'lathe work handling idle' are input places to transitions leading to 'lathe load' and 'lathe unload' exit places. This is a condition for conflict and one of potential deadlock. However in the methodology, by reading vertically down the Petri net graph indicates that 'load lathe' precedes 'start lathe' precedes 'unload lathe'. In occam, the three conditions are part of the same IF construct, and use the same values for the variables `CS.idle` and `LT.idle` respectively, so the conflict in Petri nets does not arise in occam using variables (but it could if using channels). In an `IF` construct, only one guarded-process is executed, and it corresponds to the first guard to be `TRUE`. The guarded processes cause the status handler to modify some statuses, in the case of 'load lathe' `LT.idle`, `LS.idle`, `PS.full[Lathe]`, `PS.not.lathe.assoc[Lathe]` and `PS.part.for.turning[Lathe]` become `FALSE`. If an update occurs between each of the three lathe conditions, then the condition to 'load lathe' and consequential modification of `LT.idle` to `FALSE` in the status handler, would prevent the condition to 'unload lathe', because it would use `LT.idle`, now `FALSE`, to make the decision.

6.5.5 Status Handler

The production of occam code for the status handler complies surprisingly well with the tasks defining workstation development. One major difference is in the addition of communication with the screen and keyboard. The special protocol (`SP`) is defined in the `#INCLUDE`d file "`hostio.inc`" and is the first line of file "`statHand.occ`". Procedures provided with the toolset that communicate with the screen and keyboard (e.g. `so.write.string`) are pre-compiled for use and are held in library file "`hostio.lib`". The user-defined pre-compiled procedure "`delay`" is also specified by the `#USE` compiler directive. All of the channel definitions and source code of other procedures are referred to and `#INCLUDE`d in file "`statHand.inc`" to hide much of the information which is not reflected in the status handler Petri net graph.

The processes guarded by the case alternatives represent the consequences of transitions. Petri net input places (bounded to one token, so are represented by occam Boolean variables) are emptied of tokens (variables made `FALSE`), and output places are given tokens (variables made `TRUE`). For example, take the transitions enabled by entry places 'start lathe' and 'lathe stopped' of Petri net graph lathe statuses. When the cell controller decides to start the lathe, it informs the status handler then instructs the lathe controller. In turn, the lathe controller instructs the lathe to start, waits for the lathe to stop and informs the status handler of this.

In more detail, the entry place 8 'start lathe', in the status handler's Petri of Figure 6-8, enables the transition, which fires, removing a token from places 2 'lathe not started' and 8 'start lathe', and gives a token to place 9 'lathe turning'. In occam, entry and exit places are represented by the input and output of a channel, so are not modelled as variables, but local and exit places made local, such as places 2 'lathe not started' and 9 'lathe turning' respectively are. The transition, represented in occam, see file "statHand.occ", makes variable `LS.not.started FALSE` and `LS.turning TRUE`.

When the 'lathe stopped' entry place enables the transition to fire of Figure 6-8, then tokens are removed from places 'lathe turning' and 'lathe stopped' and gives a token to place 'lathe finished'. This is implemented in occam by making `LS.turning FALSE` and `LS.finished TRUE`.

When the cell controller needs to be updated with the current state of the cell, it requests the status handler by outputting via channel '`cc2sh`'. The status handler responds, using procedure `update.statuses()` which contains the input of channel '`cc2sh`', see file "`statHand.occ`", by transmitting the machine statuses then by sending the set of pallet statuses for each of the three pallets and two other variables.

When the cell controller decides to index, it informs the status handler and instructs the conveyor to index. The conveyor controller instructs the conveyor PLC to allow the pallets to index, receives a signal when indexing is finished, then informs the status handler of this. Once the entry place 'index finished' in the status handler's Petri net, Figure 6-11, enables the transition to fire, then pallet statuses are rotated. Here, the ten statuses relating to the 'pallet at robot' (places 1 to 10 in Figure 6-14) are moved to become the statuses relating to the 'pallet at lathe', that of 'pallet at lathe' (places 1 to 10 in Figure 6-13)are moved to that of 'pallet at miller' and finally the statuses relating to the 'pallet at miller' (places 1 to 10 in Figure 6-15)are moved to become the statuses relating to the 'pallet at robot'. This is better understood conceptually, rather than producing a less readable Petri net with many arcs and transitions, and is why the places have double concentric circles to indicate other than a simple place. Occam is required to, and can simply, model indexing and the exchange of statuses relating to indexing pallets. The pallet statuses are represented by ten single dimensional arrays, each with three elements (one per pallet). The rotation in the status handler's Petri net graph is best represented in occam by exchanging the pallet arrays `PS.not.lathe.assoc[]`, `PS.lathe.assoc[]`, `PS.not.for.turning[]`, `PS.for.turning[]`, `PS.full[]`, `PS.empty[]`, `PS.not.for.milling[]`, `PS.not.for.milling[]`, `PS.miller.assoc[]` and `PS.not.miller.assoc[]`. Occam's multiple assignment allows the rotation to be mimicked more closely than normal assignment involving temporary variables, and is written in procedure `rotate.pallet.statuses()`.

6.5.6 Master Procedure

The top or master procedure, refer to "`top.occ`" in the appendices, represents the overall Petri net graph, so does not follow the generic procedure format. It is the procedure that is the first to be executed, and is written and compiled specially for either single transputer or transputer network operation.

In single transputer operation, the master procedure defines how communication between the controllers is implemented as channels between procedures. The channel definitions are held in the many header files which are written for the controller procedures. The communication between controllers is seen by examining the channel names in the arguments lists of the communicating controller procedures. The naming convention is followed throughout.

6.5.7 Termination

Termination is where the FMC is allowed to finish its operations and comes to rest. It is invoked under two conditions: the job list is complete; and an operator intervention. The user input to stop the simulation is via the keyboard, and, as with all communication with the screen and keyboard, it is part of the status handler. The information regarding the job list is also held in the status handler. It is for these reasons that the status 'terminate' is changed in the status handler. The decision when, and in which order, to stop the controllers is made by the cell controller, and depends on the status of Terminate, which is sent via update.

Termination due to completion of the job list is shown in module 'job list termination' in the status handler as Figure 6-18. It shows that the conditions necessary are an empty and unassociated pallet is at the lathe or miller while the parts list is empty. The occam procedure `job.list.termination()` is in `statHand.inc`, and is called just after the robot has unloaded, refer to tag `robot.unloaded.rc2sh`. The procedure tests the above condition and sets `Terminate` to `TRUE`.

The cell controller makes the decision to terminate or not every update. The Petri net graph is shown in Figure 6-18 and appears at the bottom of the cell controller in the overall Petri net graph. It shows more output places from the transition than input places, which is unusual. This still follows Strategy 1 output work backwards, because 'stop' entry places of all controllers, mapped to exit places of the cell controller, are satisfied by the same condition, which is if place 7 terminate holds. The order of termination is important. The workstation controllers must be stopped before the cell controller and the status handler controllers. The Petri net graph showing the conditions to terminate in the cell controller is given in Figure 6-17. The occam procedure `terminate()` in `cellCon.occ` reflects this.

Termination by operator is not shown in the status handler, nor are any communication with the screen and keyboard. The occam procedure `user.termination()` is held in `statHand.inc`, and is called within the `WHILE` loop but before the `ALT`. The procedure reads the input, if any, and a tested 'q' changes the value of `Terminate` to `TRUE`, and the occam code for which is given in the appendices.

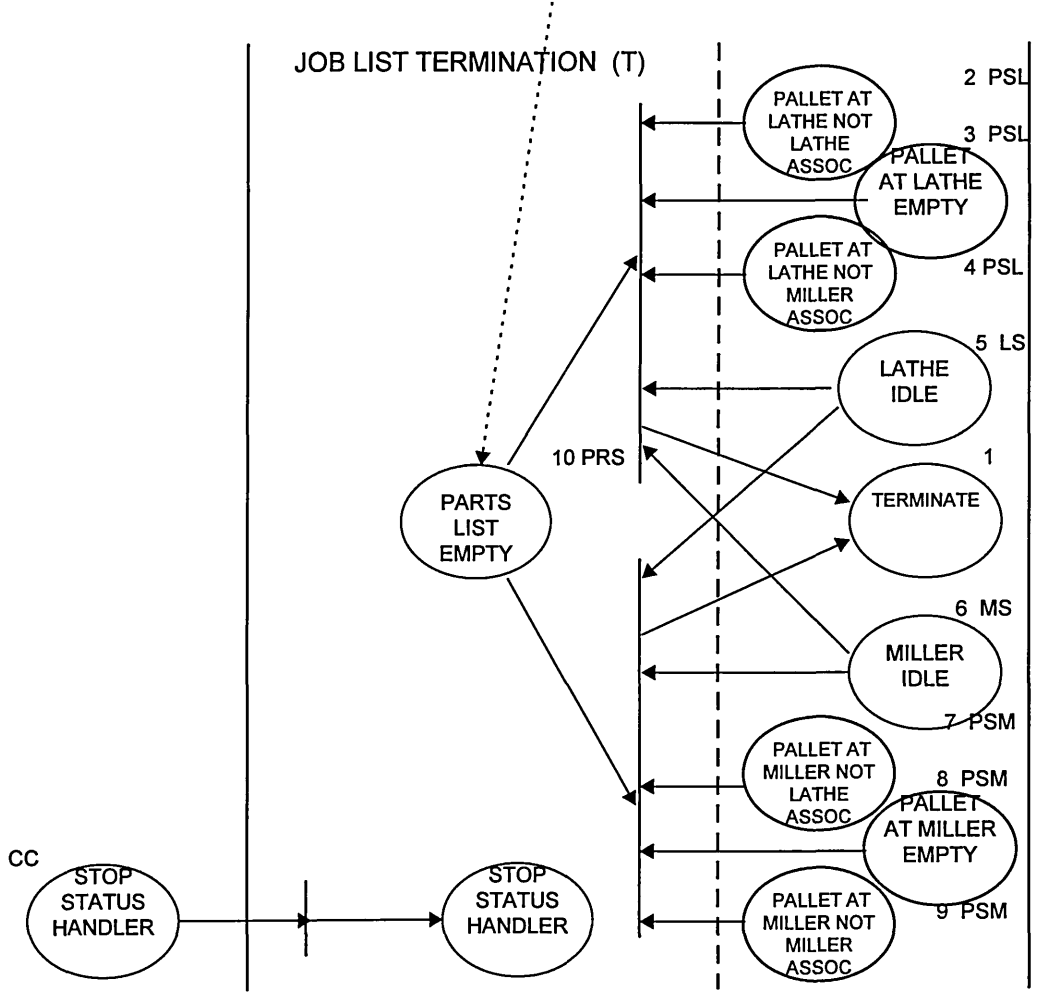


Figure 6-17 Termination due to end of parts list

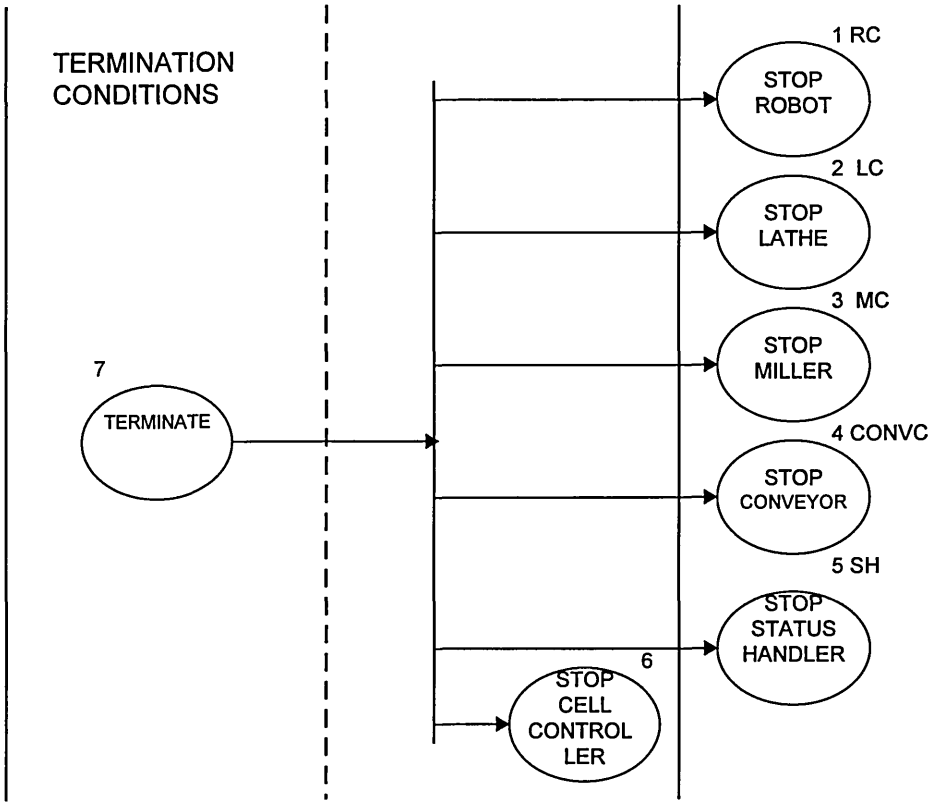
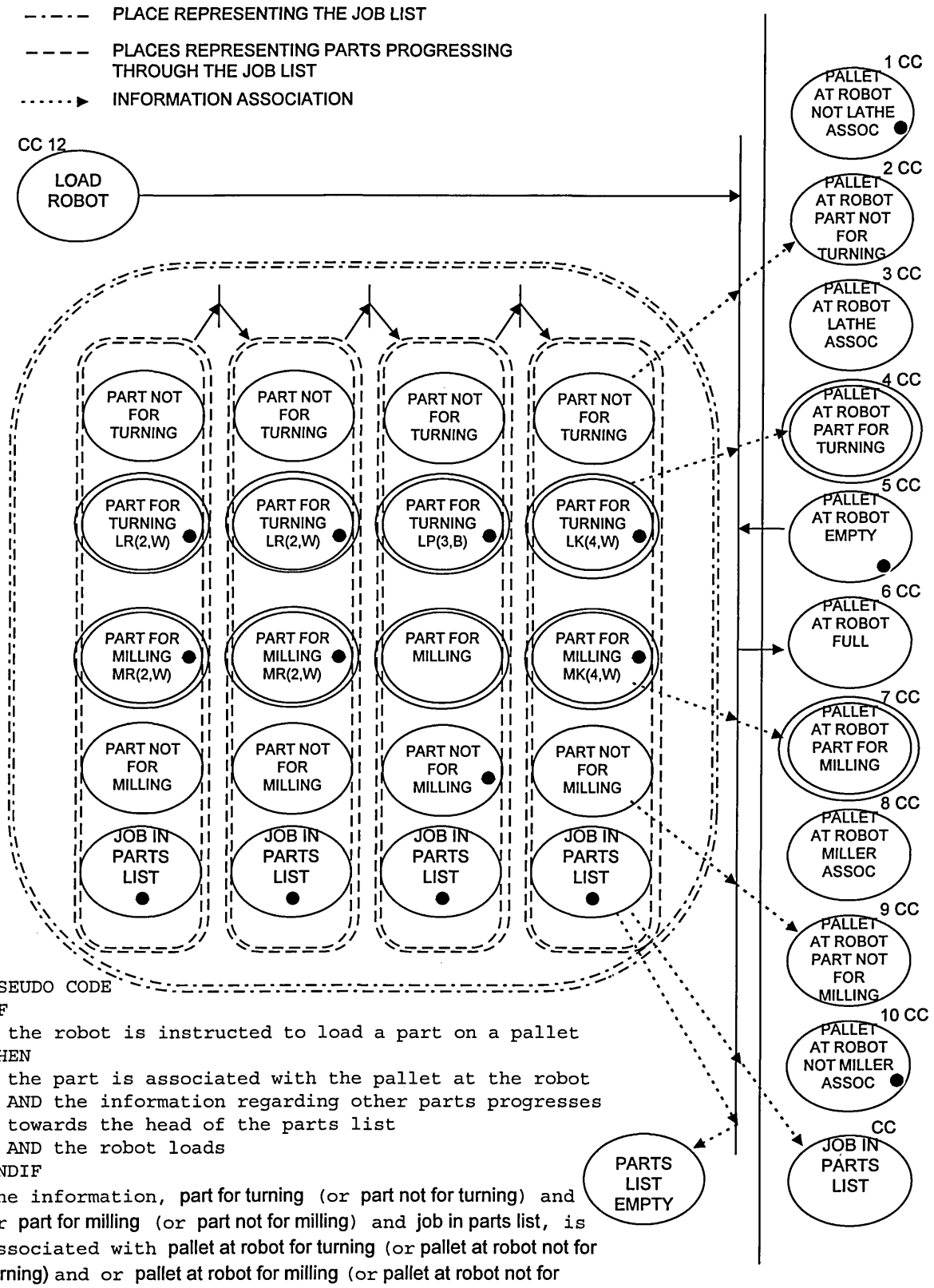


Figure 6-18 Termination conditions in the cell controller




```
PSEUDO CODE
IF
  the robot is instructed to load a part on a pallet
THEN
  the part is associated with the pallet at the robot
  AND the information regarding other parts progresses
  towards the head of the parts list
  AND the robot loads
ENDIF
```

The information, part for turning (or part not for turning) and or part for milling (or part not for milling) and job in parts list, is associated with pallet at robot for turning (or pallet at robot not for turning) and or pallet at robot for milling (or pallet at robot not for milling) and job in parts list

LK(4,W) and MR(2,W) are part program codes from the parts list of Table 2-4

Figure 6-19 Petri net and pseudo code abstraction of Figure 6-14

```
Loop:=TRUE
    WHILE Loop
      SEQ
        user.termination(Terminate)
        ALT
          cc2sh.robot ? CASE  ---------------------------- R O B O T
            load.robot.cc2sh; Load.robot.cc2sh
              SEQ
                Robot.idle                :=FALSE
                Pallet.at.robot.empty              :=FALSE
                Pallet.at.robot.full               :=TRUE
                Pallet.at.robot.part.not.for.turning := 2nd item from Job.code
                Pallet.at.robot.part.for.turning     := 3rd item from Job.code
                Code.lathe                       := next 7 items from Job.code
                Pallet.at.robot.part.not.for.milling:= 11th item from Job.code
                Pallet.at.robot.part.for.milling    := 12th item from Job.code
                Code.miller                      := next 7 items from Job.code
                Job.in.parts.list                    := 1st item from Job.code
                Job.code := Job.code + 1
                IF
                  NOT Job.in.parts.list
                    SEQ
                      Parts.list.empty := TRUE
            unload.robot.cc2sh; Unload.robot.cc2sh
              SEQ
                Robot.idle                :=FALSE
                Pallet.at.Robot.full               :=FALSE
                Pallet.at.Robot.empty              :=TRUE
          rc2sh ?  CASE
            robot.loaded.rc2sh; Robot.loaded.rc2sh
              SEQ
                Robot.idle              :=TRUE
            robot.unloaded.rc2sh; Robot.unloaded.rc2sh
              SEQ
                Robot.idle              :=TRUE
```

Figure 6-20 Pseudo code for Figure 6-19

```
PROC get.job.info(VAL INT Job.number)  -- from statHand.inc
  SEQ
    PS.part.not.for.turning[Robot]:=bool.from.char(Job.code[Job.number][1])
    PS.part.for.turning    [Robot]:=bool.from.char(Job.code[Job.number][2])
    Code.lathe             [Robot]:=[Job.code[Job.number] FROM 3 FOR 7]
    PS.part.not.for.milling[Robot]:=bool.from.char(Job.code[Job.number][10])
    PS.part.for.milling    [Robot]:=bool.from.char(Job.code[Job.number][11])
    Code.miller            [Robot]:=[Job.code[Job.number] FROM 12 FOR 7]
    Job.in.parts.list             :=bool.from.char(Job.code[Job.number][0])
:
PROC get.next.job.or.stop()
  SEQ
    get.job.info(Job.number)
    print.integer(Job.number)
    so.write.string(fs,ts,Code.lathe[Robot])
    so.write.string(fs,ts,Code.miller[Robot])
    Job.number := Job.number + 1
    IF
      (Job.number = No.of.jobs) --for a full parts list
        SEQ
          Parts.list.empty := TRUE
          Job.in.parts.list:= FALSE
          so.write.string(fs,ts," End of job list ")
      TRUE
        IF                      --if parts list is not full
          (NOT bool.from.char(Job.code[Job.number][0]))
            SEQ
              Parts.list.empty := TRUE
              Job.in.parts.list:= FALSE
              so.write.string(fs,ts," End of job list ")
          TRUE
            SKIP
:

  Loop:=TRUE -- from statHand.occ
  WHILE Loop
    SEQ
      ALT
        cc2sh.robot ? CASE  --------------------------- R O B O T
          load.robot.cc2sh; Load.robot.cc2sh
            SEQ
              so.write.string(fs,ts,"r  lod  ")
              RS.idle                :=FALSE
              PS.empty[Robot ]              :=FALSE
              PS.full[Robot ]               :=TRUE
              get.next.job.or.stop()
          unload.robot.cc2sh; Unload.robot.cc2sh
            SEQ
              so.write.string(fs,ts,"r ulod  ")
              RS.idle                :=FALSE
              PS.full[Robot ]                :=FALSE
              PS.empty[Robot ]               :=TRUE
        rc2sh ?  CASE
          robot.loaded.rc2sh; Robot.loaded.rc2sh
            SEQ
              so.write.string(fs,ts,"r  lodd ")
              RS.idle               :=TRUE
          robot.unloaded.rc2sh; Robot.unloaded.rc2sh
            SEQ
              so.write.string(fs,ts,"r ULODD ")
              RS.idle                :=TRUE
```

Figure 6-21 Occam code for Figure 6-19

6.5.8 Pseudo Code

Pseudo code is used where Petri nets cannot cope with the required logic (e.g. an inclusive OR) or to explain a complex place. This should not be confused with accompanying place descriptions, which aims to elucidate any place.

The most complicated example of the use of pseudo code is the representation of the parts list and the output arcs from the transition input by 'load robot', in the pallet at robot statuses of the status handler. The first representation is given in Figure 6-14.

Figure 6-19 shows a more detailed abstraction. The big outer place, with double circles and chained lines, represents the job list of place 15 in Figure 6-14. The tall inner places, with double circles and dashed lines, represent the jobs, which consist of attributes represented as places 'part for turning', 'part not for turning', 'part for milling', 'part not for milling' and the flag indicating that a job remains in the parts list. The attributes are items of information that are 'transferred' to the 'pallet at robot statuses', on loading at the robot, and then to the 'pallet at lathe statuses' and 'pallet at miller statuses' on indexing. The complex places with solid double circles ('part for turning' and 'part for milling') include part codes, which are discussed in sections 5.7.3 and 6.7.4.

The reasons why this violates Petri net rules are that transitions can fire when input places are not enabled, and that output places need not become enabled on firing. The arcs are dashed to indicate an 'information association' and a transfer of attributes on firing.

The places with double dashed lines representing jobs are 'connected' together by arcs and transition. This also represents a transfer of information, and happens when the transitions fire. The transitions, including that input by 'load robot', must fire at the same time, which causes the job attributes to be moved up the job list, and the attributes of job at the top are transferred to the 'pallet at robot statuses' as just described.

This description in English is verbose, and is only structured by grammar and the story line. A more methodical approach should be used, i.e. pseudo code, and is discussed in section 4.6. The first pseudo code abstraction of Figure 6-19 is given in the box at the bottom of the figure. A second, and more detailed pseudo code abstraction, is given in Figure 6-20. A third and final step is the occam code. The bottom of Figure 6-21 shows the `load.robot.cc2sh` tag of file `statHand.occ`. The procedure `get.next.job.or.stop()`, called within the process guarded by the tag, is defined in the top of the figure and is taken from file `statHand.inc`. It is possible to trace the transfer of information of part for turning to pallet at robot part for turning of Figure 6-19 through `Pallet.at.robot.part.for.turning := 3rd item from Job.code` of Figure 6-20 to `PS.part.for.turning [Robot] := bool.from.char(Job.code[Job.number][2])` of 21 as the transition fires (`bool.from.char` converts a charater to a Boolean).

This area requires greater consideration, and is left for further work.

6.5.9 Configuration for Transputers and Links

The methodology does not assist in configuring the occam code for use on a transputer network, but the overall Petri net graph does help in partitioning controllers to transputers. The recommendation of adopting tagged channels minimises the number of channels to be made into links. Converting the master procedure, of section 6.5.6, from single transputer to network execution is left for further work.

6.6 Application to Other DCSs

Section 3.5 compares four shop-floor DCSs: transputer/occam, FIP, MMS/MiniMAP and 9Tiles. The problem domain is the same for each, and is applicable to the methodology. However the methodology's solution domain can only be fully satisfied by the transputer/occam based DCS.

MMS/MIniMAP was chosen as a combination, because MMS is the only international standard for shop-floor control and MIniMAP is the most known application. MMS is an open standard that adopts the client-server model of communication. Section 4.10.4 discusses deadlock avoidance in occam using the client-server model. Protocol building in occam, and the serial communication of transputers with acknowledge, refer to section 3.5.5.7, are other necessities which make the methodology applicable to MMS.

FIP employs the producer-consumer-distributor communication model, which enables broadcasting. The methodology makes no provision for broadcasting messages, so is not applicable to FIP.

9Tiles is a buffer insertion ring network, and buffering is trivial to occam and transputers. A DCS heavily depends on its topology, and the transputer, with only four links, exacerbates the problem. However, topological considerations, such as ring applicability, are left to further work.

If transputers had many links, or occam could run on another microprocessor (possibly the T9000, see further work), then the methodology would apply to any CSP based language, e.g. Ada, on a point-to-point message passing DCS. Section 4.8.3 highlights the major differences between Ada and occam.

6.7 Overall Discussion

This section assesses the methodology against its aims, goals and strategies defined in chapter 5. It begins by examining how the methodology meets its aims, how it fulfils the needs of methodologies, and ends by making comment on the design of the methodology.

6.7.1 The Methodology Meets Its Aims

A direct comparison is made of the methodology, defined in chapter 6, against its five aims, outlined in section 5.2. A more general assessment, follows, addressing the problem and solution domains, as indicated in section 4.2.

Creating DCS requirements from the manufacturing requirements is achieved. FMCs make concurrently operating machine tools and work handling equipment to co-operate, and this is accomplished via

Strategy 2 concurrent and sequential actions and set out in Petri net graphs via Strategy 3 structuralise and modularise in task 1 of step 1. Manufacturing operations such as load lathe and start lathe are issued by level 2 workstation controllers. Using Strategy 1 output work backwards as described in section 5.3.6, then the DCS is created from the needs of the outputs.

Goal 2 comprehensibility applied to Petri net graphs and manifest in Strategy 2 concurrent and sequential actions and Strategy 3 structuralise and modularise satisfies the second aim. It is much easier to understand the overall Petri net graph than the labyrinth Petri net graph.

Goal 1 Petri net/occam equivalence is achieved by restricting outputOR Petri net communication between controllers via Strategy 1 output work backwards. Following the simple tasks of step 4 enables code to be translated from the Petri net graphs. The methodology exploits the formalisms of Petri nets and occam, but their mathematical bases do not have to be understood in order to use the methodology. However, the methodology, the Petri net firing task and the occam language must be understood. This non-reliance on mathematical comprehension should be attractive to industry.

Strategy 4 deadlock avoidance is implemented in task 1 of step 1 by imposing a cell controller and status handler controller aided by Strategy 1 output work backwards. Strategy 1 output work backwards, implemented in tasks 4 to 7 of step 2, achieves a uni-directional flow of information. The update between the cell controller and status handler is governed by the cell controller, and it is this controlled update that prevents deadlock by 'circular wait', described in section 4.10.1, while enabling closed loop control.

Occam code that compiles, executes what is wanted without deadlock and terminates, is correct, refer to section 3.2.2. The code produced, via use of the methodology does what is anted and terminates, thus fulfils the validity and verification requirements of section 3.2.2.1.

Problem and Solution Domains

The motivation for the methodology has perhaps imposed restrictions on its applicability, i.e. its problem domain, and on the target hardware and software, i.e. its solution domain.

The initial motivation was to transputerise an FMC, and the subsequent analysis highlighted the manufacturing requirements (refer to section 2.7) for safe, reliable and correct distributed control. This pre-determines much of the solution domain, by negating the selection of hardware, but leaving open the choice and development of software. The safety and distributed requirements bear heavily on the methodology, and lead to a restriction in the problem domain, which is for dependable distributed control.

The Problem Domain: Dependable Distributed Control

The safety requirement determines the use of formally based modelling, design and analysis of hardware and software components to ensure correctness. It also determines the use of reliable hardware, and possible hardware and software redundancy. The distributed requirement, originally done by an office LAN, requires correct and reliable message transmission between the nodes controlling the machine tools

and work handling. The control requirement, originally accomplished by running Pascal code on IBM compatible PCs, requires correct and reliable execution of manufacturing control software.

The Solution Domain

The solution domain is pre-empted by the imposition to use transputers. However, transputers and their programming language, occam, satisfy the distribution and control requirements by their mathematical bases, and by the transputer's on-chip communication, processing and memory and its high quality manufacture.

A significant design choice was to exploit this underlying formalism, and to introduce another mathematically base tool to assist in the design process. Petri nets were chosen for their capability of modelling, analysing and designing concurrent systems in matrix and graphical forms. The Petri net specification of the FMC was produced, refer to the 'labyrinth' Petri net graph, but it is unreadable.

This incomprehensibility indicates that it is not enough to adopt correct design and implementation tools (Petri net and occam) and use reliable hardware (transputers) to achieve dependability, but there is a need to understand the design and communicate it.

It is technically possible to analyse the Petri net specification of the FMC manually and by Petri net analysis tools. However, the absence of the latter and the unreadability restricting the former, rendered the Petri net graph almost useless.

Similarities between the structures of Petri nets and occam were noticed, and an equivalence was sought. Petri nets allow the inputAND, outputAND, inputOR and outputOR of Figure 3-13, while occam does not permit the non-deterministic outputOR. An equivalence is achieved by preventing the designer from building non-deterministic outputORs. This leads to Goal 1 Petri net/occam equivalence.

Petri net analysis can show that the design contains faults, but does not show how to correct them. Another significant design choice was to avoid faults, rather than to eliminate or tolerate them, to achieve deadlock freedom and hence correctness. Of the four manifestations of deadlock, avoiding, or controlling, 'circular wait' was chosen. This lead to the predominant uni-directional flow of information, with a controlled update, to create a single and deterministic closed loop cycle. This lead to Goal 3 pro-activity and Strategy 4 deadlock avoidance.

6.7.2 The Needs of Methodologies

In section 4.12.2 the conclusions indicate the needs of a methodology for safety related software engineering for parallel systems to be safeness, liveness, reliability, portability and scalability, and are discussed below in relation to the research methodology.

Safeness and Liveness

Section 3.2.2 discusses correctness in terms of safeness, liveness and termination. Safety properties must always be true i.e. partial correctness, mutual exclusion and no deadlock, and liveness properties must eventually be true, i.e. total correctness and termination, no livelock and no starvation. The methodology satisfies the safety properties of deadlock (section 4.10) and mutual exclusion (section 4.9). Liveness

properties are largely satisfied by occam, but unfairness is introduced by the toolset (section 3.4) and termination is successful.

Reliability

Section 3.2.3 mentions faults, fault avoidance, fault elimination and fault tolerance. Fault avoidance is a significant feature in the methodology, and its pro-active approach, described in section 5.3.1. Fault elimination, by testing, is mainly left for further work, but is discussed in section 5.3.14. Similarly, fault tolerance is discussed in section 3.2.3, but left entirely to further work.

Portability and Scalability

Section 4.2 asserts that methodologies and CASE tools should enable the production of portable software. The research methodology has a specific problem domain, i.e. DCSs, and a narrow solution domain, i.e. occam and transputers.

Methodologies and CASE tools trade-off between generality and comprehensibility. A general purpose methodology or CASE tool with a broad problem and solution domain must sacrifice the understanding of the methodology or CASE tool and therefore that of the solution. Producing a methodology or CASE tool with sufficient tasks and guidance for the many problem and solution domains would overload the user in documentation, tools and techniques. All but the best general purpose methodology or CASE tool will be a burden to use.

Portability in the solution domain of the research methodology is paradoxical. The motivation for adopting transputers was their efficiency in communication, so choice of other platforms would be a retrograde step.

Software portability, i.e. occam 2 portability, was not a goal of Inmos, so occam is solely a transputer based language. Of all of the other languages developed for the transputer, occam seems the most succinct. Other languages, such as concurrent c, have a greater variety of sequential instructions. The research methodology is not concerned with sequential instructions, so it should be applicable to many transputer based languages. Other parallel languages with alternation and point-to-point communication, i.e. the occam ALT, ? and !, e.g. Ada with its SELECT and rendez-vous, should benefit. However, the problem domains for which Ada was developed demand a more sophisticated methodology.

Scaleability is described in section 4.8.4, and relates to optimising the number of processors. The need for a DCS in an FMC is for geographical location and logical structure, refer to section 4.3, so scalability is not significant in the methodology.

6.7.3 The Design of the Methodology

In section 4.12.3 the conclusions demonstrate that Petri nets and occam are a useful combination in the development of safety related parallel software, and substantiate this with examples. However, many of the examples do not make full advantage of the visualisation inherent in Petri net graphs, or do not provide sufficient guidance especially at the earliest stages in the software life cycle. The methodology

aims to overcome these deficiencies by integrating strategies around core goals, which are called for in section 4.12.4, as follows:

- It is possible to preclude the formal refinement process by imposing an equivalence on the system specification and implementation. There are similarities between the formal structures of Petri nets and occam (Goal 1). Although there are significant irreversibilities (e.g. no occam output ALT, and no general Petri net timing and priorities), much of the inherent structure of occam can represent general Petri nets, thus reducing the number of, otherwise necessary, data structures. By restricting Petri nets to prevent the 'outputOR' communication between controllers a Petri net/occam equivalent is made, refer to Table 6-1, as is achieved by working backwards from the controller outputs (Strategy 1)
- Petri net graphs are ideal for representing the interactions of concurrent processes, but when drawn are generally incomprehensible (Goal 2). It is possible to improve readability of Petri net graphs, if
    - concurrent and sequential actions are separated (Strategy 2)
    - presented in a modular/structural fashion, with sequences of output are shown read downwards(Strategy 3)
    - arcs crossing is minimised, thus requiring place replication with cross-referencing
- Generally, when designing systems using Petri nets: models are produced, then analysed, then redesigned and re-analysed until correct. This time consuming and frustrating repetition is reduced by aiming to get it right first time (Goal 3). This can be accomplished by:
    - restricting communication to one direction with a single controlled feedback (Strategy 4), thus preventing deadlock rather than detecting and recovering from deadlock. The cell and status handler controllers are needed to issue instructions to and gather feedback from the workstations. The cell controller must also determine when to receive the state of the cell
    - limiting the number of tokens per place, preferably to one
    - starting with outputs from controllers and working backwards (Strategy 1)

Comprehensibility, Correctness and Flexibility

In safety critical or safety related DCSs, it is essential that the control software be correct. Various routes have been presented, refer to section 4.4.1:

- the most effective (theoretically), least understood and consequently least popular is the use of formal methods and formal refinement
- the least effective, most popular, but not necessarily the most comprehensible are ad hoc and informal methods
- there are many rigorous methods, that use formal or semi formal methods. Generally, they are suitable in safety related systems, often easy to use and understand, and are becoming more popular. Rigorous methodologies and CASE tools are examined below

A key issue in rigorous methodologies and CASE tools is the balance between correctness, flexibility and comprehensibility.

One argument for CASE tools is that humans cannot cope with the complexities of designing and analysing safety related and DCSs (section 4.7.6.1). Thus their use is to achieve correctness via automation, and this is the major difference between a CASE tool and the methodology on which it was based.

Comprehensibility is an important contribution to correctness. An easily understood methodology or CASE tool will lead to a more easily understood and consequently a more correct design.

A flexible methodology or CASE tool can have benefits and disadvantages. Comprehensibility can be enhanced where flexibility enables the use of better or varied representation. However, variety requires more to be known and presents more to go wrong.

Deadlock in DCSs can be used as an example to illustrate further the balance between correctness, flexibility and comprehensibility. Faults, such as deadlock, can be avoided, eliminated or tolerated, refer to section 3.2.3. Deadlock is manifest in four ways (section 4.10). An inflexible methodology or CASE tool will not permit deadlock, by preventing the user from introducing one or more of the four ways, thus ensures that the software will not deadlock. A fairly flexible methodology or CASE tool will allow the introduction of deadlock, but will test for it during analysis. Where it is detected, then the design can be altered, and re-analyse and re-modified until correct. However, where the user serendipitously produces a correct design, then the design is not understood (at least by the user). Where undetected, then fault tolerance might ensure correct operation.

A fault tolerant system may be allowed to operate knowing that the deadlock:

- will happen under specific circumstances
- will happen under unknown circumstances
- might have been undetected during inadequate analysis

Any fault tolerance adds to complexity, and diminishes comprehensibility.

This does not resolve the balance between correctness, flexibility and comprehensibility, but justifies a sacrifice in flexibility for a correct and comprehensible design, and justifies a methodology or CASE tool whose inflexibility produces correct and comprehensible designs.

Guidance Throughout the Life Cycle

The earliest stages of the development life cycle have the greatest influence on the final software, and are often the most difficult to get right (section 3.2.1).

CASE tools tend to support the later stages of software development and provide little assistance for the important life cycle beginnings. Indeed, it is hard to identify where computers can assist in the creative processes in design (refer to section 4.5).

If CASE tools are the most appropriate approach to achieving dependable parallel systems (section 4.7.6.1), then CASE tools and their underlying methodologies must provide guidance for early, pre-specification, stages in system development. Chapter 6 defines the steps and tasks of the methodology, which begins with manufacturing requirements and ends with occam code.

One significant difference in assistance given by the methodology and the occam compiler is that the compiler indicates what and where errors were found, while the methodology provides no such help. This is a good reason for keeping the methodology simple and restricting the users design freedoms, rather than having to make the methodology into a CASE tool which could highlight errors.

Petri Net/Occam Equivalence

Place-transition Petri nets and occam models can be translated in both directions, at the micro level, with significant exceptions:

Petri net to occam

- the Petri net outputOR has no occam equivalent (refer to Figure 3-13)

Occam to Petri nets

- occam priorities and timers cannot be modelled in place-transition Petri nets, but Petri nets with inhibitor arcs and time or stochastic Petri nets can
- occam Boolean variables can be modelled, but high level rather than place-transition Petri net are needed to model other data types

In the software development life cycle, only the Petri net to occam is relevant. By preventing the use of the Petri net outputOR a Petri net/occam equivalence can be established. Section 5.3.3 discusses that the methodology does not rely on the equivalence mathematically, but makes use of the structural similarities.

If Petri nets were to be modelled in a sequential language, then the sequential code would include many data structures to represent the five constituent sets (place, transition, input arc, output arc, marking). A number of these data structures are represented by the occam language structure, as shown in Table 6-1. These are exploited in the research methodology.

PETRI NET	OCCAM
places	variables (within a controller) !? communication (between controllers)
transition	IF construct (within a controller)
input arcs	choice in IF (within a controller)
output arcs	process in IF (within a controller)
marking	variable values

Table 6-1 Petri net and occam equivalence

There are features in the Petri nets that are beneficial and those that are disadvantages. State machines and marked graphs have many desirable properties, refer to section 3.3.2.1, but are deficient in modelling and decision power respectively. Petri nets and high level Petri nets have modelling power, but the Petri net designer is free to produce nets which consist of conflicts, self-loops and deadlocks and which are not bounded, safe, conservative, live nor terminate properly. By imposing appropriate restrictions, then Petri nets can be produced which have desirable properties, but at the expense of modelling power.

Solution Variations

Greater confidence can be placed on a methodology if it is repeatable, i.e. it will produce the same result for a given problem. The relatively narrow problem and solution domains of the methodology improve the likelihood that two users to arrive at the same solutions from the same problem.

One doctrine behind the occam programming language is to minimise the number of ways of achieving the same thing, and was taken from 'Occam's razor': "one must not multiply entities without necessity". However at all levels of coding, there are choices in the way of writing occam code. For example, section 4.8.3 discusses occam equivalences through 22 'laws' of occam transformation. It is therefore unlikely that any two pieces of occam code are the same. If a minimalist language allows such variations, then other languages will be worse. The generic template and the tasks of step 4 aim to reduce the diversity at the implementation stage.

In a similar way, the methodology aims to reduce diversity at the design stage. Generally, there is much greater variation at this stage, because not only is choice available in the use of the design tool, but there is choice of design tool. By defining the design tool in the methodology (Petri nets), and providing tasks as to its use, specified in steps 2 and 3, then diversity is reduced.

If the choice and use of the design and implementation tools are defined in detail by the methodology, then solutions are produced with less diversity, more repeatability, thus more confidence, but with less flexibility.

How the Methodology Reduces Doubt

Table 4-1 highlights four areas of doubt inherent in development specifications: interpretation, consistency, completeness and validity. The methodology aims to help produce occam code for DCSs from the manufacturing requirements, thereby being applicable to the requirements, system specification, design and implementation stages of software development. This section identifies the measures employed in the methodology to reduce doubt in these areas.

Completeness- is assisted by Strategy 1 output work backwards. The outputs of controllers (exit places of Petri net graphs) originate from the manufacturing requirements, for example, 'start lathe'. By working backwards, as described in step 2, to satisfy these output conditions, then eventually the inputs necessary to the controller are produced (entry places made local of Petri net graphs). Every input to a controller must either correspond to a controller output or must communicate with the manufacturing equipment. The system specification is defined in step 3, from which step 4 translates the occam code. In this way, if the manufacturing requirements are provided then their DCS needs are met.

Validity- is inherent in the methodology. Steps 1 and 2 help create the requirements of the DCS, from the manufacturing requirements, and step 3 presents the overall specification. Goal 1 Petri net/occam equivalence enables the code to be derived directly from the system specification, and the functional requirements are created in the specification language.

Consistency- is helped by the comprehensibility of the Petri net graphs, and by cross-referencing between entry and exit places and replicating places. It is tested by manual consistency checks.

Interpretation- the labels or names given to places represent actions or statuses, such as 'start lathe' and 'lathe turning', and are the same labels in the code. The interpretation of the labels can be clarified by accompanying pseudo code and place descriptions in the documentation. In this way, the labels and pseudo code created at the requirements stage are the same as those in the system specification and code.

Formal Development

Section 4.4.1 gives four approaches to design. The methodology is certainly not ad hoc, and more than an informal design tool. It cannot be classed as formal, because there is no proof of steps between the system and implementation specifications. The methodology strives to be rigorous, by adopting formal tools (Petri net and occam), and by imposing restrictions to their use then an equivalence can be made between the tools.

The formal software development life cycle of section 0 is shown in Figure 3-2. The methodology is applicable to all stages. Functional requirements can be generated via Strategy 1 output work backwards, which considers all manufacturing inputs to be DCS outputs, and forces the designer to think of conditions necessary to satisfy the outputs, as just discussed above. In this way, DCS resource requirements can be revealed. The overall Petri net graph represents the system specification, and makes evident the architectural design, which includes the cell controller and the status handler. The narrow problem and solution domains of the methodology impose detail in the ways modules are laid out, and the coding is implemented.

Devolution of Responsibility of Comprehension

The complexities of parallel safety systems are a strain for humans to understand, for all but small systems. However it is essential that humans understand the system and the specifications.

CASE tools can greatly assist in the development of complex systems, but there is a possibility that CASE users devolve responsibility for understanding specifications and the development of parallel safety systems to the CASE tool. Where the input to the CASE tool is mainly graphical, and hierarchical in particular, then it would be easy to loose the overall comprehension of the system. It is better to be able to visualise the system representation as a whole as well as in understandable modules.

Computers do not understand anything more than one machine instruction at a time. It is only the logic of the source code, its correct compilation and its execution that make humans believe that computers know what is happening. Thus, computers do not have an overall comprehension of the specification, so humans must.

This is not an argument against CASE tools, but advocates the adoption of an underlying methodology which facilitates comprehensibility and gives confidence to the CASE user and subsequently the system user at specification validation and during normal operation of the system.

Requirements' Influence

Design is a creative process that transforms the system specification ready for coding, while accounting for the non-functional system characteristics. Where the non-functional characteristics are significant, then the specification and the code are quite different structurally. Where they are insignificant then the specifications are structurally similar.

Techniques applicable to DCSs, not readily available to parallel or sequential processing, simplify the structural changes required by significant non-functional characteristics. For example:

- performance can be improved by writing the code in assembly language, which is relatively incomprehensible. However in DCSs, by dividing the workload on to more processors, then the task can be completed in less time
- reliability can be improved by writing fault tolerant software. However in DCSs, by Triple modular redundancy [Anderson 1981] the code can be run on three separate processors, their results compared and the correct one is output
- flexibility of expansion is facilitated by modularisation. In DCSs, modular processors can be added with the modular code

The research methodology helps create the DCS requirements from the manufacturing requirements resulting in modular Petri net graphs. It is required to be expandable, but performance and reliability are left for further work. However, replication of Petri net graphs modules for performance and reliability purposes is facilitated, and the combining of the Petri net graphs results in an overall specification, which is readily translatable into occam code.

It is possible for a customer to desire the distributed control of an 'entry priority roundabout', which will inherently deadlock in one or more of the four ways discussed in section 4.10. When the potential to deadlock is identified, then the techniques to manage deadlock must be applied. This disadvantageous characteristic of DCSs is in contrast to the three beneficial ones mentioned above.

The methodology does not provide analysis tools to identify deadlock, but use of the methodology will prevent deadlock by 'circular wait'.

Validation

Petri nets are capable of analysis, but the methodology makes no use of this property, however through Petri net/occam equivalence and pro-activity the Petri net formalism is exploited to 'get-it-right-first-time'. Validation through animation is also possible in Petri nets, but is believed that the transformation from Petri net graph into occam code is easier in the methodology than to produce an animation tool, refer to section 3.3.1.2, so the model validation is done as an occam simulation. This has the added advantage of including temporal properties and priorities, such as livelock.

Verification

The tools and methods have been adopted by the methodology for their appropriateness. They can be compared with the verification techniques of section 4.11 summarised in Table 4-2. Table 6-1 shows the tools and techniques used for stages in the software development life cycle of the methodology, and how

one stage verifies with its neighbour follows: The transputer, the relationship between the transputer and occam, and Petri nets are the tools employed for the design to the execution stages, and their mathematical bases substitute for verification. The steps of the methodology apply to the stages beginning at the top level requirements and ending with the program code. They also bind the design and the coding together. To satisfy the verification between other stages there was a need to establish a relationship, or develop a verification procedure, between Petri nets and occam; and techniques are needed to help draw out DCSs requirements from the manufacturing requirements. The approach taken is again to substitute rather than satisfy the verification by producing an overall methodology with core strategies which tackled both the relationship and techniques during the steps of the methodology.

STAGE \ LEVEL	TOOL/TECHNIQUE	VERIFICATION
Top level requirements	methodology Petri net	methodology
Design	Petri net	methodology
Program code	occam	Petri net/occam equivalence
Object code	transputer	occam/transputer
Execution	transputer	

Table 6-1 Tools, techniques and verification for the methodology

Pseudo Code

This thesis defines four steps in the methodology, and ends with a single transputer or simulation implementation. The tasks of step 2 enable clarification and additional definition via pseudo code, but none is used in the FMC example. The omission was to prevent clutter and unnecessary detail in the system and implementation specifications.

The job list is an example where Petri nets cannot easily describe inclusive OR options, so requires the assistance of pseudo code. The job list of Table 2-4, described in section 2.3.1, is transformed into pseudo code and translated into occam in section 6.5.8. The simulation code, in the appendices, specifies that a list of components is loaded onto three pallets to be turned and milled, see the `robot.unloaded` alternative in `statHand.occ`.

Initial Conditions

Petri net graphs allow the initial conditions to be depicted as the initial marking within the graphical specification. Occam can have the initial state defined immediately above the 'operating specification', but it seems more natural to separate the initial state from the operating specification. This is implemented as the `initial.conditions` procedure in the `statHand.inc` file.

6.7.4 Specification Observations

Updating the Cell Status and Place Replication

How often to update, to prevent deadlock, is indicated by idea sharing which is manifest on Petri net graphs by place replications and crossing arcs. Where there are no replications or crossing arcs then

updating need only occur once per index. This would characterise three situations: a simple system, that Boolean algebra had been performed or that there is little communication between controllers. Where there are no replications, but many crossing arcs, then this would indicate an unreadable Petri net graph.

Place replication is a consequence of comprehensibility. It is easier for designers to create, communicate and maintain DCSs if ideas are modularised into small chunks which interface well. Petri net graph entry and exit places are the interfaces.

The term replication was chosen, because the places represent the same idea, but the representation occurs more than once. Consistency checking will fail if different ideas are labelled with the same name. Replication of Petri net graph places generally only happens as entry and exit places made local, and not as channels.

Petri Net Place Representation

There are three types of Petri net place in the methodology. Their distinction is relevant to documentation and coding, and is an aid to consistency checks with the Petri net graphs. The meaning of the simple (single circle) place is a simple instruction, and that of a complex (double concentric circles) place is other than a simple instruction. This task is purposefully vague to allow flexibility of representation. Consolidated entry and exit places combine entry and exit places, so that they can become an occam sequential protocol.

Petri net places are passive entities, whose marked and unmarked states can be interpreted in a number of ways, such as :

1. no meaning, but are needed for structural correctness between two transitions
2. a Boolean state, rendering the label TRUE or FALSE
3. a state exists when holding, or is irrelevant or meaningless when not holding
4. a state is active or inactive
5. a reservoir, repository or counter

Inhibitor arcs, refer to section 3.3.2.2, increase the modelling power of place-transition Petri nets considerably. They can only be input arcs, and, together with their input places, allow a test for zero tokens. The interpretations assigned to 'inhibitor places' is inevitably different, e.g. the obverse of 2, 3 and 4 above.

Care must be taken when using inhibitor arcs, because the input place to a transition linked by an inhibitor arc must be emptied before the transition can fire. This requires the place to have at least one non-inhibitor arc leaving it, if it is ever given a token. It is recommended that inhibitor arcs are not used in the methodology, because the Strategy 1 output work backwards mostly leads to places having fewer arcs leaving than entering. A substitute for the inhibitor arc is introducing places with the reverse meaning. In the example Pallet at lathe statuses in the status handler, refer to Figure 6-13, a pallet is full or empty, for turning or not for turning.

Complex Representation

Places with double concentric circles represent other than simple information, see task 3 of step 2. The status handler records the state of the system by maintaining the values of the statuses. Generally, other controllers are provided with information which is constant, albeit until replenished with fresh information. For example, the cell controller requests updates in order to determine the appropriate instructions. Also the workstation controllers can either receive instructions and part programs, or receive instructions which require the loading of locally stored part programs. In the example FMC, the header files of the workstation controllers, e.g. `latheHd.inc`, store machining and loading times to represent locally stored part programs.

The job list, refer to Table 2-4 in section 2.3.1, is a set of manufacturing requirements for various similar components. Some of the chess pieces need milling and turning, and the pawns need turning only. The job list is thus a complicated data structure of indeterminate length. However, it is abstractly represented in the Petri net graphs of Figure 6-14 and Figure 6-19. The pallet at robot statuses of Figure 6-14 shows the parts list as a single place with double concentric circles. The double circles alerts the user that place 15 is more than a simple place, and is defined further elsewhere. Figure 6-19 shows the parts list as four places, each representing a part, and each drawn with double circles. A part represents a data structure consisting of a flag job in parts list, and statuses part not for turning, part for turning, part for milling and part not for milling. Statuses part for turning and part for milling are also given double circles to indicate a further abstraction, which represents component codes (e.g. LK(4,W)). Figure 6-19 also shows the preliminary pseudo code representing the transition relating to loading the robot. The final pseudo code is given in fig 7---20, and the occam code is in the appendix. The robot status ceases to be idle (`RS.idle:=FALSE`) and the status of pallet at the robot becomes full (`PS.full[Robot]:=TRUE`) in file `statHand.occ` under channel `cc2sh.robot` tag `load.robot.cc2sh`. Other statuses are changed in procedure `get.next.job.or.stop()` which is in file `statHand.inc`.

A more satisfactory approach to complex representation is left to further work.

Token Game

The structure of the Petri net graphs has been considered much more than the use of tokens. The simplicity of the workstation controllers would present few problems to the token game. The status handler shows the initial conditions in the Petri net graph marking and the token game would largely be applicable to it. The cell controller has several replicated entry places made local which could complicate the token game. Local places are assumed empty of tokens when the next update of statuses arrives.

The replicated entry and exit places of controllers should be superimposed and a new overall Petri net graph drawn before a proper token game (and other analyses) can be performed. This would be unsatisfactory in the methodology, because the resultant Petri net graph would be unreadable, and resemble the 'labyrinth' Petri net graph.

The pro-activity of the methodology precludes the need for the token game, and validation is intended to be established when the simulation code is complete.

6.7.5 Coding Observations

Three Hierarchical Levels in Occam

In a similar manner that sequential software is best written in a (convenient) hierarchy of modular code, occam parallel software, is necessarily written in a hierarchy of modular processes.

By visual inspection, refer to section 4.8.6, occam programs consist of three hierarchical levels:

- the top overall level
- the middle communications level
- the bottom sequential level

A simple program might only consist of the three distinct levels. A large system might require many sub-levels to describe the code effectively. Even though the folding editor precludes the need for many occam procedures, code produced with it and with ordinary 'flat' editors can show the three levels equally well.

Software Conservation

There are several advantages of conservation of modules of code: there is less to write, less to have to understand, and once correct the code can be reused with confidence.

During the production of code, it is quicker to work with many prepared and pre-compiled modules. The template of Figure 5-2 is the starting point for all controller procedures. Header files are `#INCLUDE`d in the controller and the top files. In the example FMC, the header files of the workstation controllers (`latheHd.inc`, `millHd.inc`, `robotHd.inc` and `convHd.inc`) are used in the lathe, miller, robot and conveyor files, respectively, and also in the header files of the cell and status handler controllers and the top file (`cellHd.inc`, `statHd.inc` and `topHd.inc`). The disadvantage occurs when a value such as `Lathe.machining.time` is altered, then the lathe, cell, status handler and the top files must be recompiled. Reuse of such header files is possible because the naming convention, taken from the Petri net graphs, provides unique names throughout the overall system. The uniqueness of the names in the code can be tested using the spell-checker of a word-processor, which disallows mixed cases and full-stops within words.

Indentation

Indentation, in occam code, represents delimitation of constructions and scoping. During the production of comprehensible code, using meaningful names, the line width is soon reached, and many processes have to be carried over and occupy several lines. Readability can be maintained while using lucid names by structuring the layout and imposing appropriate indentations, however a file might extend for many pages. For example, `robot.conditions()` in the cell controller in the appendices has Boolean expressions which take three lines. Each line should be indented at least three spaces to differentiate it from the normal two space indentation needed for delimitation.

Redundancy for Readability

Redundant `SEQ` constructs appear in the code to aid clarity and maintenance. They appear between a lengthy Boolean expression and a single guarded process of a condition, such as in `index.wanted()` in the cell controller, where its inclusion highlights the end of the Boolean expression. They appear in long processes to avoid having to indent all lines to rectify an omission, such as in `update.statuses()`, `show.machine.statuses()` and `show.pallet.statuses()` in `statHd.inc`. This is unnecessary when using the folding editor.

6.7.6 Specification Validity

The specification validity can be expressed as three important considerations:

- the manufacturing needs must be met by the DCS
- the Petri net graphs must represent a dependable DCS
- the occam code must represent the Petri net graphs

The goals and the strategies address these considerations as follows:

Strategy 1 output work backwards helps satisfy the manufacturing requirements by making them the outputs of the DCS. Working backwards from each of these outputs ensures that none is missed. Outputs are drawn on the right of a controller and its inputs are drawn on the left. Outputs and inputs of the DCS of the 'labyrinth' Petri net graph are not obvious.

The concurrent nature of an FMC is represented in Petri nets, a concurrent and mathematically based modelling tool. Petri nets allow unreadable graphical specifications to be drawn, but by following the methodology, more comprehensible ones can be produced; compare the 'labyrinth' Petri net graph and the overall Petri net graph. The methodology divides actions into concurrent and sequential ones, and determines their relative position on the overall Petri net graph by Strategy 3 structuralise and modularise. It is clear which graph relates to which controller.

The individual controller Petri net graphs, which make up the overall Petri net graph, show the output communication from, and the input communication to, the controller as exit and entry places, which cross the controller boundaries. Also the source and destination references indicate with which other controllers the controller communicates.

The synergy of the Petri net structure and place names and their discretionary accompanying descriptions and pseudo-code enable the representations of the Petri net graphs to be understood.

Dependability of the whole DCS is facilitated by Strategy 1 output work backwards and the cell controller and status handler as described in section 5.3.9. By addressing the outputs of the workstation controllers, and by working back via the cell controller, the status handler and back to the workstation controllers, then a predominately uni-directional closed loop is produced. By managing a break in the closed loop in the cell controller, then the closed loop is maintained, but deadlock by circular wait is removed. Comprehensibility aids all stages of the software development cycle.

The occam structure, modules, statuses and communication and their names are taken from the Petri net graphs. Goal 1 Petri net/occam equivalence forms the theoretical basis of the representation, and is aided by all of the strategies. The major differences between the occam and Petri net representation can be seen in Figure 5-2, and is explained in section 6.3.3. Where pseudo-code accompanies the Petri net graph, then the occam should be coded from that. The accompanying Petri net place description can appear in the occam code as comments. The other differences, discussed in section 6.5, are that the cell controller only has the update as the input channel, and that the procedures that perform human-computer interfacing are hidden from the status handler and are kept in its include file `statHand.inc`. The remainder of the code represents the Petri net graphs.

6.7.7 The Relationship Between Levels 1 and 2

Little has been discussed about the relationship between the level 1 and 2 controllers, described in sections 2.5 and 2.6, because level 1 is outside the scope of this thesis. However, there is potential for deadlock due to the loops indicated by the outputs and corresponding inputs of the level 2 workstation controllers, e.g. exit place load lathe and the entry place lathe loaded of the lathe controller in Figure 6-1.

The nature of the relationship is far stronger between levels 1 and 2 than between levels 2 and 3. The loop between levels 2 and 3 is flexible and allows any number of workstations to be added between the cell controller and status handler (ignoring deficiencies in the number of transputer links). Deadlock, by circular wait, is avoided at these levels by Strategy 1 output work backwards, refer to section 5.3.6.

Level 1 controllers are CNCs and PLCs, which are sequential. The function of the level 2 controllers are to instruct and synchronise the level 1 controllers in the loading and unloading of parts between pallet and machine tool (and to start the machine tool). This relationship need not be as flexible, and is similar to the client server model of communication described in section 3.5.2, which is a method of deadlock avoidance described in section 4.10.4.

6.7.8 Boolean Algebra

Boolean Algebra is a mathematical technique that can enable the rearrangement of Boolean statuses. It can be applied to safe Petri nets and ones with replicated places.

One consequence of comprehensibility is the replication of entry places, and is made obvious by the place numbering and the associated replication cross referencing codes, refer to section 6.3.4. An exercise was conducted to compare this approach with one which eliminated replication of entry places in an earlier version of index conditions, of Figure A3. The Boolean algebra and the resultant Petri net graph, Figure A4, are given in the appendices. The reduced Petri net graph is much less easy to understand than that of Figure A3, because apart from highlighting the lathe, miller and robot (one of the goals of the algebra) there is no meaning to the Petri net graph.

It is possible to extend the exercise to the whole cell controller to eliminate replication, thus reduce the size of the Petri net graph and subsequent occam code. As a consequence, however, both the Petri net graph and the occam code of the cell controller would have to be understood in their entirety, and could

not be broken down into modules small enough to absorb and comprehend. Drawing a reduced Petri net graph of the entire cell controller would show many more arcs crossing each other, than in the reduced Petri net graph for indexing alone.

Attempts to resolve these problems would either lead to complete integration and the unreadable 'labyrinth' Petri net graph, or to simplification and the overall Petri net graph.

6.8 Comparison of the Methodology with Other Applications

A comparison of the applications presented in section 4.7 is made with the methodology against criteria of the goals and strategies defined in section 5.4. The comparison is summarised in Table 6-2. For example, Carpenter's work in section 4.7.1.1 has a better Petri net/occam equivalence and deadlock handling capability than the methodology; provides similar pro-active guidance; but will produce a Petri net graph which is less modular and structured, which is harder to identify sequential and concurrent actions and is less comprehensible than that produced with the research methodology.

Section 4.7.	1.1	2.1	3.1	3.2	4.1	4.2	5.1	6.1	7.1	7.2	7.3
Criteria \ Name	Carpenter	Balbo	Mars	Breant	Kerridge	Gorton	Lau	Birkinsha	Manson	Jelly	Schafers
PN/occam equiv	+	+	-	-	-	-	+	-	-	N	N
Deadlock handling	+	+	+	+	+	+	+	+	-	-	-
Pro-activity	S	-	-	-	-	-	-	-	-	-	-
Modular/structural	-	-	-	-	-	-	-	-	S	S	S
Sequential/conc	-	-	-	-	-	-	-	-	S	S	S
Comprehensibility	-	-	-	-	-	-	-	-	-	-	-

Key: + better than, - worse than, S similar to the methodology, N not applicable

Table 6-2 Comparison of the methodology and applications against the goals and strategies

The applications and the methodology are examined together against each criterion in turn.

Petri Net/Occam Equivalence

Carpenter, Balbo and Lau model occam constructs (e.g. PAR, ALT and IF) in Petri nets, and combine the Petri net constructs during Petri net modelling. This results in a micro level model which is equivalent, but all models are very detailed. The methodology has a weaker relationship between Petri net and occam, which is achieved through output-work-backwards, but enables the production of Petri net models at a suitable macro level, which leads to better comprehensibility. Other applications do not prepare or restrict Petri nets especially for translation into occam. Manson's application is based on CSP, and does not use Petri nets.

Deadlock Handling

Schafers enables deadlock to be seen, but does not analyse for it or prevent it. Jelly allows analysis if Petri nets are used, but enforces neither. Manson does not ensure that the model is built according to the CSP formalism. Carpenter identifies causes of deadlock and warns against construction of such arrangements. The methodology relies on output-work-backwards to produce a Petri net model which will not deadlock. The others build complete Petri net models to enable analysis and then correct the model before the occam is produced.

Pro-activity

Only Carpenter and the methodology provide guidance to prevent the introduction of deadlock.

Modular/Structural

The methodology, Schafers, Manson and Jelly provide guidance on overall structure of the model, while none of the others do. Only the methodology determines that the code modules should be produced directly from the Petri net model modules.

Sequential/Concurrent Actions

The methodology, Schafers, Manson and Jelly enable identification of sequential and concurrent actions, while the others make no distinction. The methodology presents the entire Petri net model at one level, to emphasise the communication between concurrent actions. The other three depend on hierarchies of process graphs or data flow diagrams, where the top level shows the concurrency and the users are required to remember how the middle and lower levels interrelate.

Comprehensibility

Schafers preserves the naming of channels and variables between the system specification and the code, and produces a structured model. Jelly indicates the importance of overall structure and modularity, and differentiates between sequential and concurrent actions. However, they and the other applications present a worse overall comprehensibility, and comprehensibility improves dependability at all stages in the software development life-cycle.

The criteria are bias towards the methodology, because the goals and strategies are the foundation of the methodology. However, goals and strategies are derived from the techniques and considerations, which achieve the aims of the methodology. In turn, the aims are derived from conclusions, refer to section 4.12, which highlight deficiencies and benefits of applications and software engineering practice in developing flexible and dependable DCSs.

The applications fall into two categories: comprehensible and analysable. Schafers, Manson and Jelly present comprehensible hierarchies of process graphs or data flow diagrams, so, with the exception of Jelly which allows the use of fragmented Petri nets at the lowest levels, are not in a form to analyse. The

other applications use Petri nets as analysis rather than graphical tools, because they are hard to read, but are complete and analysable.

The methodology straddles the two categories. It enables the development of Petri net graphs which are more comprehensible, while at the same time are absent of deadlock.

7. Conclusions and Recommendations

7.1 Conclusions

Industry requires suitable software engineering tools and techniques to develop dependable and flexible DCSs for use in FMCs and other plant. However, current techniques have the following deficiencies (bold indicates contribution to knowledge):

- industry does not want to use formal tools, because they are too difficult to learn and use, and specifications are not understood by customers
- most formal techniques have a small problem domain, and apply to one or two stages in the software development life cycle
- the earlier stages in the software development life cycle are more difficult to get right, are the most pivotal and are the stages which are least targeted by rigorous tools
- rigorous tools tend to concentrate on analysis. The cycle of 'modification and re-analysis until correct' is laborious, and a correct solution might be arrived at without the user knowing why
- it is better to prevent or avoid the inclusion of faults than to eliminate or tolerate them. Fault avoidance impedes flexibility, but reduces the need for elimination and tolerance
- **graphical modelling is used heavily in parallel methodologies or CASE tools, but is not fully exploited. The graphical interfaces or modelling tools in some CASE tools hinder the comprehension of the design**
- **where CASE tools and methodologies facilitate the development of part of the design, but not the overall design, then the user is devolving the responsibility for understanding the design to the tool. The complexities of parallel and safety systems are easier for humans to understand if approached modularly, but the user must understand the whole design where safety is concerned**

There is a need for a methodology that overcomes these deficiencies while achieving dependability and flexibility in distributed control.

The objective of this work was therefore to produce a Petri net, occam and transputer based methodology for the comprehensible development of dependable and flexible distributed control applied to a flexible manufacturing cell. The methodology, defined in chapter 5 and discussed in chapter 6, largely meets this objective, and provides an original contribution to knowledge.

The methodology consists of steps and tasks to guide the designer to produce correct and reliable occam code, and provides a foundation for further work to produce the safe control of a transputer based FMC. The methodology enables the designer to:

- generate DCS requirements from the manufacturing requirements
- produce modular chunks of Petri net graphs, and lay them out in a readable structure
- make obvious the concurrent and sequential actions

- produce comprehensible system and implementation specifications
- communicate the system specification with non-mathematical customers for validation
- produce a specification more quickly and reliably which results in deadlock free code
- modify the specification for maintenance or expansion

During the work additional conclusions were revealed:

- **occam and transputers make a dependable DCS for use in FMCs, where flexible, correct, reliable, safe and real time operations are required**
- Petri nets and occam are increasingly being used in tandem
- **by restricting Petri nets, a (non-mathematical) equivalence can be made between the structures of Petri nets and occam**
- **by following an output-work-backward development approach:**
    - **the equivalence can be achieved**
    - **a predominantly uni-directional, and closed loop, communication is produced**
- **by careful management of the closed loop, deadlock by circular-wait is avoided, while maintaining the benefits of closed loop control**

The methodology compares favourably with the applications discussed in section 4.7, when judged against the methodology's goals and strategies (section 5.4) as criteria, see section 6.8.

The methodology's narrow problem and solution domains limit its applicability. Section 6.6 determines that the methodology applies to the shop-floor message standard MMS, and to systems suitable for other CSP languages such as Ada.

7.2 Recommendations for Further Work

Most of the research project is new to the School, and much of the work is viewed from fresh perspectives. These and the novel aspects discussed in the thesis provide many questions relating to the topics listed below:

1. Petri net analysis
2. occam to transputer configuration
3. topological considerations
4. testing
5. documentation
6. job lists and other data handling
7. error handling/fault tolerance
8. translation into a language other than occam
9. proof of the Petri net/occam equivalence
10. high level Petri nets
11. exploiting the transputers real time capabilities
12. address level 1 machine tool control
13. computer automation of the Petri net to occam translation
14. computerisation of the whole methodology
15. integration into a CADCAM or CIM system
16. an assessment of the methodology's usefulness given the T9000 and occam 3
17. OSI and other manufacturing standards

Petri Net Analysis

The one advantage the 'labyrinth' Petri net graph has over the overall Petri net graph, or other graphs produced according to the methodology, is that it is a complete Petri net with no replicated places and without exit places mapping to entry places, so it can be analysed as it stands, refer to section 3.3.1.5. It is unlikely that the labyrinth is successfully analysed manually, but a computer package might be able.

If the methodology is computer assisted then the mapping would be done internally and the analyses performed in the usual way. However if computerised, the code would be produced automatically, and the FMC could be simulated, which is better validation than Petri net animation.

Occam to Transputer Configuration

A fifth step, after the Petri net graph to occam translation, can be to configure the occam code onto a network of transputers. Modular code, brought out by the methodology, can quickly be partitioned, so processes are mapped to transputers and channels onto communication links, see section 3.5.5.6.

The methodology has assumed light work and communication load, and emphasises a distributed control approach, so load balancing is of no concern.

Configuration will merely require the connection of hardware, and a modification to the top or overall occam procedure to include the configuration code to perform the appropriate mapping. It is likely that this new step can be computer assisted.

Dynamic re-configuration will be necessary if fault tolerance or rapid machine changes are introduced in the FMC.

Topological Considerations

The four links of a transputer are not a problem in an occam simulation, but are significant in a DCS. The example FMC can be partitioned onto four transputers: cell controller/status handler controller, miller controller, lathe controller and robot controller/conveyor controller.

Common topologies for transputer distribution are tree, ring and farm networks. Work is required to examine how such topologies integrate into the methodology.

Testing

The goal of pro-activity and the strategy of getting it right first time should reduce the testing process. The focus of the output in the output-work-backward and modular/structure strategies should facilitate testing. The methodology employs visual inspection for reviews and consistency checks, and validation by occam prototype execution rather than by Petri net animation. A set of steps and tasks for testing might be needed that fit in with and possibly augment the core goals.

Documentation

Def Stan 00-55 determines that safety systems should be specified formally and in plain English, and cover "specification, design integration, verification validation and in-service support". The methodology begins with the manufacturing requirements to help create the DCS requirements, specify the system and specify the code. The specifications are therefore largely Petri net graphs and occam code, but pseudo code is used to clarify and augment Petri net graph specifications. Documentation contributes to comprehensibility, so is important, but how documentation is integrated into the methodology is left for further work.

Job List and Other Data Handling

Place-transition Petri nets are not suitable for representing information of the form required by an ever changing job list, and the solution of using pseudo code to overcome this deficiency is not ideal. For example, the job list in Table 2-4 specifies the job number, description code for the chess piece and the size of the chess set. The description determines which part programs to load into the lathe, the miller or both. The 'pallet at robot statuses' Petri net graph of the status handler (Figure 6-14), where the jobs (material and information) are loaded on to the pallet, shows that parts are for turning and/or milling, and the accompanying pseudo code is discussed in section 6.5.8.

Data sets are used in many DCS applications. If their representation is found to be necessary, then one solution is to use high level Petri nets.

Error Handling and Fault Tolerance

Where the integrity of the hardware and software is not appropriate, then measures to enhance reliability and correctness are needed. The range of measures goes from reporting errors and failing-safe to complete fault tolerance and recovery.

It is envisaged that error handling can conform to the predominant uni-directional communication. Errors indicated by machine tools and work handling can be sent to and dealt with by the status handler in the same way as normal messages.

Faults manifest in any of the controllers would require more sophisticated measures. It is likely that an error handling controller, running in parallel with the other controllers, is needed.

The methodology is very hierarchical, so does not lend itself to the autonomous fault tolerance approaches. For example, if the cell controller, status handler or error handler failed, then the cell would not be able to recover.

Translation into a Language Other than Occam

Languages based on CSP could be substituted for occam. It is hoped that any language developed to fully exploit the transputer also has potential. To accomplish this, the methodology would need an intermediate step between the Petri net graph synthesis (present step 3) and the translation into occam (present step 4). The intermediate step would be the translation into a generic concurrent language, like occam but with general labels, such as 'loop' for occam WHILE and 'channel input choice' for occam ALT.

Proof of the Petri Net/Occam Equivalence

The methodology exploits two mathematically based tools, but its tasks make no reference to mathematics. This should be attractive to industry where there is a reluctance to adopt formal methods, because of their complexity and difficulty of use.

To ensure confidence and enhance integrity of the methodology, then the equivalence should be proved. Occam is based on CSP (section 3.4), and Petri net on graph theory (section 3.3).

Exploitation of the Transputer's Real Time Capabilities

The FMC must be controlled in real time to maintain optimum utilisation and achieve lead time targets. Occam and the transputer have timing characteristics (see section 3.5.5), but place-transition Petri nets do not. Place-transition Petri nets were chosen for the methodology because they provide sufficient modelling and decision power, are inherently visual and the processing and communication loads are trivial in the example system, so as not to effect timing. However, where loads become significant or where timing needs to be modelled at the design stage, then high level Petri nets, e.g. timing Petri nets and stochastic Petri nets (see section 3.3.2) could be adopted, or the methodology could be augmented by temporal logic.

High Level Petri Nets

Some of the goals of the methodology arose from the tools chosen for the methodology. It would be interesting to compare methodologies based on place-transition Petri nets and high level Petri nets, against the objectives as criteria. It is anticipated that visualisation would suffer, especially where predicates, or arc functions, are written on the Petri net graphs.

Level 1 Machine Control

The methodology does not address level 1 of the CIM hierarchy, shown in Figure 2-8. The example, in appendices, presented to emphasise the synchronisation requirement of level 1 controllers, provides a tentative solution.

The major difference between level 1 and levels 2 and 3 are that the level 1 controllers deal with sequential rather than concurrent execution.

Computer Automation of the Petri Net to Occam Translation

Occam code can be generated from the Petri net graph specification according to the tasks of step 4, refer to section 5.9. The input to the code generator could be in Petri net graph form or in the form of matrices for the input and output functions

Computerisation of the Whole Methodology

The system specification can be created with the assistance of a purpose built drawing package. The package would allow the user to create the system specification within the tasks of steps 1 to 3. This could directly feed into a computer automated Petri net to occam translation to provide a complete package.

Integration into a CADCAM or CIM System

The methodology addresses the lower levels of the CIM hierarchy. It is possible to integrate the methodology into, or expand it to, a CADCAM or CIM system. The methodology aims to help in process planning and CAM control. A company employing lean-production or other such technique wishing to improve:

- time-to-market might (section 1.5) require the effect due to a new product on process planning, at the design stage
- manufacturing lead times (section 1.4) might want to know how the insertion of a rush job changes the work-in-progress of the jobs waiting for the CAM cell.

The Usefulness of the Methodology given the New T9000 and Occam 3

A significant feature of the prospective T9000 and occam 3 is that messages are automatically routed in a network of transputers, so configuration and multiplexing are made easier.

OSI and other Manufacturing Standards

Section 3.5.7 discusses the OSI/MMS version of industrial communication, and is compared with the transputer/occam version in section 3.5.5. It is likely that the occam communication protocols can be

specified according to MMS. This would require changes to the variable declarations, so it would be better to specify channel and variable definitions together, and then only header files need be altered.

Appendices

Pseudocode

```
CELL CONTROL - 1ST STAGE

Initialise system [status setup, safe check]
Input job list
WHILE (a job remains unfinished)
  IF (cell hold)
       [cell hold = operator intervention or device error]
    Report cause to operator
    Wait for operator run command
    cell hold = FALSE
  ENDIF
  REPEAT
    IF (transfer(s) required and permissible)
      Enable appropriate transfer(s)
        [between conveyor & machine(s)]
      Include part program for PUMA transfer
    ENDIF
    IF (machining operation(s) required
        ..AND permissible)
      Enable appropriate operations
      Include part program download
    ENDIF
  UNTIL NOT transfer in progress
  IF (conveyor transport required
      ..AND permissible)
    Index conveyor
  ENDIF
ENDWHILE

Reinitialise system [setup status, check safe]
```

```
CELL CONTROL - 2ND STAGE

Initialise cell control status [previous device status
  = idle;  current status = not available]
FOR all devices
  Request initialise
ENDFOR

REPEAT
  Do nothing
UNTIL (all device statuses = available)

Read job list [disk file]

[operate cell]
WHILE (component in job list) OR (component in cell)
  [handle cell hold]
  WHILE (operator hold) OR (any device unavailable)
    Report cause to operator
    Wait for operator continue command
    IF (operator intervention = hold)
      operator intervention = run
    ENDIF
  ENDWHILE

[Handle component transfers, enable machining]
REPEAT
    [Puma transfers]
    IF (puma transferring) AND NOT (cell hold)
      IF (pallet at puma has no work associated)
            ..AND (component in job list)
        Enable puma load
        Include part program
        puma status = transferring
      ELSEIF (part on pallet at puma)
          ..AND (turning not required for part at puma)
          ..AND (milling not required for part at puma)
        Enable puma unload
        Include part program
        puma status = transferring
      ENDIF
    ENDIF

    [lathe transfer]      ------------  EXAMPLE PSEUDOCODE
    IF (lathe transfer not transferring)
       ..AND NOT (cell hold)
      IF (part on pallet at lathe) AND (turning required)
          ..AND (lathe idle)
        Enable lathe load
        Include part program
        lathe transfer status = transferring
      ELSEIF (pallet at lathe transfer has work associated)
          ..AND (no part on pallet at lathe)
          ..AND (job no. at pallet = job no. at lathe)
          ..AND (lathe machining finished)
        Enable lathe unload
        lathe transfer status = transferring
      ENDIF
    ENDIF                    ------------  EXAMPLE PSEUDOCODE

[miller transfer]
    IF (miller transfer not transferring)
        .. AND NOT (cell hold)
      IF (part on pallet at miller) AND (turning required)
          ..AND (miller idle)
        Enable miller load
        Include part program
```

```
      miller transfer status = transferring
      ELSEIF (pallet at miller transfer has work
              ..associated)
          ..AND (no part on pallet at miller)
          ..AND (job no. at pallet = job no. at miller)
          ..AND (miller machining finished)
        Enable miller unload
        miller transfer status = transferring
      ENDIF
    ENDIF

    [lathe machining]
    IF (awaiting machining at lathe) AND NOT (cell hold)
      Enable lathe machining
      Include part program
      lathe status = being machined
    ENDIF

    [miller machining]
    IF (awaiting machining at miller) AND NOT (cell hold)
      Enable miller machining
      Include part program
      miller status = being machined
    ENDIF

  UNTIL (puma, lathe transfer and miller
         ..transfer not transferring)

  [Conveyor index]
  IF NOT ((component in job list)
          ..AND (pallet at puma has no work associated))
          ..AND NOT (cell hold)
    IF (a pallet holds a part not requiring
          .. milling or turning)
        ..OR ((lathe finished machining)
              ..AND (appropriate pallet not a lathe))
        ..OR ((miller finished machining)
              ..AND (appropriate pallet not a miller))
        ..OR ((lathe idle)  AND (a pallet holds a part
              ..requiring turning & not milling))
        ..OR ((miller idle)  AND (a pallet holds a part
              ..requiring milling & not turning))
      Enable conveyor index [release
         ..dogs until stations cleared]
      REPEAT
        Do nothing
      UNTIL (pallets at all stations)
    ENDIF
  ENDIF
ENDWHILE
Reinitialise system [setup status, check safe]


MESSAGE HANDLING ROUTINE

IF (conveyor finished)
  Update pallet positions
ELSEIF (lathe machining finished)
  lathe status = machining finished
ELSEIF (lathe transfer finished)
  lathe transfer status = NOT transferring
  Find 'n' such that pallet n is at lathe [n = 1,2 or 3]
  IF (pallet n part on pallet status = no)
    pallet n part on pallet status = yes
    pallet n turning required status = no
    lathe status = no part present
  ELSE
```

```
    pallet n part on pallet status = no
    lathe status = awaiting machining
    lathe status = new job number
  ENDIF
ELSEIF (miller transfer finished)
  miller transfer status = NOT transferring
  Find 'n' such that pallet n is at miller [n = 1,2 or 3]
  IF (pallet n part on pallet status = no)
    pallet n part on pallet status = yes
    pallet n turning required status = no
    miller status = no part present
  ELSE
    pallet n part on pallet status = no
    miller status = awaiting machining
    miller status = new job number
  ENDIF
ELSEIF (puma transfer finished)
  puma transfer status = not transferring
  Find 'n' such that pallet n is at puma [n = 1,2 or 3]
  IF (pallet n part on pallet status = no)
    pallet n part on pallet status = yes
    pallet n status = part associated
    pallet n part description = new part description
      [part description comprises job number,
       ..turning required, milling required]
    Increment job list number
  ELSE
    pallet n part on pallet status = no
    pallet n status = no part associated
  ENDIF
ELSEIF (device not available)
        [device = conveyor, lathe, miller, puma,
         ..lathe transfer, miller transfer]
  Save device available status
  device status = not available
ELSEIF (device available)
  Restore device available status
  device status = available
ENDIF
```

Formal Petri Net Definition

Petri Nets
Petri nets are a specification language that can model deterministic concurrent processes, and model synchronous and asynchronous logic systems. Petri nets consist of a set of places and a set of transitions, where the places represent the processes and the transitions represent the synchronisation.

The formal definition of a Petri net, C, can be given as:
C = (P,T,I,O) where
P = {p1,p2,...pn} a finite set of places, n_0.
T = {t1,t2,...tm} a finite set of transitions, m_0.
$P \cap T = \phi$ P and T are disjoint. ϕ is the empty bag.
I : P > T∞ input function, mapping transitions to places
O : P > T∞ output function, mapping transitions to places
pi is an input place of transition tj if pi ∈ I(tj)
pi is an output place of transition tj if pi ∈ O(tj)

Petri Net Graphs
A Petri net graph is a graphical representation of a Petri net structure as a bipartite directed multigraph. The places are depicted as circles and transitions as bars. Directed arcs connect places and transitions. An input arc goes from the input place to the transition, while an output arc goes from the transition to the output place.

A Petri net graph, G = (V, A) where
V = {v1,v2,...vs} set of vertices
A = {a1,a2,...vr} bag of directed arcs
ai = (vj, vk) with vj, vk ∈ V. V can be partitioned into two disjoint sets P and T such that $V = P \cup T$, $P \cap T = \phi$. For each directed arc, ai ∈ A, if ai = (vj, vk), then either vj ∈ P and vk ∈ T or vj ∈ T and vk ∈ P.
The Petri net structure and Petri net graph are equivalent.

The Petri net described so far indicates paths of control flow, but does not show the 'state' of the system. Tokens are introduced to indicate progress through a system so that modelling can occur. The marking, μ, of the tokens on the Petri net produces the marked Petri net, M.
M = (C, μ) where
μ : P > N marking function, mapping tokens to places

Firing rules
A transition can fire when all its input places are enabled. A place is enabled when it contains one or more tokens.

A transition tj ∈ T in a marked Petri net is enabled if for all pi ∈ P, μ(pi) ≥ #(pi, I(tj)) where # denotes 'number of'

When a transition fires, it removes a token from all of its input places and adds a token to all of its output places. A new marking, μ', results after a firing
given as μ'(pi) = μ(pi) - #(pi, I(tj)) + #(pi, O(tj))

Petri net state spaces
The 'state' of a Petri net is its marking. The 'next state function' of a Petri net, denoted Ù, is a change function. The change in marking μ due to firing tj is μ', or $\delta(\mu, tj) = \mu'$. Also $\mu'' = \delta(\mu', tj)$, where μ' is immediately 'reachable' from μ, and μ'' is immediately 'reachable' from μ'.

Reachability
The reachability set $[R(C,\mu)]$ for a Petri net [C] with marking μ is defined as all markings that are reachable from μ. A marking μ' is in $[R(C,\mu)]$ if there is any sequence of transition firings which will change marking μ into μ'. This is the reachability problem. The reachability set $[R(C,\mu)]$ for a Petri net [C] with marking μ is defined as

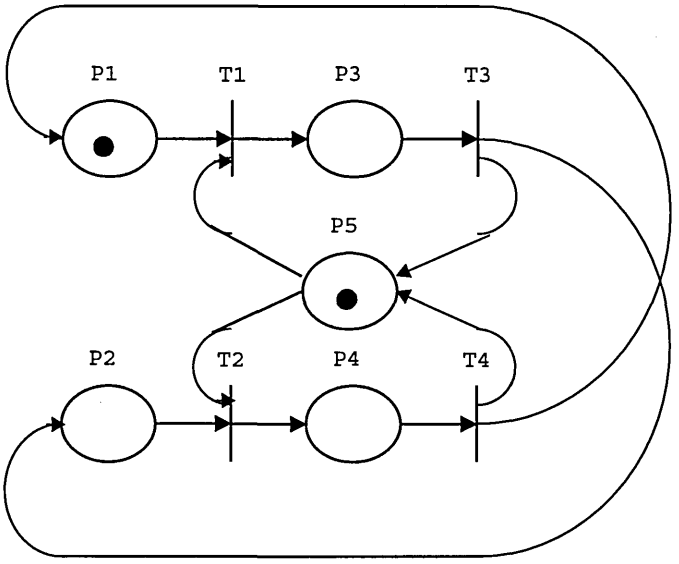
1. $\mu \in R(C,\mu)$
2. If $\mu' \in R(C,\mu)$ and $\mu'' = \delta(\mu',tj)$, then $\mu'' \in R(C,\mu)$

Petri Net Analysis Example

Reachability Tree Analysis

The Petri net graph of a simplified 'lathe transfer' is given in the figure below. The corresponding place numbers, initial marking and name are given in the table.

pi μ0 represents
1 o pallet with part
2 empty pallet
3 load machine
4 unload machine
5 o transfer device available



Reachability tree analysis

(1,0,0,0,1)
↓ t1
(0,0,1,0,0)
↓ t3
(0,1,0,0,1)
↓ t2
(0,0,0,1,0)
↓ t4

(1,0,0,0,1)
↓ t1
.

The firing vector, σ, for this Petri net repeats
σ = t1,t3,t2,t4, t1..

Matrix equation Analysis

An alternative definition of a Petri net, C, can be given as:
C = (P,T,D-,D+) where
D-[j,i] = #(pi, I(tj)) input arcs matrix to transitions
D+[j,i] = #(pi, O(tj)) output arcs matrix from transitions
are matrices that make up the 'composite change matrix', D,
where D+ - D- = D

```
   pi 1 2 3 4 5          pi 1 2 3 4 5          pi  1 2 3 4 5
     ⎡         ⎤ tj        ⎡         ⎤ tj        ⎡           ⎤tj
     |0 0 1 0 0| 1         |1 0 0 0 1| 1         |-1 0 1 0 -1| 1
D+ = |0 0 0 1 0| 2   D- =  |0 1 0 0 1| 2   D =   | 0 -1 0 1 -1| 2
     |0 1 0 0 1| 3         |0 0 1 0 0| 3         | 0 1 -1 0  1| 3
     |1 0 0 0 1| 4         |0 0 0 1 0| 4         | 1 0 0 -1  1| 4
     ⎣         ⎦           ⎣         ⎦           ⎣           ⎦
```

The firing rule can be expressed for matrices, where the next marking
μ' is given as

$$\mu' = \delta(\mu, tj) = \mu - e[j] \cdot D^- + e[j] \cdot D^+$$
$$= \mu + e[j] \cdot D$$

where e[j] is a vector of size m, with 1 at component j and 0
elsewhere. Here t3 is e[3] = (0,0,1,0).

For a sequence of transition firings, here σ = t1,t3,t2,t4,..

$$\delta(\mu, \sigma) = \delta(\mu, t1,t3,t2,t4,..)$$
$$= \mu + e[j1] \cdot D + e[j2] \cdot D + e[j1] \cdot D + e[j2] \cdot D$$
$$= \mu + f(\sigma) \cdot D$$

With initial marking μ0 = (1,0,0,0,1) t1 is enabled. This leads to

$$\mu1 = \mu0 + e[1] \cdot D$$

$$= (1,0,0,0,1) + (1,0,0,0) \cdot \begin{bmatrix} -1 & 0 & 1 & 0 & -1 \\ 0 & -1 & 0 & 1 & -1 \\ 0 & 1 & -1 & 0 & 1 \\ 1 & 0 & 0 & -1 & 1 \end{bmatrix}$$

$$= (1,0,0,0,1) + (-1,0,1,0,-1)$$
$$= (0,0,1,0,0)$$

which confirms the reachability tree analysis.

It is possible to check if one state marking is reachable from
another. For example, is μ3, (1,0,0,0,1), reachable from μ2,
(0,0,0,1,0) ?

$$\mu3 = \mu2 + f(\sigma) \cdot D$$
$$(1,0,0,0,1) = (0,0,0,1,0) + f(\sigma) \cdot D$$
$$(1,0,0,0,1) - (0,0,0,1,0) = f(\sigma) \cdot D$$
$$(1,0,0,-1,1) = \text{corresponds to t4}$$

Petri net Graphs

Figure A1 Lathe load and start at levels 2 and 3

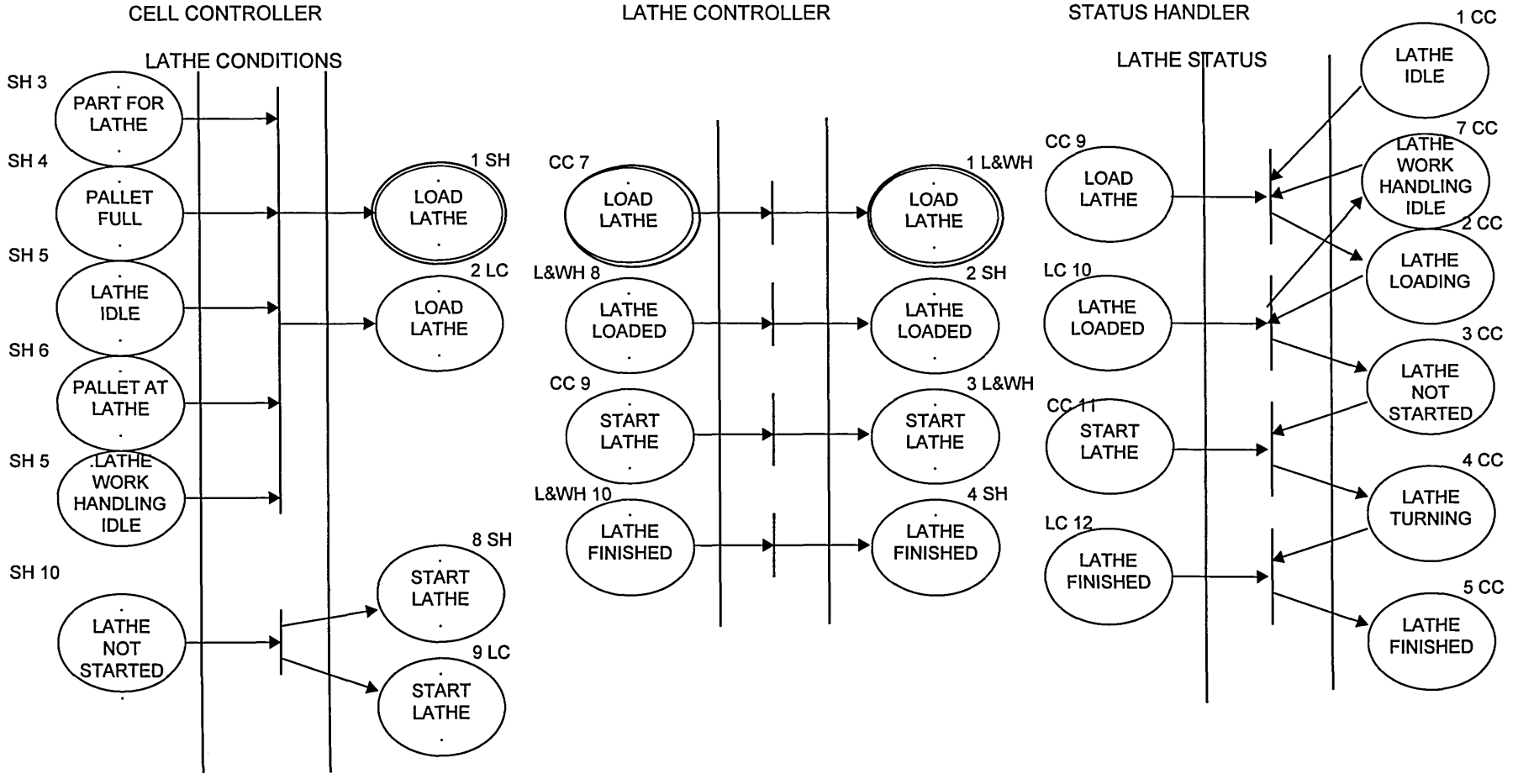
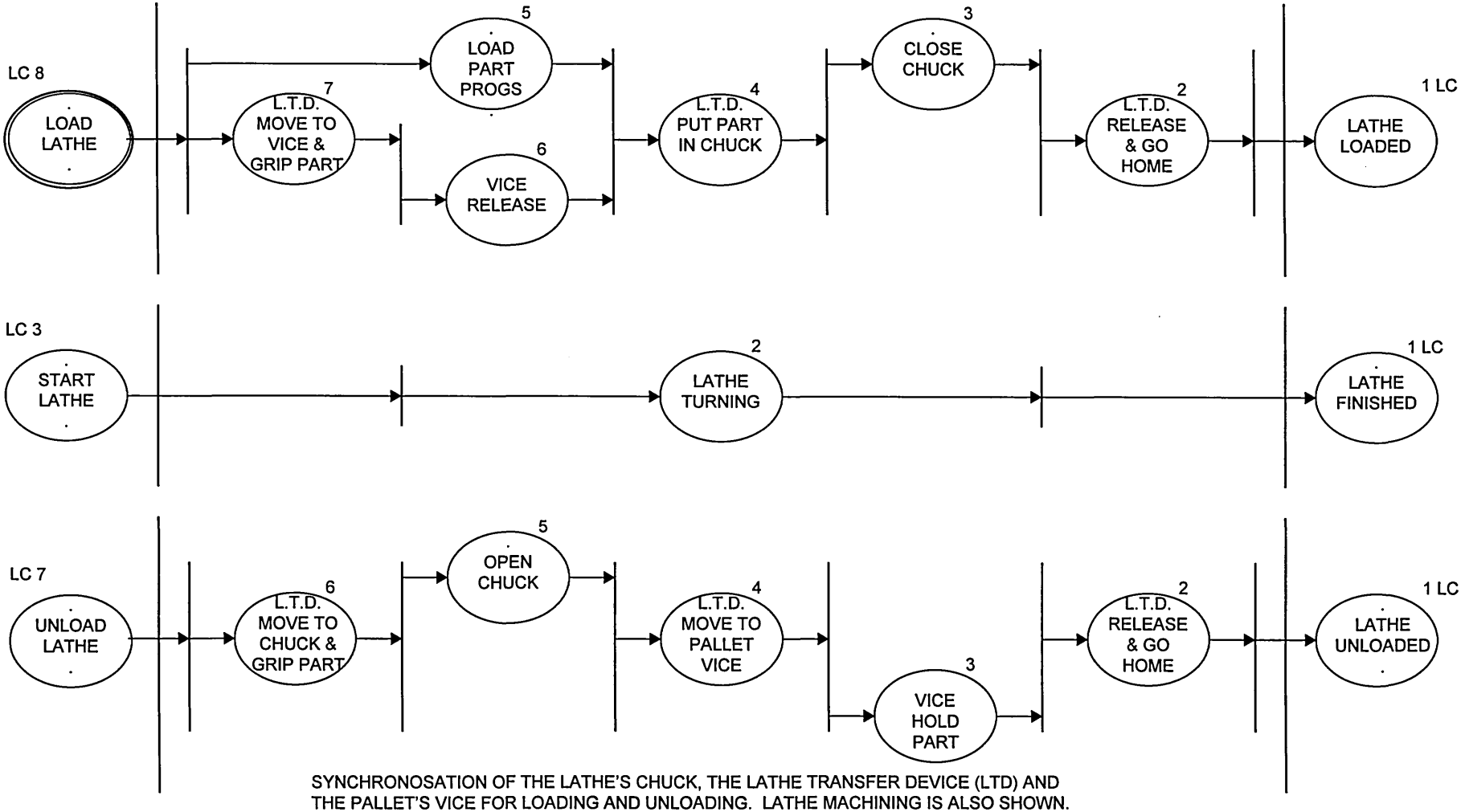


Figure A2 Lathe load, start and unload at level 1



SYNCHRONOSATION OF THE LATHE'S CHUCK, THE LATHE TRANSFER DEVICE (LTD) AND THE PALLET'S VICE FOR LOADING AND UNLOADING. LATHE MACHINING IS ALSO SHOWN.

Boolean Algebra

Logically equivalent propositions:

commutative OR: p OR q = q OR p
&: p & q = q & p

associative OR: (p OR q) OR r = p OR (q OR r)
&: (p & q) & r = p & (q & r)

distributive & over OR: p & (q OR r) = (p & q) OR (p & r)
OR over &: p OR (q & r) = (p OR q) & (p OR r)

de Morgan's laws: ~(p & q) = ~p OR ~q
: ~(p OR q) = ~p & ~q

An earlier version of the Index.wanted Petri net graph is shown in figure 1. By adopting the following abreviations: p=pallet, l=lathe, r=robot, m=miller, then the following Boolean algebra can be done.

The local places 6 to 11, 24 and 25 are made up of the following place numbers

place / description	
6 index wanted	7 OR 8
7 p wanted for l	10 OR 11
8 p for unload r	20&21&22or21&22&23
9 p wanted for m	24 OR 25
10 p wanted for l load	12&13&14OR13&14&15
11 p wanted for l unload	16&17&18OR17&18&19
24 p wanted for m load	26&27&28OR27&28&29
25 p wanted for m unload	30&31&32OR31&32&33

Where numbers 12 to 23 and 26 to 33 are entry places. However, the following entry places are replicated: 12=19=29=33, 15=16=23, 20=26=30

From the above, index wanted, **6**, can be expressed as:
12&13&14 OR 13&14&15 OR
16&17&18 OR 17&18&19 OR
20&21&22 OR 21&22&23 OR
26&27&28 OR 27&28&29 OR
30&31&32 OR 31&32&33

or with abreviated place names and brackets, and by association

6 =
10 p at r & (part for l & l idle) OR
p at m & (part for l & l idle)
OR
11 p at r & (l finished & p l assoc) OR
p at m & (l finished & p l assoc)
OR
24 p at r & (part for m & m idle) OR
p at l & (part for m & m idle)
OR
25 p at r & (m finished & p m assoc) OR
p at l & (m finished & p m assoc)
OR
8 p at l & (part not for l & part not for m) OR
p at m & (part not for l & part not for m)

```
by distribution (& over OR) gives:

10 (p at r OR p at m) & (part for l & l idle)
OR
11 (p at r OR p at m) & (l finished & p l assoc)
OR
24 (p at r OR p at l) & (part for m & m idle)
OR
25 (p at r OR p at l) & (m finished & p m assoc)
OR
8  (p at l OR p at m) & (part not for l & part not for m)

by distribution (& over OR) gives:

7  (p at r OR p at m) & (part for l & l idle) OR
   (l finished & p l assoc)
OR
9  (p at r OR p at l) & (part for m & m idle) OR
   (m finished & p m assoc)
OR
8  (p at l OR p at m) & (part not for l & part not for m)

by distribution (& over OR) gives:

7 (p at r & [(part for l & l idle) OR (l finished & p l assoc)]) OR
  (p at m & [(part for l & l idle) OR (l finished & p l assoc)])
OR
9 (p at r & [(part for m & m idle) OR (m finished & p m assoc)]) OR
  (p at l & [(part for m & m idle) OR (m finished & p m assoc)])
OR
8 (p at l & (part not for l & part not for m)) OR
  (p at m & (part not for l & part not for m))

by distribution (& over OR) gives 6 =:

 (p at r & {[(part for l & l idle) OR (l finished & p l assoc)] OR
            [(part for m & m idle) OR (m finished & p m assoc)]})
OR
 (p at l & {[(part for m & m idle) OR (m finished & p m assoc)] OR
            (part not for l & part not for m)})
OR
 (p at m & {[(part for l & l idle) OR (l finished & p l assoc)] OR
            (part not for l & part not for m)})
```

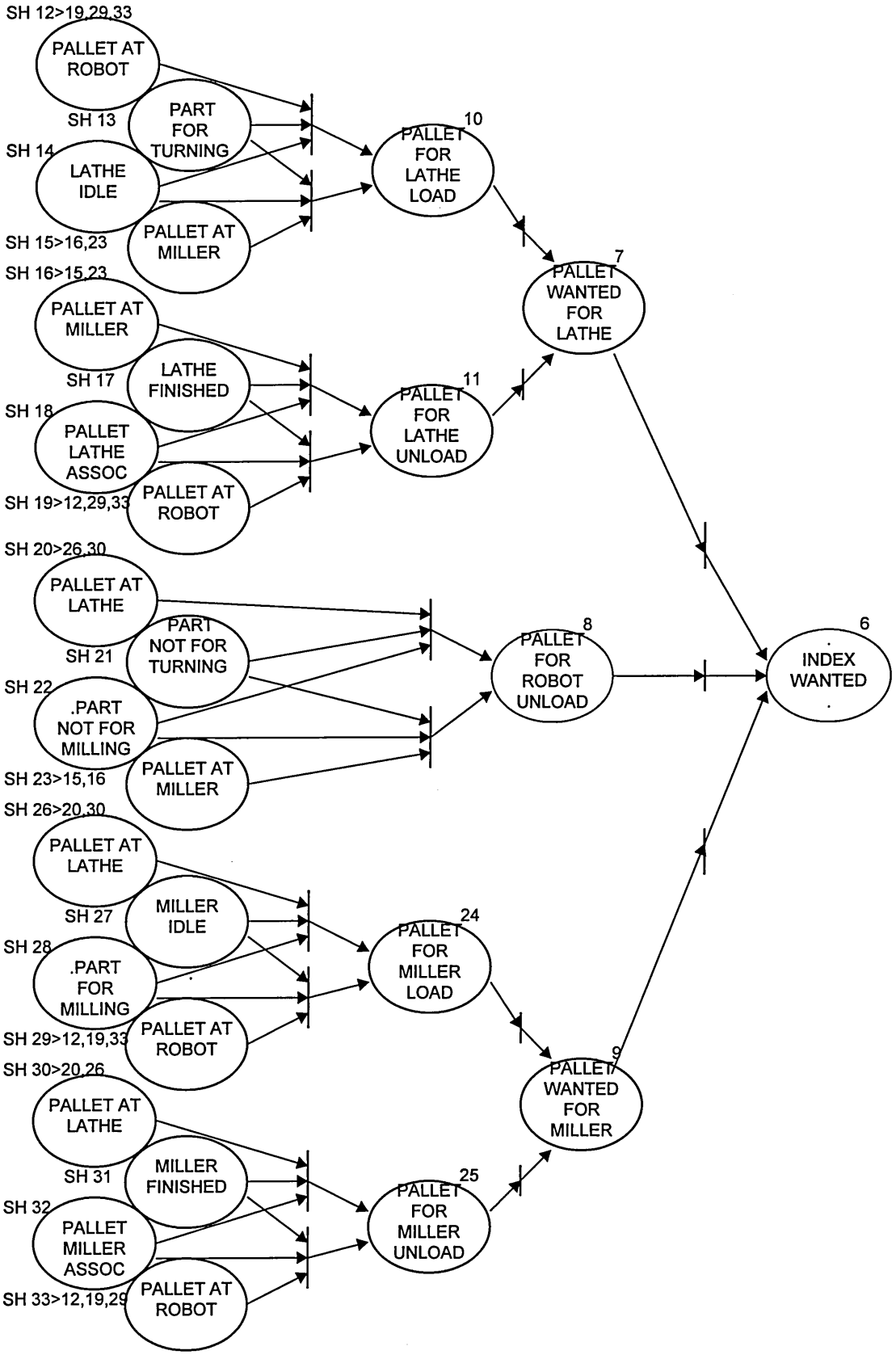


Figure A3 A previous version of index.wanted of the cell controller

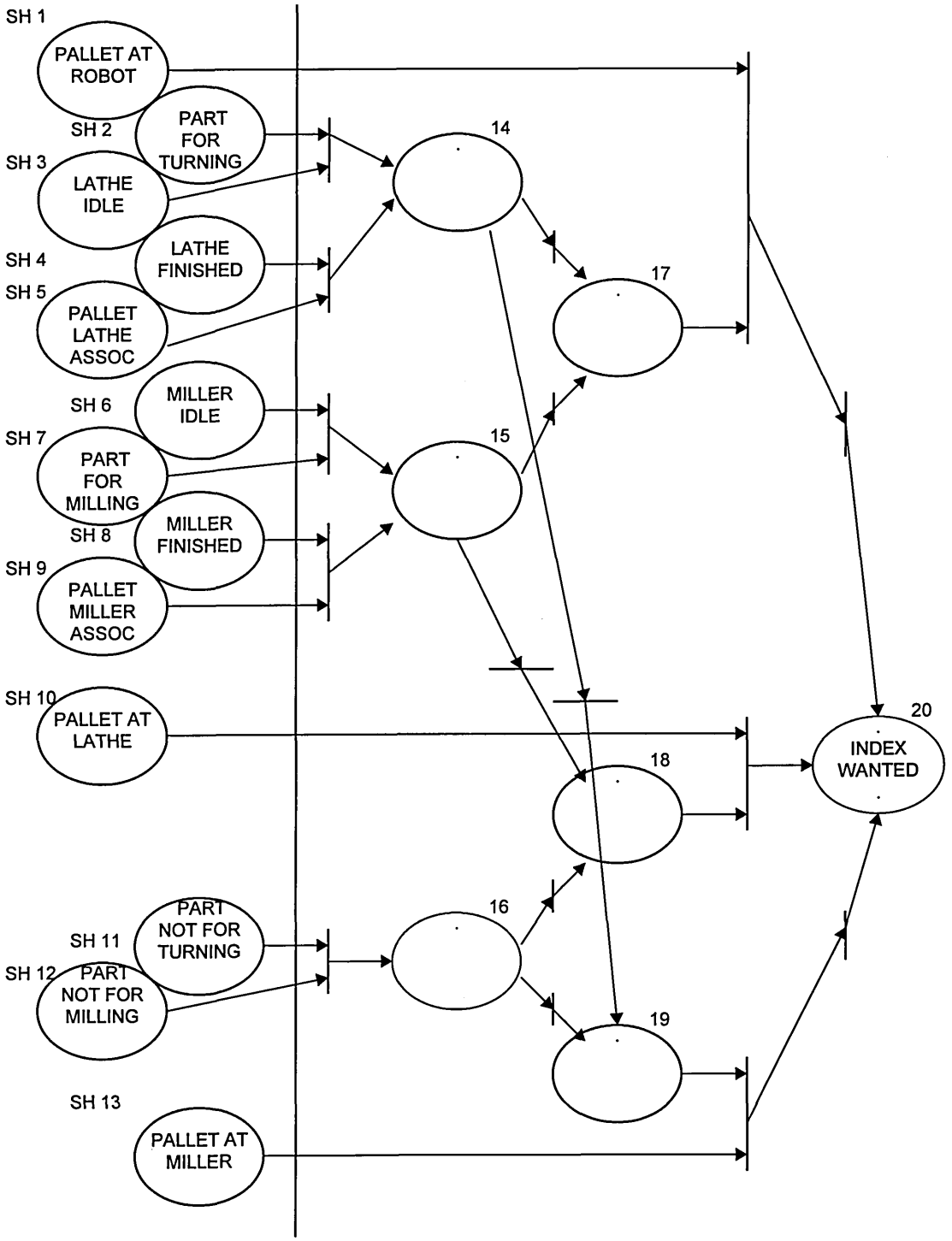


Figure A4 showing the reduction of figure A3 by Boolean algebra

Occam Code

topHd.inc

```
--cell controller to status handler update
--cell controller to status handler- r, l, m and conveyor index
#INCLUDE "CCandSH.inc"
--cell controller to robot, lathe, miller and conveyor controllers
-- and workstation controllers to status handler- r, l, m and conv
#INCLUDE "robotHd.inc"
#INCLUDE "latheHd.inc"
#INCLUDE "millerHd.inc"
#INCLUDE "convHd.inc"
```

top.occ

```
#INCLUDE "hostio.inc"
#INCLUDE "tophd.inc"
PROC program (CHAN OF SP fs,ts, []INT memory)
  #USE "hostio.lib"
  #USE "delay"         -- delay in tenths of second
  #USE "millcon"       -- miller controller
  #USE "convcon"       -- conveyor controller
  #USE "lathecon"      -- lathe controller
  #USE "robotcon"      -- robot controller
  #USE "cellcon"       -- cell controller
  #USE "statHand"      -- status handler
  CHAN OF CC2LC           cc2lc:
  CHAN OF CC2SH.LATHE     cc2sh.lathe:
  CHAN OF CC2MC           cc2mc:
  CHAN OF CC2SH.MILLER    cc2sh.miller:
  CHAN OF CC2RC           cc2rc:
  CHAN OF CC2SH.ROBOT     cc2sh.robot:
  CHAN OF CC2CONV         cc2convc:
  CHAN OF CC2SH.INDEX     cc2sh.index:
  CHAN OF LC2SH           lc2sh:
  CHAN OF MC2SH           mc2sh:
  CHAN OF RC2SH           rc2sh:
  CHAN OF CONV2SH         conv2sh:
  CHAN OF SH2CC.MACHINES sh2cc.machines:
  CHAN OF SH2CC.PALLET    sh2cc.pallet:
  CHAN OF SH2CC.OTHER     sh2cc.other:
  CHAN OF CC2SH.UPDATE    cc2sh:
  SEQ
    SEQ
      SEQ
        PAR
          cell.controller(cc2lc, cc2sh.lathe,
                          cc2mc, cc2sh.miller,
                          cc2rc, cc2sh.robot,
                          cc2convc, cc2sh.index,
                          sh2cc.machines, sh2cc.pallet,
                          sh2cc.other, cc2sh)
          status.handler (fs,ts,
                          lc2sh, cc2sh.lathe,
                          mc2sh, cc2sh.miller,
                          rc2sh, cc2sh.robot,
                          conv2sh, cc2sh.index,
                          sh2cc.machines, sh2cc.pallet,
                          sh2cc.other, cc2sh)
          miller.controller(cc2mc, mc2sh)
          lathe.controller (cc2lc, lc2sh)
          robot.controller (cc2rc, rc2sh)
          conveyor.controller(cc2convc, conv2sh)
    so.exit (fs,ts, sps.success)
:
```

CCandSH.inc

```
--status handler to cell controller feedback
PROTOCOL SH2CC.MACHINES IS BOOL; --lathe idle
                           BOOL; --lathe not started
                           BOOL; --lathe finished
                           BOOL; --lathe work handling idle
                           BOOL; --robot and work handling idle
                           BOOL; --miller idle
                           BOOL; --miller not started
                           BOOL; --miller finished
                           BOOL; --miller work handling idle
                           BOOL: --conveyor idle 10

PROTOCOL SH2CC.PALLET IS   INT;  --number
                           BOOL; --not lathe assoc
                           BOOL; --part not for turning
                           BOOL; --not lathe assoc
                           BOOL; INT::[]BYTE; --for turning & part program
                           BOOL; --empty
                           BOOL; --full
                           BOOL; INT::[]BYTE; --for milling & part program
                           BOOL; --not miller assoc
                           BOOL; --part not for turning
                           BOOL: --not miller assoc     13

PROTOCOL SH2CC.OTHER IS    BOOL; --job.in.parts.list
                           BOOL: --terminate

--cell controller to status handler update request
PROTOCOL CC2SH.UPDATE
  CASE
    update.cc2sh; BOOL
    stop.cc2sh;   BOOL
:
--cell controller to status handler- robot, lathe, miller and conveyor
PROTOCOL CC2SH.ROBOT
  CASE
    load.robot.cc2sh;   BOOL
    unload.robot.cc2sh; BOOL
:
PROTOCOL CC2SH.LATHE
  CASE
    load.lathe.cc2sh;   BOOL
    start.lathe.cc2sh;  BOOL
    unload.lathe.cc2sh; BOOL
:
PROTOCOL CC2SH.MILLER
  CASE
    load.miller.cc2sh;   BOOL
    start.miller.cc2sh;  BOOL
    unload.miller.cc2sh; BOOL
:
PROTOCOL CC2SH.INDEX
  CASE
    start.conv.index.cc2sh;  BOOL
:

VAL No.of.pallets IS 3:
VAL No.of.machines IS 3:
VAL Robot  IS 0:
VAL Miller IS 1:
VAL Lathe  IS 2:
VAL Pallet1 IS 0:
VAL Pallet2 IS 1:
VAL Pallet3 IS 2:
```

latheHd.inc

```
PROTOCOL CC2LC
  CASE
    load.lathe.cc2lc;   BOOL; INT::[]BYTE --for part program
    start.lathe.cc2lc;  BOOL
    unload.lathe.cc2lc; BOOL
    stop.cc2lc;         BOOL
:
PROTOCOL LC2SH
  CASE
    lathe.loaded.lc2sh;   BOOL
    lathe.stopped.lc2sh;  BOOL
    lathe.unloaded.lc2sh; BOOL
:
VAL Lathe.machining.time IS 229:
VAL Lathe.transfer.time IS 15:
```

latheCon.occ

```
#INCLUDE "latheHd.inc"

PROC lathe.controller(CHAN OF CC2LC cc2lc,
                      CHAN OF LC2SH lc2sh)
  #USE "delay"
  BOOL Load.lathe.cc2lc, Start.lathe.cc2lc, Unload.lathe.cc2lc:
  BOOL Lathe.loaded.lc2sh, Lathe.stopped.lc2sh, Lathe.unloaded.lc2sh:
  BOOL Loop, Stop.cc2lc:
  [7]BYTE Code.lathe:
  INT Code.size.lathe:
  SEQ
    Loop:=TRUE
    WHILE Loop
      ALT
        cc2lc ? CASE
          load.lathe.cc2lc; Load.lathe.cc2lc; Code.size.lathe :: Code.lathe
            IF
              Load.lathe.cc2lc
                SEQ
                  --find part program relating to Code.lathe
                  delay(Lathe.transfer.time)
                  lc2sh ! lathe.loaded.lc2sh; Lathe.loaded.lc2sh
          start.lathe.cc2lc; Start.lathe.cc2lc
            IF
              Start.lathe.cc2lc
                SEQ
                  delay(Lathe.machining.time)
                  lc2sh ! lathe.stopped.lc2sh; Lathe.stopped.lc2sh
          unload.lathe.cc2lc;  Unload.lathe.cc2lc
            IF
              Unload.lathe.cc2lc
                SEQ
                  delay(Lathe.transfer.time)
                  lc2sh ! lathe.unloaded.lc2sh; Lathe.unloaded.lc2sh
          stop.cc2lc; Stop.cc2lc
            IF
              Stop.cc2lc
                Loop:=FALSE
:
```

delay.occ

```
PROC delay(VAL INT tenths)
  TIMER clock:
  INT timenow:
  SEQ
    clock ? timenow
    clock ? AFTER timenow PLUS (tenths*1562)
:
```

cellHd.inc

```
--cell controller to status handler update
--cell controller to status handler- r, l, m and conveyor index
#INCLUDE "CCandSH.inc"
--cell controller to robot, lathe, miller and conveyor controllers
-- and workstation controllers to status handler- r, l, m and conv
#INCLUDE "robotHd.inc"
#INCLUDE "latheHd.inc"
#INCLUDE "millerHd.inc"
#INCLUDE "convHd.inc"
```

cellCon.occ

```
#INCLUDE "cellHd.inc"
PROC cell.controller(CHAN OF CC2LC          cc2lc,
                     CHAN OF CC2SH.LATHE    cc2sh.lathe,
                     CHAN OF CC2MC          cc2mc,
                     CHAN OF CC2SH.MILLER   cc2sh.miller,
                     CHAN OF CC2RC          cc2rc,
                     CHAN OF CC2SH.ROBOT    cc2sh.robot,
                     CHAN OF CC2CONV        cc2conv,
                     CHAN OF CC2SH.INDEX    cc2sh.index,
                     CHAN OF SH2CC.MACHINES sh2cc.machines,
                     CHAN OF SH2CC.PALLET   sh2cc.pallet,
                     CHAN OF SH2CC.OTHER    sh2cc.other,
                     CHAN OF CC2SH.UPDATE   cc2sh)
  #USE "delay"
  BOOL LS.idle, LS.not.started, LS.finished, LT.idle:
  BOOL MS.idle, MS.not.started, MS.finished, MT.idle:
  BOOL RS.idle, CS.idle:
  [No.of.machines]INT PS.number:
  [No.of.machines]BOOL PS.not.lathe.assoc,
           PS.part.not.for.turning,
           PS.lathe.assoc,
           PS.part.for.turning,
           PS.full,
           PS.empty,
           PS.part.for.milling,
           PS.miller.assoc,
           PS.part.not.for.milling,
           PS.not.miller.assoc:
  BOOL Job.in.parts.list, Terminate:
  BOOL Load.lathe.cc2lc, Start.lathe.cc2lc, Unload.lathe.cc2lc:
  BOOL Load.miller.cc2mc, Start.miller.cc2mc, Unload.miller.cc2mc:
  BOOL Load.robot.cc2rc, Unload.robot.cc2rc:
  BOOL Start.conv.index.cc2convc:
  BOOL Load.lathe.cc2sh, Start.lathe.cc2sh, Unload.lathe.cc2sh:
  BOOL Load.miller.cc2sh, Start.miller.cc2sh, Unload.miller.cc2sh:
  BOOL Load.robot.cc2sh, Unload.robot.cc2sh:
  BOOL Start.conv.index.cc2sh:
  [No.of.machines]INT Code.size.lathe,Code.size.miller:
  [No.of.machines][7]BYTE Code.lathe,Code.miller:
  BOOL Update:
  BOOL Index.wanted,Loop:
```

```
PROC terminate(VAL BOOL Terminate)
  SEQ
    IF
      Terminate
        SEQ
          PAR
            cc2mc       ! stop.cc2mc; TRUE
            cc2lc       ! stop.cc2lc; TRUE
            cc2rc       ! stop.cc2rc; TRUE
            cc2conv  ! stop.cc2convc; TRUE
            Loop:=FALSE
          cc2sh ! stop.cc2sh; TRUE
      TRUE
        SKIP
:



PROC update.statuses()
  SEQ
    cc2sh ! update.cc2sh; Update
    sh2cc.machines ? LS.idle;LS.not.started;LS.finished;LT.idle;RS.idle;
                     MS.idle;MS.not.started;MS.finished;MT.idle;CS.idle
    SEQ Machine.loop = 0 FOR No.of.machines
      sh2cc.pallet ? PS.number  [Machine.loop];
         PS.not.lathe.assoc     [Machine.loop];
         PS.part.not.for.turning[Machine.loop];
         PS.lathe.assoc         [Machine.loop];
         PS.part.for.turning    [Machine.loop];
         Code.size.lathe[Machine.loop] :: Code.lathe[Machine.loop];
         PS.full                [Machine.loop];
         PS.empty               [Machine.loop];
         PS.part.for.milling    [Machine.loop];
         Code.size.miller[Machine.loop]::Code.miller[Machine.loop];
         PS.miller.assoc        [Machine.loop];
         PS.part.not.for.milling[Machine.loop];
         PS.not.miller.assoc    [Machine.loop]
    sh2cc.other ? Job.in.parts.list;
                  Terminate
:
PROC lathe.conditions()  ------------------
  SEQ
    IF
      PS.part.for.turning[Lathe] AND PS.full[Lathe] AND
         LS.idle AND LT.idle AND CS.idle
        SEQ
          cc2sh.lathe ! load.lathe.cc2sh; Load.lathe.cc2sh
          cc2lc       ! load.lathe.cc2lc; Load.lathe.cc2lc;
                        Code.size.lathe[PS.number[Lathe]]
                           ::Code.lathe[PS.number[Lathe]]
      LS.not.started
        SEQ
          cc2sh.lathe ! start.lathe.cc2sh; Start.lathe.cc2sh
          cc2lc       ! start.lathe.cc2lc; Start.lathe.cc2lc

      PS.lathe.assoc[Lathe] AND LT.idle AND LS.finished AND CS.idle
        SEQ
          cc2sh.lathe ! unload.lathe.cc2sh; Unload.lathe.cc2sh
          cc2lc       ! unload.lathe.cc2lc; Unload.lathe.cc2lc
      TRUE
        SKIP --machining, transfering or error
:
```

```
PROC miller.conditions()
  SEQ
    IF
      PS.part.for.milling[Miller] AND PS.full[Miller] AND
         MT.idle AND MS.idle AND CS.idle
        SEQ
          cc2sh.miller ! load.miller.cc2sh; Load.miller.cc2sh
          cc2mc        ! load.miller.cc2mc; Load.miller.cc2mc;
                         Code.size.miller[PS.number[Miller]]
                            ::Code.miller[PS.number[Miller]]
      MS.not.started
        SEQ
          cc2sh.miller ! start.miller.cc2sh; Start.miller.cc2sh
          cc2mc        ! start.miller.cc2mc; Start.miller.cc2mc

      PS.miller.assoc[Miller] AND MT.idle AND MS.finished AND CS.idle
        SEQ
          cc2sh.miller ! unload.miller.cc2sh; Unload.miller.cc2sh
          cc2mc        ! unload.miller.cc2mc; Unload.miller.cc2mc
      TRUE
        SKIP --machining, transfering or error
:
PROC robot.conditions()
  SEQ
    IF
      PS.not.lathe.assoc[Robot] AND PS.not.miller.assoc[Robot] AND
         PS.empty[Robot] AND RS.idle AND CS.idle AND Job.in.parts.list
        SEQ
          cc2sh.robot ! load.robot.cc2sh; Load.robot.cc2sh
          cc2rc       ! load.robot.cc2rc; Load.robot.cc2rc

      PS.part.not.for.milling[Robot] AND
         PS.part.not.for.turning[Robot] AND
         PS.full[Robot] AND RS.idle AND CS.idle
        SEQ
          cc2sh.robot ! unload.robot.cc2sh; Unload.robot.cc2sh
          cc2rc       ! unload.robot.cc2rc; Unload.robot.cc2rc
      TRUE
        SKIP -- transfering or error
:
PROC index.wanted()
  SEQ
    IF
      ((PS.full[Robot] AND                              --for L load at R
         PS.part.for.turning[Robot] AND LS.idle)  OR

         (PS.full[Miller] AND                             --for L load at M
         PS.part.for.turning[Miller] AND LS.idle) OR

         (LS.finished AND PS.lathe.assoc[Robot])     OR --for L unload at R

         (LS.finished AND PS.lathe.assoc[Miller])     ) --for L unload at M

        SEQ
          Index.wanted:=TRUE
      TRUE
        SEQ
          --Index.wanted:=FALSE
          SKIP --Index not wanted
    IF
      ((PS.full[Lathe] AND                               --for R unload at L
         PS.part.not.for.turning[Lathe] AND
         PS.part.not.for.milling[Lathe])          OR

      (PS.full[Miller] AND                              --for R unload at M
         PS.part.not.for.turning[Miller] AND
         PS.part.not.for.milling[Miller])         OR
```

```
        (PS.empty[Miller] AND                          --for R load at M
           PS.not.lathe.assoc[Miller] AND
           PS.not.miller.assoc[Miller] AND
           Job.in.parts.list)                       OR

        (PS.empty[Lathe] AND                          --for R load at L
           PS.not.lathe.assoc[Lathe] AND
           PS.not.miller.assoc[Lathe] AND
           Job.in.parts.list)                       OR

        (PS.full[Robot] AND                           --for M load at R
           PS.part.for.milling[Robot] AND MS.idle)  OR

        (PS.full[Lathe] AND                           --for M load at L
           PS.part.for.milling[Lathe] AND MS.idle)  OR

        (MS.finished AND PS.miller.assoc[Robot])    OR --for M unload at R

        (MS.finished AND PS.miller.assoc[Lathe])    )  --for M unload at L
          SEQ
            Index.wanted:=TRUE
        TRUE
          SEQ
            --Index.wanted:=FALSE
            SKIP --Index not wanted
:
PROC index.allowed()
  SEQ
    IF
      (LT.idle AND MT.idle AND RS.idle AND Index.wanted) -- AND CS.idle)
        SEQ
          cc2sh.index ! start.conv.index.cc2sh; Start.conv.index.cc2sh
          cc2conv ! start.conv.index.cc2convc;  Start.conv.index.cc2convc
          Index.wanted:=FALSE
      TRUE
        Index.wanted:=FALSE
:
PROC initialise()
  SEQ
    Load.lathe.cc2lc, Start.lathe.cc2lc, Unload.lathe.cc2lc:=TRUE,TRUE,TRUE
    Load.miller.cc2mc, Start.miller.cc2mc,Unload.miller.cc2mc:=TRUE,TRUE,TRUE
    Load.robot.cc2rc, Unload.robot.cc2rc:=TRUE,TRUE
    Start.conv.index.cc2convc:=TRUE
    Load.lathe.cc2sh, Start.lathe.cc2sh, Unload.lathe.cc2sh:=TRUE,TRUE,TRUE
    Load.miller.cc2sh, Start.miller.cc2sh,Unload.miller.cc2sh:=TRUE,TRUE,TRUE
    Load.robot.cc2sh, Unload.robot.cc2sh:=TRUE,TRUE
    Start.conv.index.cc2sh:=TRUE
:
SEQ ------------------------------------------------MAIN PROCEDURE
  Loop:=TRUE
  initialise()
  WHILE Loop
    SEQ
      SEQ
        update.statuses()
        robot.conditions()
        update.statuses()
        lathe.conditions()
        update.statuses()
        miller.conditions()
        update.statuses()
        index.wanted()
        index.allowed()
        terminate(Terminate)
:
```

statHd.inc

```
--cell controller to status handler update
--cell controller to status handler- r, l, m and conveyor index
#INCLUDE "CCandSH.inc"
--cell controller to robot, lathe, miller and conveyor controllers
-- and workstation controllers to status handler- r, l, m and conv
#INCLUDE "robotHd.inc"
#INCLUDE "latheHd.inc"
#INCLUDE "millerHd.inc"
#INCLUDE "convHd.inc"
```

statHand.inc

```
VAL No.of.jobs IS 8:
VAL Chars.per.job IS 19:
VAL Chars.per.code IS 7:
[No.of.machines]INT  PS.number:
[No.of.machines]BOOL PS.not.lathe.assoc,
                     PS.part.not.for.turning,
                     PS.lathe.assoc,
                     PS.part.for.turning,
                     PS.full,
                     PS.empty,
                     PS.part.for.milling,
                     PS.miller.assoc,
                     PS.part.not.for.milling,
                     PS.not.miller.assoc:
BOOL Job.in.parts.list, Parts.list.empty,Terminate:
[No.of.jobs]INT Code.size.lathe,Code.size.miller:
[No.of.jobs][Chars.per.job]BYTE Job.code:
[No.of.jobs][Chars.per.code]BYTE Code.lathe,Code.miller:
BOOL LS.idle, LS.not.started, LS.finished, LT.idle:
BOOL LS.loading, LS.turning, LS.unloading:
BOOL MS.idle, MS.not.started, MS.finished, MT.idle:
BOOL MS.loading, MS.milling, MS.unloading:
BOOL RS.idle, RS.loading, RS.unloading, CS.idle:
BOOL Load.lathe.cc2sh, Start.lathe.cc2sh, Unload.lathe.cc2sh:
BOOL Lathe.loaded.lc2sh, Lathe.stopped.lc2sh, Lathe.unloaded.lc2sh:
BOOL Load.robot.cc2sh, Unload.robot.cc2sh:
BOOL Robot.loaded.rc2sh, Robot.unloaded.rc2sh:
BOOL Load.miller.cc2sh, Start.miller.cc2sh, Unload.miller.cc2sh:
BOOL Miller.loaded.mc2sh, Miller.stopped.mc2sh, Miller.unloaded.mc2sh:
BOOL Start.conv.index.cc2sh, Conv.index.stopped.convc2sh:
BOOL Loop, Update.cc2sh, Stop.cc2sh:
INT Pallet:
INT Index.count,Job.number,Job.at.lathe,Job.at.miller:

BOOL FUNCTION bool.from.char(VAL BYTE Char)
  BOOL TorF:
  VALOF
    IF
      Char = 'F'
        TorF := FALSE
      Char = 'T'
        TorF := TRUE
    RESULT TorF
:
```

```
------------------COMMUNICATION BETWEEN CELL CONTROLLER AND STATUS HANDLER
PROC update.statuses()
  SEQ
    SEQ
      sh2cc.machines ! LS.idle;LS.not.started;LS.finished;LT.idle;RS.idle;
                       MS.idle;MS.not.started;MS.finished;MT.idle;CS.idle
      SEQ Machine.loop = 0 FOR No.of.machines
        sh2cc.pallet ! PS.number  [Machine.loop];
           PS.not.lathe.assoc     [Machine.loop];
           PS.part.not.for.turning[Machine.loop];
           PS.lathe.assoc         [Machine.loop];
           PS.part.for.turning    [Machine.loop];
           Code.size.lathe [Machine.loop]:: Code.lathe[Machine.loop];
           PS.full                [Machine.loop];
           PS.empty               [Machine.loop];
           PS.part.for.milling    [Machine.loop];
           Code.size.miller[Machine.loop]::Code.miller[Machine.loop];
           PS.miller.assoc        [Machine.loop];
           PS.part.not.for.milling[Machine.loop];
           PS.not.miller.assoc    [Machine.loop]
      sh2cc.other ! Job.in.parts.list;
                    Terminate
:
----------------------------------------CHANGES IN PALLET STATUSES
PROC rotate.pallet.statuses()
  SEQ -- pallet moves: to robot from miller, to m from l, and to l from r
    PS.number   [Robot ], PS.number[Miller], PS.number[Lathe ]:=
       PS.number[Miller], PS.number[Lathe ], PS.number[Robot ]
    PS.not.lathe.assoc[Robot ], PS.not.lathe.assoc[Miller],
       PS.not.lathe.assoc[Lathe ]:= PS.not.lathe.assoc  [Miller],
       PS.not.lathe.assoc[Lathe ],  PS.not.lathe.assoc  [Robot ]
    PS.part.not.for.turning   [Robot ], PS.part.not.for.turning[Miller],
       PS.part.not.for.turning[Lathe ]:= PS.part.not.for.turning[Miller],
       PS.part.not.for.turning[Lathe ], PS.part.not.for.turning[Robot ]
    PS.lathe.assoc   [Robot ], PS.lathe.assoc[Miller],
       PS.lathe.assoc[Lathe ]:=PS.lathe.assoc[Miller],
       PS.lathe.assoc[Lathe ], PS.lathe.assoc[Robot ]
    PS.part.for.turning   [Robot ], PS.part.for.turning[Miller],
       PS.part.for.turning[Lathe ]:=PS.part.for.turning[Miller],
       PS.part.for.turning[Lathe ], PS.part.for.turning[Robot ]
    PS.full   [Robot ], PS.full[Miller], PS.full[Lathe ]:=
       PS.full[Miller], PS.full[Lathe ], PS.full[Robot ]
    PS.empty   [Robot ], PS.empty[Miller], PS.empty[Lathe ]:=
       PS.empty[Miller], PS.empty[Lathe ], PS.empty[Robot ]
    PS.part.for.milling   [Robot ], PS.part.for.milling[Miller],
       PS.part.for.milling[Lathe ]:=PS.part.for.milling [Miller],
       PS.part.for.milling[Lathe ], PS.part.for.milling [Robot ]
    PS.miller.assoc   [Robot ], PS.miller.assoc[Miller],
       PS.miller.assoc[Lathe ]:=PS.miller.assoc[Miller],
       PS.miller.assoc[Lathe ], PS.miller.assoc[Robot ]
    PS.part.not.for.milling   [Robot ], PS.part.not.for.milling[Miller],
       PS.part.not.for.milling[Lathe ]:=PS.part.not.for.milling[Miller],
       PS.part.not.for.milling[Lathe ], PS.part.not.for.milling[Robot ]
    PS.not.miller.assoc    [Robot ], PS.not.miller.assoc[Miller],
       PS.not.miller.assoc[Lathe ]:=PS.not.miller.assoc [Miller],
       PS.not.miller.assoc[Lathe ], PS.not.miller.assoc[Robot ]
:
-----------------------------------------------------TERMINATION
PROC job.list.termination(BOOL Terminate)
  SEQ
    IF
      ((Parts.list.empty AND LS.idle AND MS.idle AND
         PS.empty[Miller] AND PS.not.lathe.assoc[Miller] AND
         PS.not.miller.assoc[Miller])                          OR
         (Parts.list.empty AND LS.idle AND MS.idle AND
         PS.empty[Lathe] AND PS.not.lathe.assoc[Lathe] AND
         PS.not.miller.assoc[Lathe]))
```

```
        SEQ
          Terminate := TRUE
      TRUE
        SEQ
          SKIP
:
PROC user.termination(BOOL Terminate)
  BYTE key,result:
  SEQ
    so.pollkey(fs,ts, key, result)
    IF
      key='q'
        SEQ
          so.write.char(fs,ts,key)
          Terminate := TRUE
      TRUE
        SKIP
:
---------------------------------------------------------- SCREEN OUTPUT
PROC print.integer(VAL INT Int)
  [5]BYTE Int.to.screen:
  SEQ
    Int.to.screen:= "     "
    Int.to.screen[3]:= (BYTE (Int+48))
    so.write.string(fs,ts, Int.to.screen)
:
PROC print.bool(VAL BOOL Bool)
  [5]BYTE Message:
  SEQ
    IF
      Bool = TRUE
        Message:= " T   "
      Bool = FALSE
        Message:= " F   "
    so.write.string(fs,ts, Message)
:
PROC show.job.list()
  SEQ
    so.write.string.nl(fs,ts,"Job list")
    SEQ Loop = 0 FOR (SIZE Job.code)
      SEQ
        so.write.string.nl(fs,ts,Job.code[Loop])
:
PROC show.machine.status()
  SEQ
    SEQ
      so.write.string.nl(fs,ts,
        "   idle  lod Nstat mach fin  Ulod T:Idle ")
      so.write.string(fs,ts,"L: ")
      print.bool( LS.idle )
      print.bool( LS.loading )
      print.bool( LS.not.started )
      print.bool( LS.turning )
      print.bool( LS.finished )
      print.bool( LS.unloading )
      print.bool( LT.idle )
      so.write.nl(fs,ts)
      so.write.string(fs,ts,"M: ")
      print.bool( MS.idle )
      print.bool( MS.loading )
      print.bool( MS.not.started )
      print.bool( MS.milling )
      print.bool( MS.finished )
      print.bool( MS.unloading )
      print.bool( MT.idle )
      so.write.nl(fs,ts)
```

```
      so.write.string(fs,ts,    "          C: ")
      print.bool( CS.idle )
      so.write.string(fs,ts,    "             R: ")
      print.bool( RS.idle )
      so.write.nl(fs,ts)
:
PROC show.machine.from.integer(VAL INT number)
  SEQ
    IF
      number = Robot
        so.write.string(fs,ts, "Robot ")
      number = Miller
        so.write.string(fs,ts, "Miller")
      number = Lathe
        so.write.string(fs,ts, "Lathe ")
:
PROC show.pallet.at.RLM()
  SEQ
    so.write.string(fs,ts,
       "  indexed ---------------------------- ")
    so.write.string(fs,ts,"R:")
    print.integer(PS.number[Robot])
    so.write.string(fs,ts,"M:")
    print.integer(PS.number[Miller])
    so.write.string(fs,ts,"L:")
    print.integer(PS.number[Lathe])
    IF
      LS.idle
        SEQ
          so.write.string(fs,ts,"l:")
          print.integer(-2)
      TRUE
        SEQ
          so.write.string(fs,ts,"l:")
          print.integer(Job.at.lathe)
    IF
      MS.idle
        SEQ
          so.write.string(fs,ts,"m:")
          print.integer(-2)
      TRUE
        SEQ
          so.write.string(fs,ts,"m:")
          print.integer(Job.at.miller)
    so.write.nl(fs,ts)
:
PROC show.pallet.status()
  SEQ
    SEQ
      so.write.string.nl(fs,ts,
         "Mach   pal# full 4L   4M   N4L  N4M  empt Lass Mass NLassNMass")
      SEQ Machine.loop = 0 FOR No.of.pallets
        SEQ
          show.machine.from.integer          (Machine.loop)
          print.integer(PS.number            [Machine.loop])
          print.bool( PS.full                [Machine.loop])
          print.bool( PS.part.for.turning    [Machine.loop])
          print.bool( PS.part.for.milling    [Machine.loop])
          print.bool( PS.part.not.for.turning[Machine.loop])
          print.bool( PS.part.not.for.milling[Machine.loop])
          print.bool( PS.empty               [Machine.loop])
          print.bool( PS.lathe.assoc         [Machine.loop])
          print.bool( PS.miller.assoc        [Machine.loop])
          print.bool( PS.not.lathe.assoc     [Machine.loop])
          print.bool( PS.not.miller.assoc    [Machine.loop])
          so.write.nl(fs,ts)
:
```

```
-------------------------------------------------- JOB RELATED
PROC get.job.info(VAL INT Job.number)
  SEQ
    PS.part.not.for.turning[Robot]:=bool.from.char(Job.code[Job.number][1])
    PS.part.for.turning    [Robot]:=bool.from.char(Job.code[Job.number][2])
    Code.lathe             [Robot]:=[Job.code[Job.number] FROM 3 FOR 7]
    PS.part.not.for.milling[Robot]:=bool.from.char(Job.code[Job.number][10])
    PS.part.for.milling    [Robot]:=bool.from.char(Job.code[Job.number][11])
    Code.miller            [Robot]:=[Job.code[Job.number] FROM 12 FOR 7]
    Job.in.parts.list             :=bool.from.char(Job.code[Job.number][0])
:
PROC get.next.job.or.stop()
  SEQ
    SEQ
      get.job.info(Job.number)
      print.integer(Job.number)
      so.write.string(fs,ts,Code.lathe[Robot])
      so.write.string(fs,ts,Code.miller[Robot])
      Job.number := Job.number + 1
      IF
        (Job.number = No.of.jobs) --for a full parts list
          SEQ
            Parts.list.empty := TRUE
            Job.in.parts.list:= FALSE
            so.write.string(fs,ts," End of job list ")
        TRUE
          IF                      --if parts list is not full
            (NOT bool.from.char(Job.code[Job.number][0]))
              SEQ
                Parts.list.empty := TRUE
                Job.in.parts.list:= FALSE
                so.write.string(fs,ts," End of job list ")
            TRUE
              SKIP
:
PROC initial.conditions() ------------------------------------------

  PROC lathe.statuses()
    SEQ
      LS.idle              :=TRUE
      LS.loading           :=FALSE
      LS.not.started       :=FALSE
      LS.turning           :=FALSE
      LS.finished          :=FALSE
      LS.unloading         :=FALSE
      LT.idle              :=TRUE
  :
  PROC miller.statuses()
    SEQ
      MS.idle              :=TRUE
      MS.loading           :=FALSE
      MS.not.started       :=FALSE
      MS.milling           :=FALSE
      MS.finished          :=FALSE
      MS.unloading         :=FALSE
      MT.idle              :=TRUE
  :
  BOOL FUNCTION eof(VAL BYTE Result)                          --129
    BOOL End.of.file:
    VALOF
      IF
        Result = spr.ok              --end of file reached      0
          End.of.file := TRUE
        Result = spr.operation.failed --end of file not reached  128
          End.of.file := FALSE
        Result = 129 (BYTE)   --end of file not reached  128
          End.of.file := FALSE
```

```
    RESULT End.of.file
:
PROC read.job.list()
  INT32 File.id:
  BYTE Result,Eof.result:
  INT Len.of.data, Job.loop:
  [Chars.per.job]BYTE Data:
  SEQ
    so.open(fs,ts, "JOBLIST.OCC", spt.text, spm.input, File.id, Result)
    Job.loop:=0
    Eof.result:=129 (BYTE) --not end of file
    WHILE NOT eof(Eof.result) --eof marker must be at the end of
      SEQ                     --the final component line
        so.gets(fs,ts, File.id, Len.of.data, Data, Result) --read
        so.write.string.nl (fs,ts, Data)
        Job.code[Job.loop] := Data
        so.eof(fs,ts,File.id, Eof.result)
        Job.loop:=Job.loop + 1
    so.close(fs,ts,File.id,Result)
:
PROC job.statuses()
  SEQ
    SEQ Job.loop = 0 FOR No.of.jobs
      SEQ
        Job.code    [Job.loop] := "FTF-------TF-------"
        Code.lathe  [Job.loop] := [Job.code[Job.loop] FROM 3 FOR 7]
        Code.miller[Job.loop] := [Job.code[Job.loop] FROM 12 FOR 7]
        Code.size.lathe [Job.loop]  := 7
        Code.size.miller[Job.loop]  := 7
    --parts list
    --Job.code[0]:="TFTLK(4,W)FTMK(4,W)"
    --Job.code[1]:="TFTLP(3,B)TFM-(-,-)"
    --Job.code[2]:="TFTLR(2,W)FTMR(2,W)"
    --Job.code[3]:="TFTLR(2,W)FTMR(2,W)"
    --Job.code[4]:="TFTLR(2,W)FTMR(2,W)"
    --Code.lathe   := ["LK(4,W)","LP(3,B)", "LR(2,W)", "LR(2,W)"]
    --Code.miller  := ["MK(4,W)","MP(0,0)", "MR(2,W)", "MR(2,W)"]
    --Code.size.lathe   := [7,7,7,7,7,7,7,7]
    --Code.size.miller  := [7,7,7,7,7,7,7,7]
    Job.in.parts.list:=TRUE
    Parts.list.empty:=FALSE
    Job.at.lathe,Job.at.miller:=-1,-1
:
PROC pallet.statuses()
  SEQ
    SEQ Loop = 0 FOR No.of.pallets
      SEQ
        PS.number               [Loop]:=0
        PS.full                 [Loop]:=FALSE
        PS.part.for.turning     [Loop]:=TRUE
        PS.part.for.milling     [Loop]:=FALSE
        PS.part.not.for.turning[Loop]:=FALSE
        PS.part.not.for.milling[Loop]:=TRUE
        PS.empty                [Loop]:=TRUE
        PS.lathe.assoc          [Loop]:=FALSE
        PS.miller.assoc         [Loop]:=FALSE
        PS.not.lathe.assoc      [Loop]:=TRUE
        PS.not.miller.assoc     [Loop]:=TRUE
:
PROC other.statuses()
  SEQ
    RS.idle              :=TRUE
    CS.idle              :=TRUE
    Pallet               :=0
    PS.number          [Robot ]:=Robot
    PS.number          [Miller]:=Miller
    PS.number          [Lathe ]:=Lathe
```

```
      Terminate:=FALSE
      Index.count:=0
      Job.number :=0
  :
  SEQ
    job.statuses()
    read.job.list()
    pallet.statuses()
    lathe.statuses()
    miller.statuses()
    other.statuses()
:
```

statHand.occ

```
#INCLUDE "hostio.inc"
#INCLUDE "statHd.inc"
PROC status.handler(CHAN OF SP             fs, ts,
                    CHAN OF LC2SH          lc2sh,
                    CHAN OF CC2SH.LATHE    cc2sh.lathe,
                    CHAN OF MC2SH          mc2sh,
                    CHAN OF CC2SH.MILLER   cc2sh.miller,
                    CHAN OF RC2SH          rc2sh,
                    CHAN OF CC2SH.ROBOT    cc2sh.robot,
                    CHAN OF CONV2SH        conv2sh,
                    CHAN OF CC2SH.INDEX    cc2sh.index,
                    CHAN OF SH2CC.MACHINES sh2cc.machines,
                    CHAN OF SH2CC.PALLET   sh2cc.pallet,
                    CHAN OF SH2CC.OTHER    sh2cc.other,
                    CHAN OF CC2SH.UPDATE   cc2sh)
  #USE "hostio.lib"
  #INCLUDE "stathand.inc" --channel definitions,
  --        initial.conditions, update.statuses,
  --        show.machine.status, show.pallet.status,
  --        get.next.job.or.stop, show.job.list
  --        rotate.pallet.statuses, print.integer, print.bool
  SEQ ------------------------------------------------ MAIN PROCEDURE
    initial.conditions()
    so.write.string(fs,ts,"Not")
    show.pallet.at.RLM() --prints R:0 L:2 M:1
    Loop:=TRUE
    WHILE Loop
      SEQ
        user.termination(Terminate)
        ALT
          cc2sh.lathe   ?  CASE ------------------------ L A T H E
            load.lathe.cc2sh; Load.lathe.cc2sh
              SEQ
                print.integer(PS.number[Lathe])
                so.write.string(fs,ts,"l  lod  ")
                LS.idle                :=FALSE
                LT.idle                :=FALSE
                PS.full[Lathe ]              :=FALSE
                PS.not.lathe.assoc[Lathe ]   :=FALSE
                PS.part.for.turning[Lathe ]  :=FALSE
                Job.at.lathe:= PS.number[Lathe]
                PS.empty[Lathe ]             :=TRUE
                PS.lathe.assoc[Lathe ]       :=TRUE
                LS.loading             :=TRUE
            start.lathe.cc2sh; Start.lathe.cc2sh
              SEQ
                so.write.string(fs,ts,"l start ")
                LS.not.started         :=FALSE
                LS.turning             :=TRUE
            unload.lathe.cc2sh; Unload.lathe.cc2sh
              SEQ
                print.integer(PS.number[Lathe])
                so.write.string(fs,ts,"l ulod  ")
                LS.finished            :=FALSE
                LT.idle                :=FALSE
                PS.empty[Lathe ]               :=FALSE
                PS.lathe.assoc[Lathe ]         :=FALSE
                PS.part.not.for.turning[Lathe ]:=TRUE
                PS.not.lathe.assoc[Lathe ]     :=TRUE
                PS.full[Lathe ]                :=TRUE
                LS.unloading           :=TRUE
```

```
lc2sh ? CASE
  lathe.loaded.lc2sh; Lathe.loaded.lc2sh
    SEQ
      print.integer(PS.number[Lathe])
      so.write.string(fs,ts,"l  lodd ")
      LS.loading              :=FALSE
      LS.not.started          :=TRUE
      LT.idle                 :=TRUE
  lathe.stopped.lc2sh; Lathe.stopped.lc2sh
    SEQ
      so.write.string(fs,ts,"l  fins ")
      LS.turning              :=FALSE
      LS.finished             :=TRUE
  lathe.unloaded.lc2sh; Lathe.unloaded.lc2sh
    SEQ
      print.integer(PS.number[Lathe])
      so.write.string(fs,ts,"l ulodd ")
      LS.unloading            :=FALSE
      LS.idle                 :=TRUE
      LT.idle                 :=TRUE
cc2sh.robot ? CASE  ---------------------------- R O B O T
  load.robot.cc2sh; Load.robot.cc2sh
    SEQ
      print.integer(PS.number[Robot])
      so.write.string(fs,ts,"r  lod  ")
      RS.idle                 :=FALSE
      PS.empty[Robot ]                :=FALSE
      PS.full[Robot ]                 :=TRUE
      get.next.job.or.stop()
  unload.robot.cc2sh; Unload.robot.cc2sh
    SEQ
      print.integer(PS.number[Robot])
      so.write.string(fs,ts,"r ulod  ")
      RS.idle                 :=FALSE
      PS.full[Robot ]                  :=FALSE
      PS.empty[Robot ]                 :=TRUE
rc2sh ?  CASE
  robot.loaded.rc2sh; Robot.loaded.rc2sh
    SEQ
      print.integer(PS.number[Robot])
      so.write.string(fs,ts,"r  lodd ")
      RS.idle                :=TRUE
  robot.unloaded.rc2sh; Robot.unloaded.rc2sh
    SEQ
      print.integer(PS.number[Robot])
      so.write.string(fs,ts,"r ULODD ")
      RS.idle                 :=TRUE
      job.list.termination(Terminate)
cc2sh.miller ?  CASE ------------------------- M I L L E R
  load.miller.cc2sh; Load.miller.cc2sh
    SEQ
      print.integer(PS.number[Miller])
      so.write.string(fs,ts,"m  lod  ")
      MS.idle                 :=FALSE
      MT.idle                 :=FALSE
      PS.full[Miller]                 :=FALSE
      PS.not.miller.assoc[Miller]     :=FALSE
      PS.part.for.milling[Miller]     :=FALSE
      Job.at.miller:=PS.number[Miller]
      PS.empty[Miller]                :=TRUE
      PS.miller.assoc[Miller]         :=TRUE
      MS.loading              :=TRUE
  start.miller.cc2sh; Start.miller.cc2sh
    SEQ
      so.write.string(fs,ts,"m start ")
      MS.not.started          :=FALSE
      MS.milling              :=TRUE
```

```
    unload.miller.cc2sh; Unload.miller.cc2sh
      SEQ
        print.integer(PS.number[Miller])
        so.write.string(fs,ts,"m ulod  ")
        MS.finished              :=FALSE
        MT.idle                  :=FALSE
        PS.empty[Miller]                 :=FALSE
        PS.miller.assoc[Miller]          :=FALSE
        PS.part.not.for.milling[Miller]:=TRUE
        PS.not.miller.assoc[Miller]      :=TRUE
        PS.full[Miller]                  :=TRUE
        MS.unloading             :=TRUE
  mc2sh  ?  CASE
    miller.loaded.mc2sh; Miller.loaded.mc2sh
      SEQ
        print.integer(PS.number[Miller])
        so.write.string(fs,ts,"m  lodd ")
        MS.loading               :=FALSE
        MS.not.started           :=TRUE
        MT.idle                  :=TRUE
    miller.stopped.mc2sh; Miller.stopped.mc2sh
      SEQ
        so.write.string(fs,ts,"m  fins ")
        MS.milling               :=FALSE
        MS.finished              :=TRUE
    miller.unloaded.mc2sh; Miller.unloaded.mc2sh
      SEQ
        print.integer(PS.number[Miller])
        so.write.string(fs,ts,"m ulodd ")
        MS.unloading             :=FALSE
        MS.idle                  :=TRUE
        MT.idle                  :=TRUE
  cc2sh.index ? CASE  ------------------------ INDEX STARTED
    start.conv.index.cc2sh; Start.conv.index.cc2sh
      SEQ
        CS.idle                  :=FALSE
  conv2sh ? CASE  ------------------------------- INDEX END
    conv.index.stopped.convc2sh; Conv.index.stopped.convc2sh
      SEQ
        SEQ
          CS.idle                :=TRUE
          rotate.pallet.statuses()
          so.write.nl(fs,ts)
          Index.count := Index.count + 1
          so.write.int(fs,ts, Index.count, 3)
          show.pallet.at.RLM() --prints R:0 L:2 M:1
          --show.machine.status() --inspect statuses
          --show.pallet.status()  --on index
  cc2sh ? CASE  -------------------- U P D A T E
    update.cc2sh; Update.cc2sh
      SEQ
        --so.write.char(fs,ts,'.') --shows update frequency
        update.statuses()
    stop.cc2sh; Stop.cc2sh
      IF
        Stop.cc2sh
          SEQ
            Loop:=FALSE
            so.write.nl(fs,ts)
            so.write.string.nl(fs,ts,"Terminated")
        TRUE
          SKIP
:
```

millerHd.inc

```
PROTOCOL CC2MC
  CASE
    load.miller.cc2mc;   BOOL; INT::[]BYTE --for part program
    start.miller.cc2mc;  BOOL
    unload.miller.cc2mc; BOOL
    stop.cc2mc;          BOOL
:
PROTOCOL MC2SH
  CASE
    miller.loaded.mc2sh;   BOOL
    miller.stopped.mc2sh;  BOOL
    miller.unloaded.mc2sh; BOOL
:
VAL Miller.machining.time IS 141:
VAL Miller.transfer.time IS 13:
```

millCon.occ

```
#INCLUDE "millerHd.inc"

PROC miller.controller(CHAN OF CC2MC cc2mc,
                       CHAN OF MC2SH mc2sh)
  #USE "delay"
  BOOL Load.miller.cc2mc, Start.miller.cc2mc, Unload.miller.cc2mc:
  BOOL Miller.loaded.mc2sh, Miller.stopped.mc2sh, Miller.unloaded.mc2sh:
  BOOL Loop, Stop.miller.cc2mc:
  [7]BYTE Code.miller:
  INT Code.size.miller:
  SEQ
    Loop:=TRUE
    WHILE Loop
      ALT
        cc2mc ? CASE
          load.miller.cc2mc; Load.miller.cc2mc; Code.size.miller :: Code.miller
            IF
              Load.miller.cc2mc
                SEQ
                  --find part program relating to Code.miller
                  delay(Miller.transfer.time)
                  mc2sh ! miller.loaded.mc2sh; Miller.loaded.mc2sh
              TRUE
                SKIP
          start.miller.cc2mc; Start.miller.cc2mc
            IF
              Start.miller.cc2mc
                SEQ
                  delay(Miller.machining.time)
                  mc2sh ! miller.stopped.mc2sh; Miller.stopped.mc2sh
          unload.miller.cc2mc; Unload.miller.cc2mc
            IF
              Unload.miller.cc2mc
                SEQ
                  delay(Miller.transfer.time)
                  mc2sh ! miller.unloaded.mc2sh; Miller.unloaded.mc2sh
              TRUE
                SKIP
          stop.cc2mc; Stop.miller.cc2mc
            IF
              Stop.miller.cc2mc
                Loop:=FALSE
:
```

robotHd.inc

```
PROTOCOL CC2RC
  CASE
    load.robot.cc2rc;   BOOL
    unload.robot.cc2rc; BOOL
    stop.cc2rc;         BOOL
:
PROTOCOL RC2SH
  CASE
    robot.loaded.rc2sh;   BOOL
    robot.unloaded.rc2sh; BOOL
:
VAL Robot.transfer.time IS 5:
```

robotCon.occ

```
#INCLUDE "robotHd.inc"

PROC robot.controller(CHAN OF CC2RC cc2rc,
                      CHAN OF RC2SH rc2sh)
  #USE "delay"
  BOOL Load.robot.cc2rc, Unload.robot.cc2rc:
  BOOL Robot.loaded.rc2sh, Robot.unloaded.rc2sh:
  BOOL Loop, Stop.robot.cc2rc:
  SEQ
    Loop:=TRUE
    WHILE Loop
      ALT
        cc2rc ? CASE
          load.robot.cc2rc; Load.robot.cc2rc
            IF
              Load.robot.cc2rc
                SEQ
                  delay(Robot.transfer.time)
                  rc2sh ! robot.loaded.rc2sh; Robot.loaded.rc2sh
              TRUE
                SKIP
          unload.robot.cc2rc; Unload.robot.cc2rc
            IF
              Unload.robot.cc2rc
                SEQ
                  delay(Robot.transfer.time)
                  rc2sh ! robot.unloaded.rc2sh; Robot.unloaded.rc2sh
              TRUE
                SKIP
          stop.cc2rc; Stop.robot.cc2rc
            IF
              Stop.robot.cc2rc
                Loop:=FALSE
:
```

convHd.inc

```
PROTOCOL CC2CONV
  CASE
    start.conv.index.cc2convc;    BOOL
    stop.cc2convc;                BOOL
:
PROTOCOL CONV2SH
  CASE
    conv.index.stopped.convc2sh; BOOL
:
VAL Index.time IS 41:
```

convCon.occ

```
#INCLUDE "convhd.inc"

PROC conveyor.controller(CHAN OF CC2CONV cc2convc,
                         CHAN OF CONV2SH convc2sh)
  #USE "delay"
  BOOL Start.conv.index.cc2convc, Conv.index.stopped.convc2sh:
  BOOL Loop, Stop.cc2convc:
  SEQ
    Loop:=TRUE
    WHILE Loop
      ALT
        cc2convc ? CASE
          start.conv.index.cc2convc; Start.conv.index.cc2convc
            IF
              Start.conv.index.cc2convc
                SEQ
                  delay(Index.time)
                  convc2sh ! conv.index.stopped.convc2sh;
Conv.index.stopped.convc2sh
              TRUE
                SKIP
          stop.cc2convc; Stop.cc2convc
            IF
              Stop.cc2convc
                Loop:=FALSE
:
```

OUTPUT

```
Booting root transputer...ok
TFTLK(4,W)FTMK(4,W)
TFTLP(3,B)TFM-(-,-)
TFTLK(4,W)FTMK(4,W)
TFTLR(2,W)FTMR(2,W)
Not  indexed ---------------------------- R:   0 M:   1 L:   2 l:   . m:   .
   0 r  lod     0 LK(4,W)MK(4,W)   0 r  lodd
  1  indexed ---------------------------- R:   1 M:   2 L:   0 l:   . m:   .
   1 r  lod     1 LP(3,B)M-(-,-)   0 l  lod     1 r  lodd    0 l  lodd
  2  indexed ---------------------------- R:   2 M:   0 L:   1 l:   0 m:   .
   2 r  lod     2 LK(4,W)MK(4,W)l start    2 r  lodd
  3  indexed ---------------------------- R:   0 M:   1 L:   2 l:   0 m:   .

  4  indexed ---------------------------- R:   1 M:   2 L:   0 l:   0 m:   .
   2 m  lod     2 m  lodd m start
  5  indexed ---------------------------- R:   2 M:   0 L:   1 l:   0 m:   2
l  fins m  fins
  6  indexed ---------------------------- R:   0 M:   1 L:   2 l:   0 m:   2

  7  indexed ---------------------------- R:   1 M:   2 L:   0 l:   0 m:   2
   0 l ulod     2 m ulod     2 m ulodd    0 l ulodd
  8  indexed ---------------------------- R:   2 M:   0 L:   1 l:   . m:   .
   1 l  lod     0 m  lod     0 m  lodd m start    1 l  lodd l start
  9  indexed ---------------------------- R:   0 M:   1 L:   2 l:   1 m:   0
m  fins
 10  indexed ---------------------------- R:   1 M:   2 L:   0 l:   1 m:   0

 11  indexed ---------------------------- R:   2 M:   0 L:   1 l:   1 m:   0
   0 m ulod  l  fins    1 l ulod     0 m ulodd    1 l ulodd
 12  indexed ---------------------------- R:   0 M:   1 L:   2 l:   . m:   .
   0 r ulod     2 l  lod     0 r ULODD    0 r  lod     3 LR(2,W)MR(2,W) End of
job list    0 r  lodd    2 l  lodd
 13  indexed ---------------------------- R:   1 M:   2 L:   0 l:   2 m:   .
   1 r ulod  l start    1 r ULODD
 14  indexed ---------------------------- R:   2 M:   0 L:   1 l:   2 m:   .
   0 m  lod     0 m  lodd m start
 15  indexed ---------------------------- R:   0 M:   1 L:   2 l:   2 m:   0
m  fins l  fins
 16  indexed ---------------------------- R:   1 M:   2 L:   0 l:   2 m:   0

 17  indexed ---------------------------- R:   2 M:   0 L:   1 l:   2 m:   0
   0 m ulod     0 m ulodd
 18  indexed ---------------------------- R:   0 M:   1 L:   2 l:   2 m:   .
   2 l ulod     2 l ulodd
 19  indexed ---------------------------- R:   1 M:   2 L:   0 l:   . m:   .
   0 l  lod     0 l  lodd
 20  indexed ---------------------------- R:   2 M:   0 L:   1 l:   0 m:   .
   2 r ulod  l start    2 r ULODD
 21  indexed ---------------------------- R:   0 M:   1 L:   2 l:   0 m:   .
l  fins
 22  indexed ---------------------------- R:   1 M:   2 L:   0 l:   0 m:   .
   0 l ulod     0 l ulodd
 23  indexed ---------------------------- R:   2 M:   0 L:   1 l:   . m:   .

 24  indexed ---------------------------- R:   0 M:   1 L:   2 l:   . m:   .
   0 r ulod     0 r ULODD
 25  indexed ---------------------------- R:   1 M:   2 L:   0 l:   . m:   .

Terminated
```

References

LNCS = Lecture Notes in Computer Science by Springer Verlag

00-55 1991, The Procurement of Safety Critical Software in Defence Equipment, Interim Defence Standard 00-55 UK MoD.

00-56 1991, Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment, Interim Defence Standard 00-56 UK MoD.

9Tiles, NineTiles Computer Systems Ltd, Waterbeach, Cambridge CB5 9HW UK.

Alford 1982. Alford MW, Ansart, Hommel, Lamport, Liskov, Mullery and Schneider, Distributed Systems - Methods and Tools for Specification, LNCS190.

Anderson 1981 Anderson T and Lee, Fault Tolerance - Principles and Practice, Prentice Hall.

Armitage 1988 Armitage B, Dunlop, Hutchinson and Yu, Fieldbus- an Emerging Communications Standard, Microprocessors and Microsystems v12 Dec pp555-562.

Austin 1993 Austin S and Parkin G, Formal Methods- a Survey, National Physical Laboratory Teddington Middlesex UK.

Balbo 1992 Balbo G, Performance Issues in Parallel Programming, in Jenson K (ed.) Application and Theory in Petri Nets '92, LNCS616 ISBN 0-387-55676-1 p1-23.

Banaszak 1990 Banaszak ZA and Krogh, Deadlock Avoidance in FMSs with Concurrently Competing Flows, IEEE Trans Robotics and Automation v6 n6 pp724-734.

Barns 1988 Barns A; Programming in occam 2; Addison- Wesley; ISBN 0-201-17371-9.

Ben-Ari 1990 Ben-Ari M, Principles of Concurrent and Distributed Programming, Prentice Hall ISBN 0-13-711821-X.

Best 1991 Best E, Overview of the Results of the ESPRIT Basic Research Action DEMON, PNPM91 IEEE TH0386-3 p224-235.

Birkinshaw 1994 Birkinshaw CI , Croll, Marriott and Nixon, Engineering Safety-Related Parallel Systems, Information and Software Technology v36 n7 p449-456.

Breant 1991 Breant F, Rapid Prototyping from Petri Net on a Loosely Coupled Parallel Architecture, Proc 3rd Int. Conf Applications of Transputers IOS Press Aug p644-649.

Browne 1985 Browne J, Duboios, Rathmill, Sethi and Steke, Classification of FMSs, in Bignell V et al, Manufacturing Systems- context applications and techniques, Blackwell ISBN 0-631-14378-5, p111-119.

BS 7000:1989, Guide to Managing Product Design.

BS-ISO 7498:1984, Information Processing Systems- Open Systems Interconnection - Basic Reference Model.

Carpenter 1987 Carpenter GF, The Use of Occam and Petri Nets in the Simulation of Logic Structures for the Control of Loosely Distributed Systems, Proc UK Simulation Council UKSC '87 Bangor UK Sept p30-35.

Croll 1990 Croll P and Nixon P, The Safety of Occam, Occam User Group Newsletter n12 Jan pp69-73.

Crowe 1989 Crowe WD, Hasson and Strain-Clarke, A CASE tool for designing deadlock free occam programs, in Wexler J (ed) 11th Occam User Group IOS Press p23-35.

Cullyer 1991 Cullyer WJ: Goodenough SJ and Wichmann BA, The Choice of Computer Language for use in Safety Critical Systems, Software Engineering Journal March pp51-58.

De Carlini 1991 De Carlini U and Villano U; Transputers and Parallel Architectures- Message Passing Distributed Systems; Ellis Horwood; ISBN 0-13-929050-8.

Duff 1992 Duff JR, Shortening the Manufacturing Queue, Professional Engineering May pp20-21.

FIP, Factory Instrumentation Protocol, 3 Bis Rue de la Salpetriere 54000 Nancy France.

Galletly 1990 Galletly J, Occam 2, Pitman, ISBN 0-273-03067-1.

Genrich 1987 Genrich HJ, Predicate/transition nets, in Advances in Petri nets '86 LNCS254 p207-247.

Gorton 1993 Gorton I, Parallel Program Design Using High Level Petri Nets, Concurrency- Practice and Experience v5 n2 April p87-104.

Gould 1992 Gould P, Concurrent Engineering for Survival, Professional Engineering July pp25-26.

Gracanin 1994 Gracanin D Srinivasan and Valavanis, Parameterized Petri nets and their Application to Planning and Co-ordination in Intelligent Systems, IEEE SMC24 n10 Oct p1483-1497.

Gray 1994 Gray DP and Cook, European Car Market, in Barns I and Davison (ed.) Business in Europe, Heineman Butterworth.

Grant 1990 Grant J, Time critical networks, Computer Systems Europe Jan p19-24.

Greene 1985 Greene TJ and Sadowski, Cellular Manufacturing Control, in Bignell V et al, Manufacturing Systems- Context Applications and Techniques, Blackwell ISBN 0-631-14378-5.

Heiner 1994 Heiner M Ventre and Wikarski, A Petri net based methodology to integrate qualitative and quantitative analysis, Information and Software Technology v36 n7 July p435-442.

Hoare 1985 Hoare CAR, Communicating Sequential Processes, Prentice Hall ISBN 0-13-153289-8.

Hockney 1988 Hockney R and Jesshope C, Parallel Computers 2- Architecture Programming and Algorithms, Adam Hilger ISBN 0-85274-812-4.

IEC/SC65A Draft Standard on Industrial-process Measurement and Control.

IEEE-802.3, CSMA/CD Access Method and Physical Layer Specifications, 1984.

IEEE-802.4, Token Passing Bus Access Method and Physical Layer Specifications, 1984.

IEEE-802.5, Token Ring Bus Access Method and Physical Layer Specifications, 1984.

Inmos 1988 Occam2 Reference Manual, Prentice Hall ISBN 0-13-629312-3.

Inmos 1989a Transputer Applications Notebook- Systems and Performance.

Inmos 1989b Transputer Databook- second edition.

ISO 9000 Quality Management and Quality Assurance Standards

iTools, Occam2 Toolset User Manual, Computer Systems Architects 950 North University Avenue Provo UT 84604 USA

Jelly 1994a Jelly 1 IE, Gorton and Gray, Special Issue on Software Engineering for Parallel Systems, Information and Software Technology v36 n7 p379-380.

Jelly 1994b Jelly I and Gorton, Software Engineering for Parallel Systems, Information and Software Technology v36 n7 p381-396.

Jelly 1993 Jelly IE Gorton and Gray, Using PARSE for transputer software development, World Transputer Congress '93 Aarchen Germany Sept IOS Press p950-964.

Jensen 1991 Jensen K, Coloured Petri nets- a High Level Language for Design and Analysis, in Advances in Petri nets '90 LNCS486 p342-416.

Jesty 1993 Jesty PH Buckley and West, The Development of Safe Advanced Road Transport Telematic Software, Microprocessors and Microsystems v17 n1 pp37-45.

Jones 1988 Jones G and Goldsmith M, Programming in Occam2, Prentice Hall ISBN 0-13-730334-3.

Joosen 1989 Joosen W and Verbaeten, A deadlock detection tool for occam, in Wexler J (ed) 11th Occam User Group IOS Press p36-54.

Kerridge 1984 Kerridge J and Simpson, Three Solutions for a Robot Arm Controller using Pascal-Plus Occam and Edison, Software- Practise and Experience v14 p3-15.

Kerridge 1987 Kerridge J; Occam Programming: a Practical Approach; Blackwell; ISBN 0-632-01659-0.

Kirk 1991 Kirk M, Time Critical Communication Systems, IEE Computing and Control Engineering J Jan pp35-42.

Lau 1993 Lau MWS and Seet, The use of Petri nets for occam programming for transputers, Advances in Engineering Software v17 p155-163.

Linge 1986 Linge N, Emerging LAN Technologies, Microprocessors and Microsystems v10 n1 Jan p17-24.

Mair 1993 Mair G, Mastering Manufacturing, MacMillan ISBN 0-333-54230-4.

Manson 1994 Manson GA, Sahib and Elavazuthi, Design and Code Derivation in the PCSC Methodology, Information and Software Technology v36 n7 p413-417.

MARS, MASI Blaise Pascal Institute, Universite P & M Curie, 4 Place Jussieu, 75252 Paris cedex 05, France.

McCormik 1977 McCormik BJ, Introducing Economics, Penguin.

McDermid 1991 McDermid JA, Software Engineer's Reference Book, Butterworth-Heineman ISBN 0-750-61040-9.

MIL-STD-883C 1990, Test Methods and Procedures for Microelectronics.

MMS, BS-ISO-IEC 9506:1990 Manufacturing Message Specification.

Morales 1994 Morales R, Flexible Production- Restructuring of the International Automobile Industry, Polity.

Morgan 1988 Morgan PJ, Precise Design Helps Production, Professional Engineering Sept p61-63.

Moser 1990 Moser LE and Melliar-Smith PM, Formal Verification of Safety Critical Systems, Software - Practice and Experience v20 n8 Aug pp799-821.

Norcliffe 1995 Norcliffe A, The Revolution in Mathematics due to Computing, Inaugural Lecture, Sheffield Hallam University Nov.

Opitz 1971 Opitz H and Wlendahl, Group Technology and Manufacturing Systems for Small and Medium Quantity Production, Int. J Production Research v9 n1 pp181-203.

Peterson 1981 Peterson J L, Petri Net Theory and the Modelling of Systems, Prentice Hall ISBN 0-13-661983-5.

Pountain 1987 Pountain D and May D; A Tutorial Introduction to Occam Programming; BSP Professional; ISBN 0-632-01847-X;.

Reeve 1993 Reeve A, Fielding a Revolutionary Standard, Professional Engineering June p14-15.

Roman 1985 Roman GC, A Taxonomy of Current Issues in Requirements Engineering, IEEE Computer April p14-21.

Schafers 1993 Scafers L Scheidler and Kramer, TRAPPER a Graphical programming environment for embedded MIMD computers, World Transputer Congress '93 Aarchen Germany Sept IOS Press p1023-1034.

Shimeall 1991 Shimeall TJ and Leveson NG, An Empirical Comparison of Software Fault Tolerance and Fault Elimination, IEEE SE17 N2 Feb pp173-182.

Silva 1989 Silva M, Petri nets and flexible manufacturing, in Rozenberg (ed) LNCS v424 p374-417.

Steinmetz 1987 Steinmetz R, Relationship between Petri Nets and the Synchronisation Mechanisms of the Concurrent Languages Chill and Occam, 8th European Workshop Application and Theory of Petri Nets May p509-529.

Thompson 1987 Thompson J, CIM- Bringing the Islands Together, Chartered Mechanical Engineer March p49.

Voss 1988 Voss CA, Just-in-time, Chartered Mechanical Engineer April p29-31.

Welch 1993 Welch P Justo and Willcock, High level paradigms for deadlock free high performance systems, World Transputer Congress Aarchen Germany Sept p981-1004.

Wemmerlov 1987 Wemmerlov U and Hyer, Research Issues in Cellular Manufacturing, Int. J of Production Research v25 n3 pp413-431.

Weston 1987 Weston R Sumpter and Gascoigne, Industrial Computer Networks and the Role of MAP-part II, Microprocessors and Microsystems v11 n1 Jan pp21-33.

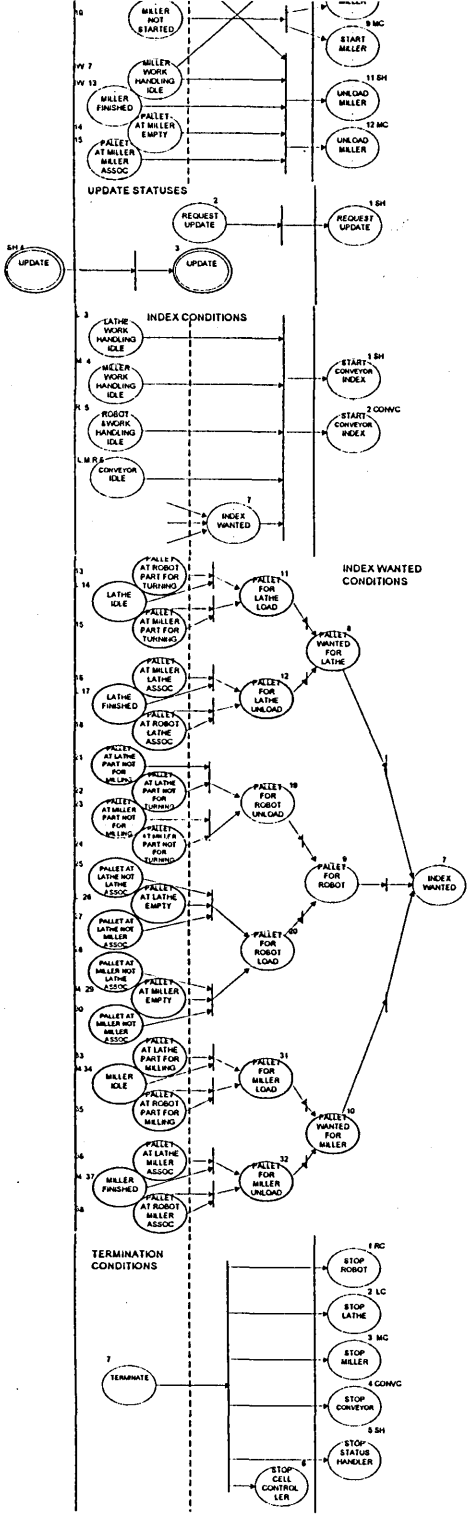
Weston 1991 Weston RH, Coutts, Hodgson, Murgatroyd and Gascoigne, Generally Applicable Cell Controllers and Examples of their Use, in Williams DJ and Rogers P (ed), Manufacturing Cells-Control Programming and Integration, Butterworth ISBN 0-7506-0235-X

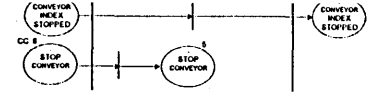
Williams 1994 Williams DJ, Manufacturing Systems - an introduction to the technology, Chapmam Hall.

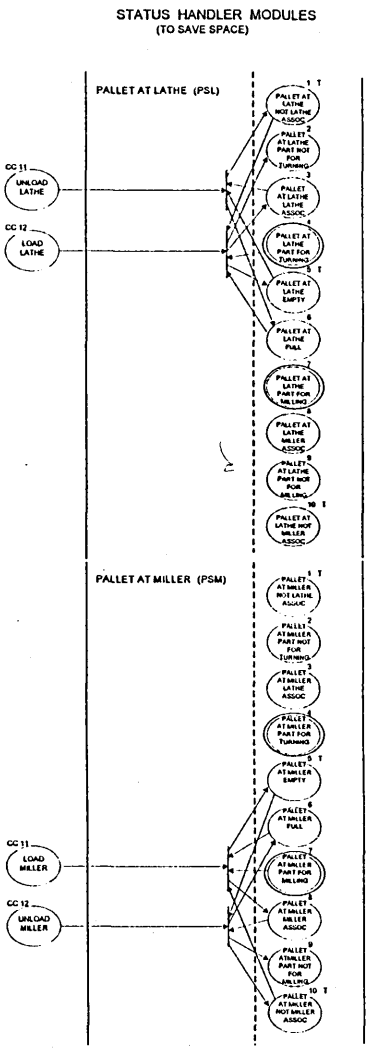
Xu 1991 Xu Z and de Vel, Petri Net Modelling of Occam Programs for Detecting Indeterminacy Non-termination and Deadlock Anomalies, Petri Nets and Performance Models- Proc 4th Int Workshop Ausralia, IEEE Press ISBN 0-8186-2285-7, p116-124.

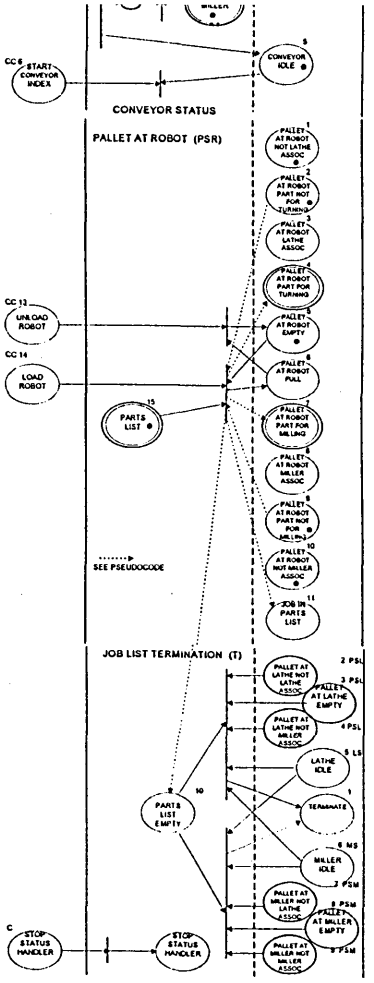
Yeomans 1985 Yeomans RW Choudry and Hagen (ed.), Design Tasks for a CIM System- CEC ESPRIT, Elsevier Science Publishers ISBN 0-444-878122.


CLOSE DOS
CLOSE DOS
PALLET AT MILLER









The central of the three columns includes two modules which are part of the status handler

The overall Petri net graph