Functionally Encapsulated Modules: A Computer Aided Software Engineering Methodology for the Implementation of Computer Integrated Manufacturing.

Thesis Submitted to the University of Wales for the Degree of

Doctor of Philosophy

By

Eric Llewellyn, BSc (Hons) AFHEA FBCS CITP
Department of Business and Computing
University of Wales Newport
May 2009

Declarations

DECLARATION

This work has not previously been accepted in any substance for any degree and is not being currently submitted in candidature for any degree.

Signed ..... (candidate)

Date ..... 1/5/2009 .....

STATEMENT 1

This thesis is the result of my own investigations, except where otherwise stated.
Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed ..... (candidate)

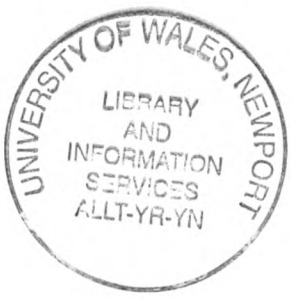
Date ..... 1/5/2009 .....

STATEMENT 2

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ..... (candidate)

Date ..... 1/5/2009 .....



Acknowledgements

I would like to thank my supervisors Dr Martin Stanton and Dr Torbjørn Dahl for their guidance and support, Mary Evans and Angharad Jones for their friendship and belief that I would finish. Special thanks to Tony Corner, Alan Hayes, Dr David Morris and Dr Andrew Thomas for giving me the time to write up, and all my colleagues at Newport for their moral support and encouragement. I would also like to thank my wife Angela, Rachel and my children Daniel, Bethany, Angharad and Mair for losing my time for all these years. Finally, I would like to dedicate this work to my Mother who has struggled long and hard to raise us and I hope this pays back some of that hardship.

Summary

The thesis describes a method for the rapid, incremental design and implementation of manufacturing systems utilising a combined object-oriented and structured Petri net formalism. The background to the problems facing manufacturing organisations wishing to implement computers into manufacturing systems is presented along with a discussion of how software engineering techniques can be applied to overcome them. Modularity and object-orientation are proposed as a way of enhancing the development of manufacturing systems. A review of current techniques for modelling manufacturing systems is presented which outlines the benefits and drawbacks of a number of methods. A three-level control architecture is developed which distributes complexity amongst the low levels of the system. The control structure is combined with a behavioural constraint object to ensure that maximum reuse can be gained from objects in the system. A formalism for integrating Petri nets into the UML is outlined, entitled Functionally Encapsulated Modules. These modules provide full object-oriented capabilities coupled with the functional modelling power of Petri nets. State space explosion is reduced as Petri nets are used only for modelling the functionality of objects. However, the modules also retain the abilities of simulation tool and mathematical proof of the original Petri net. The methodology and modelling tools are evaluated by applying them to a discrete event manufacturing system. Conclusions are then drawn on the various aspects of the work and details of further research possibilities are described.

Table of Contents

List of Figures

List of Tables

Chapter 1
Background to the Research and Thesis Outline

This chapter analyses systems modelling in a manufacturing context and establishes the scope for cross fertilisation of this field from techniques successfully applied in the discipline of software engineering. To achieve this goal, the concept of a hardware/software object (HSO) is defined as an object which visualises a machine not as a hardware entity, but as the software that controls it. Utilising the HSO approach facilitates a more direct correlation between the design of a manufacturing system, containing a myriad of interrelating hardware and software, with that of traditional software development. This chapter outlines the motivation and scope of the work presented in this thesis. Initially, the work is contextualised before the aims and objects are clearly stated. A background to the problems facing twenty-first century manufacturing organisations is described that highlights the change in philosophy since the introduction of large scale mass production during the Second World War, along with its associated problems for system designers.

1.1 Introduction to the Work

This research work closes the loop on the object-oriented design methodology for the implementation of Computer Integrated Manufacturing (CIM). This is achieved by merging the Unified Modelling Language (UML) and Structured Petri nets (Stanton, 1999). A combined methodology and modelling tool has been created which enables manufacturing system designers to develop systems which utilise the full reuse capabilities of object-orientation combined with the powerful functional modelling of Petri nets. Providing true object-oriented capabilities to Petri net graphs is an original contribution of this work. A novel methodology has been developed which enables system designers to take a top-down approach to system development. The methodology facilitates a fully modular and incremental approach to the design, development and implementation of manufacturing systems. This work also expands upon the concept of a combined Hardware/Software Object (HSO) to reduce design complexity. HSOs have been conclusively defined and are inherent in the full development process described in this thesis. Utilising HSOs enables manufacturing systems, containing a range of hardware and software, to be visualised as purely software systems. The work also introduces the novel concept of behavioural objects which are designed to maximise reuse capabilities within manufacturing systems. The behavioural objects also ensure that manufacturing system designers can take advantage of a

library of fully designed and tested components to speed up future system design or redesign. A new three level control structure which distributes the complexity of the system from a control perspective has been designed that fully integrates with the design methodology proposed in this work. The control structure proposed in this work is designed to ensure that maximum reuse capabilities are achieved in the design of system controllers. Finally the unique modelling tool entitled Functionally Encapsulated Modules (FEM), created in this work, enable the generation of simulation models and provide a method of mapping to fully functional control code.

1.2 Aim of the Research

The aim of this research is to close the loop on object-oriented modelling in a manufacturing context. This is achieved by utilising a novel methodology and modelling technique based upon the UML and structured Petri nets. The aim achieved in this work will provide manufacturing organisations with a solution to the three problems (goals) identified in this chapter:

1. Speed of design and development;
2. Costs to be kept to a minimum;
3. Quality through consistency.

This will be achieved by establishing an object-oriented methodology for the analysis and design of manufacturing systems which allows such systems to be rapidly designed and incrementally implemented. The methodology used needs to satisfy the dual aims of being intuitive for users to understand, but detailed enough to actually see the process through to implementation. The technique will also provide system designers with a simulation tool, which can subsequently generate optimised control code.

1.3 Achieving the Objectives

The work reported here provides manufacturing system designers with a Computer Aided Software Engineering (CASE) methodology for the incremental implementation of Computer Integrated Manufacturing (CIM). A methodology is defined in this work, which contains a number of elements that work towards achieving this aim.

The proposed design method is based upon a novel approach called Functionally Encapsulated Modules (FEM). FEM are a combination of the Unified Modelling Language (UML), which has become the *de facto* standard for modelling software systems in the sphere of software engineering, and structured Petri nets as defined by Stanton (1999). Initially the system is examined and discussions with users carried out to define the boundaries of the system(s) under consideration. This enables the construction of a suite of iteratively refined diagrams representing the modular structure of the system. The primary models define both the boundaries of the system under investigation and the input and output processes that are utilised at sub module level within the system. Further these models are used to ensure a coherent interface between the individually designed sub-systems under consideration and the larger system as a whole. Use-case diagrams are used to provide a user-centric view of the system that enables designers to capture the main processes of the system as visualised by its stakeholders. Here a user can be described as a human who

interacts with the system in some way, or a separate process which requests some operation be carried out by another module in the system. This highly modular approach to systems' design provides a loosely coupled system (Pressman, 2004) that is receptive to the incremental nature that this work emphasises as all-important to modern manufacturing organisations. The end result of this stage is the identification of a number of sub-systems or modules, within the system under consideration. These can be analysed and automated independently of each other to reduce the disruption to the system as a whole, or can be considered concurrently by teams.

Having established the building blocks of each module, the classes in object-oriented parlance, using the UML notation a model of the static system can be produced. This model is then enhanced with structured Petri nets which capture both the dynamic behaviour and state representation of the system. The Petri nets can be used to simulate system processes and the changes in state undergone by the various HSOs in the system to ensure suitability for purpose, safety of operation and allow optimisation of the processes.

One of the key features of object-oriented modelling languages is their ability to provide template classes, which can be used to build a collection of reusable components. These components provide a library of pre-built objects that can be taken as needed and used to build

modules or indeed complete systems. However, one problem with this approach is the need to amend objects for different systems that in many instances can mean a considerable amount of modification is needed before a generic object can actually be used within a new system. This work overcomes this problem by defining 'constraint objects' which are used to provide an interface between the main system control object (goal control) and the lower level module and sub-module (task control) objects. This allows designers to make full use of the features presented by modern object oriented languages, whilst maintaining full reuse capabilities to a level unachievable in most current systems.

The throwaway approach to developing system prototypes can mean hundreds of wasted man hours as once a system is designed the prototype, which has been used for simulation, is discarded and new software is developed. In this work a novel approach is taken which utilises the Petri net objects for the modelling and simulation of the system, but which can then generate optimised and robust code for controlling the final implementation. The tool can also be used for further optimisation and debugging later in the life of the working system, or indeed as a useful way of visualising and simulating the finished system.

1.4 The Change in Manufacturing Philosophy

During the '30 glorious years' between 1945 and 1975, as defined by Waldner (1992), world economies have shifted from markets of abundance to those dominated by supply. The manufacturing response has been to move from large-scale mass-production to a highly customised and generally small-scale production of products. The origins of this shift can be traced back to the privations suffered during World War Two. Immediately following the war public demand was for mass produced, low cost items, however with the war a distant memory, increasingly customers began to demand higher quality, lower cost and highly customisable products. Typically the current lifespan of a product can be measured in months rather than years. In many cases existing products are enhanced rather than replaced, for example the Apple iPod range is revamped approximately every twelve months. Though the basic technological structure of the iPod, remains the same revisions generally apply to aesthetics or modular components, such as increased storage capacity. From the production perspective these can be considered to be alternatives in components during the production phase which can be accommodated from a system designer's viewpoint as optional flows in the build process. It is clear that once the basic functionality of a product is developed enhancements are added in the form of modular additions.

Any organisation wishing to survive in the global economy of the twenty-first century must respond quickly to abrupt market variations as this global manufacturing environment becomes highly dynamic and increasingly competitive (Wong *et al*, 1999). In this context reliability and flexibility become the important factors in production processes (Waldner, 1992). Flexibility in this context can be interpreted as the need to respond quickly to market fluctuations which requires designers to produce system designs quickly and efficiently in order to meet new demands, although, as previously highlighted, this can relate to product enhancement equally as to new product development. Despite customer requirements for low cost items in the shortest timescale possible, there is still an expectation that products will maintain the highest standard in terms of quality, cost and reliability (Prasad 1999, Kara *et al* 1999, Minderhoud 1999). This has enforced changes in the way designers and manufacturing engineers develop their systems (Jiang *et al*, 2002).

Manufacturing organisations, in common with other businesses in the twenty-first century, are driven by the economies of a global market. In order to remain competitive it is established in this work that it is imperative for them to meet three key goals - get their goods to the market in a shorter time period, produce their products at a lower cost, and achieve a higher quality than their competitors. These goals provide the fundamental underpinnings of the manufacturing

requirements for competitive advantage and this work aims to provide a solution which addresses all of these needs.

1.5 Three Goals for Manufacturing Organisations

1.5.1 Goal One – Speed

Manufacturing system designers need a methodology which facilitates a quick turnaround of a new system from design, or redesign, through to implementation. However, due to the need for competitive advantage and customer satisfaction a 'first time right' approach is also required. Manufacturing organisations cannot afford to utilise long and cumbersome techniques due to the fact that the market may have quickly moved on, making the design obsolete before it gets beyond the conceptual or design stages. Conversely any design methodology used must ensure that the completed system achieves the user requirements fully. Within the discipline of software engineering, CASE tools have had a major impact on the speed with which an item can be conceptualised and subsequently visualised in a model. However due to the complexity involved in manufacturing systems – composed of a myriad of inter-connected hardware, software and communication systems – their upgrade or redesign can have a considerable impact on the time it will take to move from this visualised design to final production.

Software engineers have made progress in this area by utilising a modular approach to the design of software systems. By breaking a

system down into a number of distinct modules it is possible to divide the workload amongst a number of developers who can each dedicated their entire efforts on a subset of the system. This can have a dramatic impact on the speed at which systems are developed, though it does require a high degree of consistency amongst developers to ensure that all modules can integrate to form a final system. In a manufacturing context concurrent development of this type would dramatically reduce the time required to develop complete and working systems. Many researchers have proposed an integrated design method using concurrent engineering (Lu *et al* 1999, Chen and Jan 2000, Herder and Weijnen 2000, Senin *et al* 2000, Wu and O'Grady, 2000). However, most literature deals mainly with issues related to assembly, cost reduction and quality deployment (Dembeck and Gibson 1999, Ke 1999, Liu and Yang 1999, Swanstrom and Hawke 1999). A loosely coupled system as proposed by Pressman (2004), and as discussed in more detail in chapter 2, would facilitate an incremental upgrade of parts of a system with minimal impact on other components. This technique also eliminates the problems associated with the 'islands of automation' identified by Hannam (1997) and discussed later in this chapter.

The widely used concept of object-orientation and the development of class libraries, quite common in software engineering, are also applicable to manufacturing systems. By utilising readymade, pre-

tested, high quality components that have previously been developed, manufacturing system designers can considerably reduce the time and cost needed to develop a system, especially if the components and processes used are a variation of those currently in existence. Utilising fully tested pre-made components will go a long way to addressing issues with quality in the implementation stages by introducing a degree of certainty that the component will do the job for which it was intended. The use of hardware/software objects, where a system component is thought of in terms of what it can do rather than how it works, lends itself well to object-orientation.

1.5.2 Goal Two – Cost

As with any business it is important to maximise profits by reducing development costs. Utilising reusable code libraries as outlined in goal one, means that the effort of designers, developers and testers is captured, allowing subsequent new designs or those using similar components to take full advantage of work previously undertaken. This can give a considerable reduction in design and development costs as much of the system will not need to be re-engineered. To maximise the benefit of code-reuse a method needs to be established which enables the optimum use of generic components. Generally, the control software for a system or the objects themselves will have to be customised to ensure compatibility each type a new system is

developed. This can be a time consuming venture which can mean that, in some cases, it is quicker to develop new software.

This work has examined the control structure of a manufacturing system and established an optimum way of designing controllers and objects which maximises their reuse capabilities. One of the fundamentals of modularity is a robustly designed public interface and manufacturing facilities require a system that will facilitate the 'plug and play' type approach enjoyed in the computing field. This will enable parts of a system to be upgraded with a minimal impact on the rest of the system, and will allow manufacturing organisations to upgrade with reduced system down-time. Such a system would fully adhere to the criteria of modularity proposed by Meyer (1997) which is discussed in more detail in chapter 2.

1.5.3 Goal Three – Quality through Consistency

Quality does not simply refer to the product and its development/manufacturing processes, but realistically it should apply to all levels in an organisation. Utilising different design methodologies at various levels of the organisation does nothing to aid communication between stakeholders. Ideally, manufacturing organisations require a methodology which captures enterprise level through to functional detail, by utilising a design methodology that can be applied to all aspects of the organisation in a uniform and standardised way. This

ensures that communication between stakeholders at all levels of the organisation becomes more intuitive thereby ensuring that the end product or system closes matches user expectations. The Unified Modelling Language (UML) provides software engineers with a method for modelling the processes at all levels of an organisation in a standardised manner, thereby facilitating a robust method of capturing user requirements. The applicability of the UML as a consistent method of modelling all aspects of a manufacturing organisation has been evaluated and clearly established in this work. The standard UML models have been adapted to incorporate a Petri net graph which reduces the number of models required and yet ensures the system can be abstracted in a manner that is understandable to all stakeholders.

The development of class libraries also means that high quality, fully tested components are available off the shelf ensuring their robustness in new systems. Further a modular approach, which satisfies the criteria outlined by Meyer (1997), ensures that in the event of errors the impact on the rest of the system is minimal, if at all, and it is possible to diagnose and trace faults into very specific parts of the system, thereby reducing the amount of time spent maintaining, repairing or upgrading the system.

1.6 Computer Integrated Manufacturing

Computer Integrated Manufacturing (CIM) has been proposed as one solution to the manufacturing problems outlined above, where computer systems have been implemented into the manufacturing environment to increase speed and efficiency, however without detailed and thorough planning this can lead to a whole new range of problems. For sustainable increases in market share and profit margins, it is the system development practices that require attention (Chin *et al*, 2005). Manufacturing organisations can rarely close down completely for an upgrade as the disruption and potential market loss is too great. When a new product is to be manufactured, the organisation must ensure that the turnaround time is as quick as possible. CIM helps to achieve this by implementing computers into the organisation, with the aim of reducing turnaround times, increasing quality and minimising costs, thereby moving some way to addressing the three goals previously outlined. However, research shows (Hannam, 1997) that one of the main problems with CIM are 'islands of automation', which occur as a result of the computerisation of individual departments or even cells within an organisation. These computerised sections are generally implemented with no ability to communicate with other units within the facility. Two very important facts about CIM can be drawn from the literature:

1. Computerisation can help to achieve the three goals of manufacturing organisations though it does have inherent problems;

2. Disruption to the operation of the overall system needs to be minimised whilst upgrades are in progress.

Each of the points identified above are evaluated in more detail below:

1.6.1 Computerisation as a solution

Computerisation is vital in manufacturing and CIM is an important aspect of such systems. Plainly speaking, a computer can do things more rapidly than a human and is less prone to mistakes, especially when working in hazardous environments or long, unsociable hours. The computer also provides tools to integrate the whole process from product conception to marketing. For instance, it is possible to use a computer aided design tool to produce a first draft of the idea, the computer would then be used to aid in requirements gathering, to simulate the production processes and ultimately as a controller for the finished system. A computerised solution, however, is only as good as the human operators and system designers and therefore it is imperative that the design methodology used is thorough, robust, all encompassing and intuitive enough to facilitate communication between system designers and end users. Ideally the method used should

facilitate the modelling of all levels of system within an organisation from business processes to hardware and software itself.

1.6.2 Incremental Implementation to Minimise Disruption

As outlined previously there is a crucial need to reduce disruption within manufacturing organisations when redesigns or upgrades are taking place, and therefore a non-disruptive method of incremental implementation is needed. The major benefit of incremental implementation is that the organisation or facility can be broken down into a series of discrete modules, which are upgraded in isolation from each other. This means that the disruption to other units is minimal due to the fact that the unit being 'conceptually upgraded' remains available all the way through to the physical upgrade. The incremental approach can be facilitated by using the object-oriented concept of 'encapsulation' whereby a very clear interface to the unit is designed, with no concern for the 'hidden' detail of how it actually functions. For example, if a large and complex cell containing several items of machinery were tasked with producing one machined component upon receipt of one raw material, then the interface to this unit can be defined as one input and one output. In order to effectively integrate with other units in the system all that is required by the controller is knowledge of these inputs and outputs. The incremental approach is suggested as a method of speeding up the design through to implementation process subsequently overcoming the problems identified. In addition its

modular nature lends itself well to the design of class libraries, where a series of pre-designed and pre-tested components can be used to build cells and departments in a modular fashion. The well defined interfaces outlined above enable manufacturing organisations to overcome the problems associated with islands of automation. This work will describe a method and formalism for developing such interfaces based on a combination of well defined public interfaces, *via* the UML and through message passing *via* Petri net control and feedback places (which are discussed further in Chapter 3).

1.7 The Need for Integration

Waldner (1992) defines integration as the need to 'remove the boundaries between the functions of a company which for justifiable historic reasons were previously split up'. The larger an organisation, the more its functions are distributed between a number of different departments, each with its own goals and responsibilities. The difficulty this situation presents is that often, each of the departments may pursue its own agenda and pay little attention to the overall objectives, which are to satisfy the customer in the shortest time, at the lowest cost and to the highest quality. CIM was developed with the aim of establishing a close relationship between various functional units by capitalising on the most basic resource available to a business: information (Waldner, 1992). However Waldner (1992) goes on to highlight how the rush to computerise has meant that frequently automation has been carried out purely for automation's sake leading to the expensive automation of systems that are more efficient in their manual form. Another problem, highlighted earlier, are 'islands of automation' (Hannam, 1997) where individual components or departments within a manufacturing organisation have been automated with no thought to how they will communicate with other equipment, cells or even departments. With the rapid change in technology and the flexibility required to change production to meet market demands, this problem has been exacerbated. The challenge therefore is to integrate

all functions across a manufacturing organisation where each sub-system or department must meet its own goals and carry out its specific tasks, but this should be placed within the context of the larger aim of the organisation as a whole and this problem is conclusively addressed in this work by analysing modelling techniques than can span a whole organisation.

1.7.1 Software for Integration

Software can be considered as the integrated manufacturing problem due to the fact that whilst the technology exists the software and modelling methods needed to use it in an integrated fashion does not. This problem is addressed in this work by the use of HSOs. The relatively small amount of integrated systems that do exist are large are reported as being application specific, difficult to maintain, difficult to change, difficult to port between hardware and expensive (Naylor and Voltz, 1987). Software is the intelligent part of the system and it is therefore vitally important to get this part right, however the hardware cannot be ignored as without this the system does not function. To address this problem this work has examined the concept of hardware/software objects and established a clear method for their integration into the methodology developed.

1.7.2 Hardware/Software Objects

In order to achieve complete integration of manufacturing systems, it is important to begin with the lowest level of a system. Here the hardware and software of a device is "encapsulated" as a hardware/software object (HSO). A device is thought of in terms of its functionality as a unit and not separately as a piece of hardware and its control software. Naylor and Voltz (1987) define such a component has having three basic characteristics:

1. A well defined public interface;
2. An internal implementation that is inaccessible to the user; and
3. Both the visible part and the inaccessible implementation of software components should be separately compilable from the program components that use them.

A well-defined public interface allows the object to be reused in various situations, other than those for which it may have initially been designed. Coupled with the inaccessible internal implementation, all that a user and the system's controller require is a knowledge of what functions the object is capable of performing. No knowledge is needed of how these actions are actually performed. This alleviates the need for the system designer to have a detailed understanding of the specific implementation of a component. Instead all they need is an understanding of what its goal is. Conversely it allows the complexity of

the software in overall control of the system to be uncomplicated by allowing it to act as a sequencer of modules, each of which sequences sub-modules as required with the complexity being dispersed through the lower levels of the system. This gives the loosely coupled system described by Pressman (2004) where changes to one object have minimal impact on the rest of the system.

1.8 Merging the Unified Modelling Language and Petri Net Graphs

The competitiveness of manufacturing and the globalisation of markets mean that any new product must arrive with the customer as quickly as possible. This may mean the upgrade or redesign of a facility and its processes, and one way to achieve this is through the incremental approach described in section 1.7. The modular nature of manufacturing systems lends itself well to an object-oriented methodology. However, these tend to be long and drawn out procedures due to the number of resultant diagrams and their complexity. Petri nets are in common usage in manufacturing system design, as attested by the quantity of published material available (see Chapter 3). Several attempts have been made at integrating object-oriented techniques with Petri net theory but these tend to fall short of full object-orientation. The main difficulty to overcome with Petri net graphs is 'state space explosion' which describes the complexity of diagram needed to reproduce even a simple system making communication between designers and users difficult. A novel contribution presented in this thesis overcomes this problem by utilising structured Petri nets to model only the functional detail of objects within the system, whilst the remainder of the system can utilise the benefits of the UML.

1.9 Thesis Structure

Chapter 2 will discuss the rationale for utilising an incremental design approach to overcome the problems identified in this chapter. Modularity will be highlighted as a vital first step in achieving such an incremental design technique and Meyer's criteria (Meyer, 1997) will be illustrated as a benchmark which ensures the work fully meets the criteria for modularity. A comprehensive review and analysis of object-orientation and the UML to manufacturing organisations will be undertaken along with a literature review outlining the various attempts at providing manufacturing system modellers with an object-oriented technique akin to that found in software engineering disciplines. The concepts of loosely coupled well designed interfaces will be examined as a method of achieving hardware/software objects which will overcome the problems of islands of automation.

Chapter 3 evaluates Petri nets as an existing, widely used, method for modelling manufacturing systems, and a brief description will be given of the evolution of the Petri net model. A review of extensions to the basic formalism will be demonstrated, along with an overview of the advantages and disadvantages of each method. A comprehensive literature review of Petri net theory will be presented, and in particular an in-depth discussion will be given of attempts at providing an object-oriented Petri net technique. The relative strengths and weaknesses of

these attempts will be evaluated, including methods of overcoming the state space explosion problem, and their use in simulation.

Chapter 4 will introduce Structured Petri nets and a novel modelling technique will be presented which demonstrates how these can be integrated with the UML to provide a complete manufacturing modelling technique. This chapter will show how the resultant Functionally Encapsulated Modules (FEM) take full advantage of current object-oriented techniques and use the flexibility of structured Petri nets to provide a versatile, mathematically provable modelling method. The problems of creating a generic family of classes will be outlined and a solution is presented in the form of three-level control architecture. The three components of this constraint-based approach will be discussed and goal, task and environmental constraints will be introduced. Examples will be given to show the flexibility of this approach, and a technique for re-creating traditional Petri nets will be described. The mathematical provability of this approach will be briefly outlined before a summary of the technique is given.

Chapter 5 will provide a case-study which demonstrates how the methodology developed in this work applies to a modern manufacturing system. The chapter will demonstrate how the resultant models can initially be used for simulation and what if analysis before automatically generating the control code for implementation. A demonstration will be

given of how, using the approach developed in Chapters 1 – 4, a model can be developed which can be used to simulate the system and/or to physically provide working control code. The focus in this chapter will be developing a technique whereby FEM and behavioural constraints can be utilised to generate an optimised pseudo code module, which can then be interpreted and compiled in whichever language is necessary.

Finally chapter 6, presents a detailed analysis of how this thesis and the original work described in this thesis provides a solution to the manufacturing problems outlined in this chapter. An indication of future work arising from the main body of the dissertation is also presented.

1.10 Overview of the Work

This work describes a method for the rapid and incremental analysis, design and implementation of manufacturing systems and their control software. The method incorporates a modular approach to manufacturing system's design by utilising a combined object-oriented and Petri net method for the development of both hardware and software elements.

Additionally, the methodology proposed allows for the construction of highly generic reusable components utilising a constraint-based approach that can aid in speeding up the design and development of manufacturing systems. The technique allows system designers to build partial or complete manufacturing systems from a library of pre-defined reusable components. These components are pre-tested, ensuring their reliability and quality, and can be readily adapted to the needs of new systems, in many cases with minimal, if any, modification.

Further, the proposed technique permits the object-oriented model to be used as a simulation tool which allows organisations to evaluate and optimise new systems and processes before they are implemented, ensuring full satisfaction of user requirements, complete system testing and facilitating the evaluation of alternative design scenarios before procurement of expensive equipment takes place.

Finally, a generative approach is used to enable the model to automatically generate the code, which controls the system once it has been implemented. The control code is enhanced by the fact that the system will have been fully tested via simulation before optimised code is generated.

1.11 Indication of Contributions

The main contributions arising from the work are outlined below and each is discussed in further detail In Chapter 6:

1.11.1 The Application of the UML to Manufacturing Systems

A novel approach has been taken in applying the UML to manufacturing systems analysis and design (Llewellyn et al, 2000). This offers many benefits for manufacturing organisations including the provision of a reusable system, and the opportunity to build a library of classes, which makes subsequent designs or modifications to existing systems more intuitive. The UML provides manufacturing organisations with the benefits of object-orientation that have been successfully implemented in the software engineering community for sometime. These benefits include encapsulation, inheritance and the ability to use class hierarchies. By focusing on the objects and their interactions via a public interface, the dynamics of the system can be presented to technical and non-technical users, allowing the designer to focus on what the object/system is to do, without an in-depth knowledge of how it does it. The UML also facilitates the unique ability to model all aspects of a manufacturing organisation from business processes through to shop floor machinery.

1.11.2 Defining a Methodology for the Incremental Analysis of CIM Systems

An incremental approach to the analysis of CIM systems enables manufacturing organisations to computerise anything from individual manufacturing workstations through to entire departments on a staged basis. This reduces the need to close down entire facilities, and allows upgrades to be carried out as and when required. Existing systems can be modified to work alongside new systems. By using a design approach that utilises use-case analysis it is possible to capture the user requirements for a system more accurately and in a format which enhances communication between system modellers and stakeholders. The design stages of a use-case driven approach take into account the needs of all levels of the workforce, ensuring all personnel are involved in the process. The initial use-case scenarios used to capture the system requirements can be reused at the testing stage to verify all requirements are adequately met.

1.11.3 Development of a Three Level Control Architecture optimised for reuse capabilities

The hybrid bottom-up and top-down approach of the incremental methodology proposed enables the controllers required at all levels of the system to be adequately modelled and ensures the functionality of the system is maintained. The modular approach proposed also allows system changes to be more easily accommodated. The object-oriented

approach to the system design allows designers to capture the system at its most generic, but also provides a method of capturing constraints on the system such as obstacles to the dynamic capabilities of objects. Three types of constraint have been developed – goal, behavioural/environmental and task and these enable the system to make maximum use of the benefits offered by object-orientation.

1.11.4 Merging the UML and Petri Nets

A technique for successfully combining the UML and Petri nets has been developed called Functionally Encapsulated Modules (FEM) (Llewellyn et al, 2001). The technique takes two existing UML diagrams namely sequence and behaviour diagrams, and replaces them with a single Petri net. This approach offers the design simplicity of the original Petri net and combines them with the proven advantages of object-oriented analysis and design. FEM reduce the number of diagrams required to model both state and behaviour of systems and individual objects and a comparison of the FEM versus traditional UML approach is demonstrated in Chapter 4. If required the Petri net elements of a FEM can be combined to form one large net that can be verified using proven Petri net techniques. This approach provides a modular, object-oriented technique for utilising Petri net models whilst eliminating state space explosion. The FEM's develop a unique method of capturing the attributes of both software and hardware which can be intuitively implemented into any manufacturing system. The

encapsulation of hardware and software with a distinct user interface allows the designer, and the users of the system, to visualise the objects that make up the system's model without worrying about the inherent complexity.

1.11.5 Simulation and Automated Code Generation

The "token player" aspect of structured Petri nets will enable the models resulting for the use of this approach to provide the basis of a simulation tool. Further, the combined Petri net/UML approach lends itself well to the automatic generation of control code and a methodology for this technique will be presented. The code generation aspect of the models relies on the simplicity of Petri net graphs where places are represented as Boolean values whilst transitions represent decision statements. The automated procedure will generate pseudo code, which can be intuitively converted into any programming language.

1.12 Chapter Summary

This chapter has provided on overview of this research thesis, and presented the main contributions presented herein. The move from large-scale mass production to the development of highly customised, small scale production has been discussed and this work has identified the main resultant problems faced by manufacturing organisations wishing to compete in the global economy of the twenty-first century:

- Speed. With an increasing number of competitors, it is imperative that organisations reduce the time it takes to move from the conception of a new product, to the finished item being available for the consumer.
- Cost. Customers are now demanding lower priced goods than ever before and therefore it is important that goods are manufactured in the most optimised fashion possible.
- Quality through Consistency. Though it may seem offset by the above two points, consumers demand high quality products, and offering anything less can have serious implications for the all important customer loyalty factor.

It is clear that product life-spans are measured in much shorter time periods than were previously the norm, however it is also apparent that in many cases products undergo redevelopment or enhancement rather than completely new development. This work aims to establish that

these types of upgrades can be visualised and implemented as flows to existing systems, considerably reducing redevelopment times and costs. It is apparent that manufacturing organisations adopting an incremental and modular approach to systems' development could overcome the problems of speed, cost and quality through consistency and would benefit greatly from the use of combined hardware/software objects that would enable designers to concentrate on what the system does rather than get bogged down in the detail of how it does it.

A well-defined public interface to such objects would ensure that the islands of automation problem is completely addressed by breaking the system down into a number of distinct modules which can be divided amongst a team of developers. The use of public interfaces will also reduce downtime considerably as for much of the process modules are only conceptually upgraded. This solution will dramatically reduce development times and be a vital step in addressing quality by consistency.

Generic and highly reusable objects will enable system builders to utilise previously designed high quality components that will rapidly decrease development times whilst maintaining quality. Utilising such pre-tested, high quality components would clearly address the need for a "first time right" design.

By utilising the UML for the design of such systems manufacturing organisations can benefit from the ability of the technique to model all elements of the company enhancing communication amongst stakeholders and ensuring organisational consistency. The integration of Petri nets into the UML reduces the number of models required and solves the state space explosion problem. Such a technique provides a ready-made simulation and testing tool and lends itself well to the automatic generation of control code considerably reducing the time to implementation.

This remainder of this work will analyse systems modelling in a manufacturing context and will establish the optimum methods from the discipline of software engineering to overcome the problems identified. Whilst software engineering does not generally concern itself with hardware, the use of hardware/software objects (HSO) as developed in this work enables the system modeller to consider any manufacturing design problem purely as software. This ensures that the techniques developed in this work have full applicability in a manufacturing context where systems are composed of a myriad of complex and inter-related hardware and software elements.

References

Chen, Y.M. and Jan, Y.D. 2000. Enabling allied concurrent engineering through distributed engineering information management. *Robotics and Computer-Integrated Manufacturing*. **16**(1), pp.9–27.

Chin, Kwai-Sang, Lam, J., Chan, J.S.F., Poon, K.K. and Yang, Jianbo 2005. A CIMOSA presentation of an integrated product design review framework. *International Journal of Computer Integrated Manufacturing*. **84**(4), pp.260- 278

Dembeck, W. and Gibson, D. 1999. Integrating the quality assurance function into the new product development process. *Annual Quality Congress Transactions Proceedings of the 1999 ASQ's 53rd Annual Quality Congress*. **53**(0), pp.238–243.

Hannam, Roger. 1997. *Computer Integrated Manufacturing: from concepts to realisation*. Harlow: Addison-Wesley. 0201175460

Herder, P.M. and Weijnen, M.P.C. 2000. Concurrent engineering approach to chemical process design. *International Journal of Production Economics*. **64**(1), pp.311–318.

Jiang, Z., Harrison, D.K., Cheng, K. 2002. An Integrated Concurrent Engineering Approach to the Design and Manufacture of Complex Systems. *The International Journal of Advanced Manufacturing Technology*. **20**(5), pp.319-325.

Kara, S., Kayis, B. and Kaebernick, H. 1999. Modelling concurrent engineering projects under uncertainty. *Concurrent Engineering.* **7**(3), 269-274.

Llewellyn, E.W., Stanton, M.J., Roberts, G.N. 2000. Towards the implementation of the Unified Modelling Language (UML) into a Computer Integrated Manufacturing (CIM) environment. *14th International Conference on Systems Engineering.* 12th – 14th September 2000. Coventry, UK, pp. 398 – 403.

Llewellyn, E.W., Stanton, M.J., Roberts, G.N. 2001. Discrete event systems design based upon the UML and Petri net objects. *3rd Workshop on European Scientific and Industrial Collaboration.* 27th – 29th June 2001. Twente, The Netherlands, pp. 211-219

Liu, T.I. and Yang, X.M. 1999. Design for quality and reliability using expert system and computer spreadsheet. *Journal of the Franklin Institute.* **336**(7), pp.1063–1074.

Lu, S.C.Y., Shpitalni, M. and Gadh, R. 1999. Virtual and augmented reality technologies for product realization. *CIRP Annals – Manufacturing Technology.* **48**(2), pp.471–495.

Meyer, Bertrand. 1997. *Object-Oriented software construction.* 2nd edn. London: Prentice-Hall. 0136291554.

Naylor, A. W. and Volz, R. A. 1987. Design of integrated manufacturing control software. *IEEE Transactions on Systems, Man, and Cybernetics*. **17**(6), pp. 881-897.

Prasad, B. 1999. A model for optimizing performance based on reliability, lifecycle costs and other measurements. *Journal of Production Planning and Control*. **10**(3), pp. 286–300.

Pressman, Roger S. 2004. *Software engineering: A practitioner's approach*. 6th edn. London: McGraw. 0071238409

Senin, N., Groppetti, R. and Wallace, D.R. 2000. Concurrent assembly planning with generic algorithms. *Robotics and Computer-Integrated Manufacturing*. **16**(1), pp. 65–72.

Stanton, M. J. 1999 .*Doctoral Thesis: Structured Petri Nets for the Design and Implementation of Manufacturing Control Software with Fault Monitoring Capabilities.* University of Wales College, Newport.

Swanstrom, F.M. and Hawke, T. 1999. Design for manufacturing and assembly (DFMA): a case study in cost reduction for composite wingtip structures. *International SAMPE Technical Conference Proceedings of the 1999 31st International SAMPE Technical Conference*. 23rd – 27th May 1999. Long Beach, California, pp. 101– 113.

Waldner, Jean-Baptiste. 1992. *CIM: Principles of Computer Integrated Manufacturing.* London: Wiley. 047193450X

WU, T. and O'GRADY, P. 1999. Concurrent engineering approach to design for assembly. *Concurrent Engineering Research and Applications.* **7**(3), pp. 231–243.

Chapter 2
Object Orientation for Manufacturing System Design

As noted in the title and first chapter this original work aims to establish a Computer Aided Software Engineering (CASE) methodology for the implementation of Computer Integrated Manufacturing (CIM). This will be achieved by developing a methodology based on the discipline of software engineering which is customised to meet the needs of manufacturing system designers. To fulfil the stated aim, this chapter establishes the elements of software engineering generally, and CASE specifically, which can provide benefits to the analysis, design and implementation of manufacturing systems. Current research literature surrounding the discipline of object-orientation is evaluated and its development is charted from the early methods through to the Unified Modelling Language (UML), which is the current de facto standard. The chapter analyses object-orientation as a method of achieving the requirement for manufacturing organisations to achieve a modular and incremental approach as outlined earlier in the first chapter. The benefits of an object-oriented analysis and design approach for the development of complex manufacturing systems are established and it will be proven that modularity in manufacturing can be achieved via modular decomposition utilising object-orientation. The chapter highlights how such an approach leads to the conceptual

integration of manufacturing systems thereby overcoming the problems associated with islands of automation. The need for an integrated approach to the analysis and design of manufacturing systems is discussed and the benefits of using an object-oriented methodology to achieve this are given.

2.1 Introduction

This work approaches manufacturing systems engineering as a software problem. This can be achieved by encapsulating hardware and software into a unified object-oriented framework. Software systems are not without their own problems, Smith *et al* (1999) estimate that for every six software systems that are completed two are cancelled and that the average software development project overshoots its budget and schedule by fifty percent. To challenge to overcome for software designers is to increase productivity and enhance quality. Holloway and Bidgood (1991) define these as:

1. **Quality** in practice means agreeing that each deliverable conforms to requirements and ensures that the end product will meet the customer's stated business objectives.
2. **Productivity** is the consistent application of an appropriate methodology, with its associated methods, techniques and deliverables so as to lessen the risk of wasting resources.

The development of control software for manufacturing systems faces the same challenges. From the literature it can be seen that:

- It is imperative to accurately capture user requirements and subsequently to ensure that the finished product fully meets those requirements;

- It is vital to ensure that a methodology is utilised that minimises resource wastage.

The established methods of software development have been labour intensive, error prone, slow and extremely costly. Much skill is needed to marry the business knowledge of users with the computer experience of analysts and programmers. As a result the developed systems have not always fully satisfied the needs of stakeholders. This is often due to a failure of the system design methodology, especially the user requirements gathering stage. Alternatively it can occur as a result of the lack of understanding of the methodology on behalf of the staff. What is needed is a methodology which is intuitive to understand for all users and which accurately captures requirements. The methodology also needs to ensure that the user requirements are refined precisely into the completed system.

Using traditional methods each program is individually designed, coded, tested and documented with the result that programs can only be maintained – debugged or updated – if their design and construction has been adequately documented by the original designer(s). Their logic is so interwoven that it can be impossible to unravel. In fact, it may be faster to throw a program away and start again, than to try and change it. This leads us to two main software related problems (Finkelstein, 1989):

1. **Maintenance problems** – Making a simple change in a program may lead to other changes. Furthermore, these changes may introduce errors that require even further change, so leading to yet more change and errors. Maintenance problems can be thought of as having a cascading effect;
2. **Communication problems** – The problem is, largely, due to a lack of effective communication. IT analysts, database administrators and programmers use computer jargon that is foreign to most users and management. Similarly, the day-to-day terminology of business matters may be unintelligible to many analysts and IT staff. This communication problem is compounded by the long lead-time before the developed systems are delivered to the users.

These problems must be addressed in the design methodology used. A loosely coupled system as discussed in this chapter will overcome these problems by decoupling modules so that changes have minimal impact. Communication problems can be resolved by having a graphical modelling tool which is intuitive for users at all levels. Such a tool needs to be able to break down the barriers between stakeholders of a technical and those of a non-technical background. However, the tool also needs to be capable, by iterations, of capturing all levels of system detail through to implementation.

2.2 Computer Aided Software Engineering (CASE)

Before a conclusive definition of Computer Aided Software Engineering (CASE) can be established it is crucial to understand the discipline of Software Engineering. Stevens (1991) states that it is "the process of inventing, improving and selecting among alternative solutions and then describing computer programs that meet users' requirements within the constraints of the environment and based on the chosen alternative." Once again the importance of user requirements is stressed and it is clear that the design methodology is the key to achieving this.

The term CASE relates generally to the automation of this software development process. Many CASE products are now available which aid in automating systems development and improving the productivity of analysts and programmers, sometimes by as much as two or three times. As stated earlier fifty percent of software projects go over time and budget so it is clear that a two or threefold improvement in productivity is a highly desirable feature. Historically, the most significant productivity increases in manufacturing or building processes have come about when powerful tools augment human skills. One man and a bulldozer can probably shift more earth in a day than fifty men working with hand tools. Automated tool support for software engineers should therefore lead to improvements in software productivity. CASE is now generally accepted as the name for

this automated support for the software engineering process (Finkelstein, 1989).

However, Sommerville (2006) noted that the first generations of CASE products have not led to the high level of productivity improvements which were predicted by their vendors. There are various reasons for this:

- Problems of managing complexity in the product and in its development process;
- Current CASE products represent 'islands of automation' where various process activities are supported to a greater or lesser extent;
- Adopters of CASE technology sometimes underestimated the training and process adaptation costs which are essential for the effective introduction of CASE. They often skimped on these costs with the consequence that the CASE technology was under-utilised.

Three different levels of CASE technology can be identified (Finkelstein, 1989):

1. **Production-process support technology**. This includes support for process activities such as specification, design, implementation and testing;
2. **Process management technology**. This includes tools to support process modelling and process management;

3. **Meta-CASE technology**. Meta-CASE tools are used to create production-process management support tools.

2.2.1 The Goal of CASE

To understand the goal of CASE it is vital to understand the objective of traditional design methodologies which are to produce consistent, high quality, implementable system specifications (Holloway & Bidgood, 1991). Sodhi (1991) states that the principal properties of a good software engineering system are that "the final product has achieved the software engineering objectives that meet the requirements and satisfy the customer." A good software product also has the following properties:

- **Functionality** – it works as the user requires;
- **Performance** – it achieves its tasks in the timescale required;
- **Economy** – it is as cost effective as possible;
- **Robustness** – it is useful for the maximum possible time;
- **Methodology** – it is based on a transparent and useful design methodology;
- **Documentation** – it has useful documentation;
- **User Interface** – it is user friendly;
- **External Interfaces** – it has well defined external interfaces.

Fisher (1991) describes how CASE tools can substantially reduce or eliminate many of the design and development problems inherent in medium to large software products. He proposes that the ultimate goal of

CASE technology is to separate design process from the actual code generation. If this is true then surely CASE tools are exclusive to the analysis and design stages of software/system development. However, it seems more appropriate that the goal of CASE is to enable the user to concentrate on the business aspects of the problem and reduce the technical complexity. Further CASE tools can be thought of as a complimentary element to the skill of the developer. CASE tools should enable projects to produce consistent, high quality, implementable systems. It can be seen, therefore, that the goal of CASE is to integrate with, and aid, the traditional software development process. To be effective CASE tools must, fit into and work with existing software and hardware.

2.3 *Formal Methodologies*

Software development requires a complex set of activities to be carried out some in sequence, others in parallel. Structured methodologies have evolved to provide assistance and direction to those involved in this process. A structured approach supplies a framework for action within which managers can manage and all participants can work constructively on specific activities which generate predetermined products (Holloway & Bidgood, 1991). As discussed earlier, in order to understand CASE, it is important to understand traditional structured methodologies. Several structured methodologies have been developed which provide a design framework as a set of formalisms and practices which have become the basis for software development. Although not perfect and largely relying on the thoroughness of the individual practitioner, these methodologies have allowed software developers to build more complex systems. Usually, these methodologies encourage the decomposition of large software systems into sets of smaller modules. The interfaces between these modules are well-designed by the software architect, allowing individual programmers to independently construct and test their respective assigned modules. Then during the final stages of software development process, all of the modules are integrated to form the final program (Fisher, 1991).

2.4 The Waterfall software development method

The Waterfall model was first introduced by W. Royce in 1970. It is the traditional model followed by most software developers and upon which most methodologies are based. The software evolution proceeds in an orderly sequence of transition from one phase to the next in linear order. It can be roughly subdivided into the following stages:

1. **Requirement Analysis.** Establish the user's requirements for the system. Produce models which aid in capturing user requirements that can aid in communication between stakeholders, designers and developers;
2. **Design Specification**. Compose a system blueprint, showing what to build and how to build it. Design specifications include module decompositions, data structure definitions, file format definitions, and important algorithm descriptions.
3. **Implementation**. Code, test and debug each module designed in the design specification;
4. **Unit Test and Integration**. A unit test is performed on each module built during the implementation phase; the modules are then integrated into a single program structure. The program as a whole is then tested to make sure the modules fit together and perform as designed;
5. **Maintenance**. Fix any bugs or problems found by users of the released version.

2.4.1 The problems with the waterfall development method

There are several problems inherent with the waterfall model as shown in Table 2-1 below (taken from Fisher, 1991):

Phase	Failure Symptom
Requirements Analysis	• No written requirements; • Incompletely specified requirements; • No user interface mock-up; • No end-user involvement.
Design Specification	• Lack of or insufficient design documents; • Poorly specified data structures and file formats; • Infrequent or no design reviews.
Implementation	• Lack of or insufficient coding standards; • Infrequent or no code reviews; • Poor in-line code documentation.
Unit Test & Integration	• Insufficient module testing; • Lack of proper or complete test suite; • Lack of an independent quality assurance group.
Maintenance	• Too many bug reports.

Table 2-1: The problems with the waterfall development method

To summarise these points it can be seen that:

- At the requirements analysis stage it is crucial to establish the user requirements fully. The finished product will fail if the requirements are not fully compliant with user requirements;
- The design specification needs to be constantly and consistency cross referenced with the user requirements to ensure they are fully met;
- Code needs to be well documented and adopt the relevant coding conventions to ensure subsequent maintenance, modification or upgrades are possible;
- Modules need to be tested individually and subsequently as part of the system;
- If all the stages above are completed the system will not suffer from extensive bugs at the maintenance stage.

2.4.2 The Benefits of CASE Tools

CASE tools need to be employed at the outset of the design and development process. In doing so the project should yield lower overall costs and better results in the implementation and maintenance phases. Design and development times will almost always be reduced by using CASE tools. But perhaps their most satisfying benefit comes in the form of insurance, or peace of mind, that the job is being done properly, on schedule and to the end-user's specification. CASE tools should aid in

establishing user requirements well before the implementation begins. However, much of the actual value depends on how well it is integrated into the organisation. To summarise the main benefits to be gained using CASE tools are (Fisher, 1991):

- A more complete Requirements Specification;
- More accurate design specifications;
- Up to date design specifications;
- Reduced development time;
- Highly extensible/maintainable code;
- Simplify. A major goal of CASE technology is to decompose requirements and designs into manageable components. Their function is to simplify, explain and reduce;
- Reduce costs by yielding higher quality specifications and designs;
- Produce quantitive and verifiable designs as each requirement in the software implementation must be verifiable and traceable back to the requirements document;
- Support Change;
- Show, not say. Good CASE tools present specification and design information visually.

2.4.3 CASE categories

CASE tools fall into several distinct categories (Fisher, 1991):

Upper CASE tools which are graphical tools for defining system requirements;

Lower CASE tools such as tools for developing prototypes;

Integrated CASE tools which are a combination of the above.

2.4.4 Reverse Engineering Tools

Reverse engineering is the exact opposite of engineering. It aims to go back from code and files to the original system design and thence to the system requirement. Reverse engineering tools must extract details of the essential business functions and data from ageing, but critical applications. This allows an organisation to salvage the investment of money and time, programming and database design skills, and user knowledge that built the original system. In essence, reverse engineering is a bottom-up process that has to assess the value and quality of existing systems by means of portfolio analysis.

2.4.5 Integrated Project Support Environment (IPSE) tools

IPSE products originally derived from software project management needs, and were based around software specification and project control methods. Typically, IPSEs provide the sort of project and configuration management facilities that most current CASE tools lack.

2.4.6 Project Management Tools

The degree to which CASE tools are available to support individual project management functions varies considerably. Tasks such as estimating and scheduling are well supported but others, such as risk management or the handling of actual contract conditions have little or no CASE help.

2.4.7 Verification, Validation and Testing (VV&T) Tools

Verification, validation and testing cover the processes associated with ensuring that a product is delivered correctly in the manner required, and that it meets its defined requirements.

Verification compares the output of each phase of the systems development life cycle with the requirements derived from the previous phase. The objective is to ensure that the deliverables produced by a phase fulfil all the requirements for that phase.

Validation checks that the original phase specifications were meaningful and appropriate, both in terms of the ultimate user requirements and in terms of the development methodology employed. In later stages of system development, this may be achieved by executing a working prototype and observing its behaviour. Testing is the process used to validate and verify.

2.4.8 Why do CASE tools fail?

Holloway and Bidgood (1991) have identified eleven main causes for the failure of CASE tools within an organisation:

- No methodology or standards in place
- Ignoring the importance of management
- Too much emphasis of CASE as the 'silver bullet' solution
- Confusion about what the CASE tool does
- Misuse of the tool
- Perception of CASE as a risk
- Unwillingness to change current methods
- Uncertainty, lack of consensus about what problem the CASE tool is trying to solve
- Poor integration of tools
- Inadequate functionality
- Poor documentation and training

2.4.9 Why do software projects fail?

Three main themes underlie software project failure (Fisher, 1991):

- Lack of complete requirements definition. If you lack a firm idea of what you are building, it is very difficult to build it right! Although plain common sense, this is often the most overlooked part of software development – identifying the system's requirements;

- No development methodology. Once you know exactly what you are going to build, you need to select and design techniques and establish implementation procedures. Following a formal methodology – a set of design techniques, development procedures, coding standards, checkpoints and work rules – helps ensure software design completeness and implementation quality;

- Improper design partitioning. An incomplete requirements specification leads to the development of the wrong software, but an improper design leads to low quality implementation. Design should be partitioned into manageable components and modules with formal pathways for importing and exporting data. Poorly partitioned designs lead to nightmarish code!

2.5 The Benefits of Object-Oriented Manufacturing Modelling

Before it is possible to discuss modelling in detail, it is firstly important to establish what is meant by the term modelling. From a computing perspective, a model is described by Booch *et al* (2005) as "a simplification of reality". Booch *et al* (2005) also state that "we build models so that we can better understand the system we are developing". When discussing complex systems they go on to state that "we build models of complex systems because we cannot comprehend such a system in its entirety". In engineering terminology a model is a "device that simply duplicates the behaviour of the system itself" (Cassandros & Lafortune, 1999). Whilst these meanings seem to have their basis in the same idea, there is a distinct difference between modelling in object-oriented terms and modelling in manufacturing engineering terms. In the latter, a model is a mathematical representation of a system whilst in the former it is a series of diagrams and their associated documentation.

So it can be seen that a model is a way of simplifying (abstracting) the necessary detail from a system in order to represent it in a diagrammatic or mathematical form. Generally, the resultant models are used to confirm user requirements, and aid in the communication of ideas between users and developers. Models enable developers to provide robust and yet

flexible solutions which are able to meet current user requirements and expand as their needs grow or change.

A full discussion of object-oriented techniques is available in (Booch, 1994, Pressman, 2004 and Sommerville, 2006). Here the focus is on the manufacturing application of object-oriented design. In a manufacturing environment machinery can intuitively be thought of as objects such as mills, lathes and so on (Adiga, 1993). The state variables of these objects will change at discrete points in time in response to events such as the completion of a machining operation. There is therefore a natural one-to-one correspondence between the physical items in the factory and the instances of software objects that represent them (Glassey and Adiga, 1989). Systems' design provides a formidable challenge which consumes large amounts of capital and human resources. The frequency of change in such a dynamic environment means that the design process needs to be highly flexible (Wong *et al*, 1999). This can be accomplished using object-oriented techniques where the design concentrates on reusability, reconfigurability and scalability.

Manufacturing systems are quite complex and varied in nature. It is impractical to imagine that a single solution or software package will address all the needs of all manufacturing firms. Therefore, a practical approach to designing software for managing a CIM system is to build generic solutions to the greatest possible extent, and then to customise

them to suit the needs of each firm. Thus, generic software object class libraries, customisable through sub classing, provide a good starting point in the design of practical software.

2.5.1 Abstraction issues

Manufacturing people think of their systems in terms of parts, conveyers, lathes or drilling machines etc. In other words they think in terms of 'objects'. An OO approach allows designers and programmers to construct software counterparts of manufacturing entities easily with little conceptual mismatch.

2.5.2 Simulation and control

Discrete event simulation has emerged as a powerful and popular tool for the analysis and design of manufacturing systems in the 1980s. Both simulation and control systems require a model of the real world. Ideally, one would like to be able to share for simulation purposes the state model developed for control purposes (or vice-versa), including both the structure and the data. Also, since simulation is quite popular as a tool used to validate control strategies, control modules implementing these strategies have to be developed. Again, sharing these modules between the simulation and the control tool can improve productivity and also the consistency of the application. OO techniques give a unique opportunity to develop a system that can be used initially as a simulation tool and, later as a production or control software. (Adiga, 1993)

2.5.3 Incremental Development

The technologies, finance or experience required to build an install CIM systems may not be present in all companies. This has led many people to believe that the most appropriate way to implement advanced manufacturing technology is in an incremental manner. Prototyping reduces the risk involved in implementing large products.

2.5.4 Customisation and maintenance

Pan, Tenenbaum and Glicksman (1989) identify two major shortcomings of current CIM systems: they are difficult to customise and maintain; and they have very limited problem solving and decision making capabilities. The first shortcoming can be addressed easily in an OO system. Individual objects can be customised through the sub classing enabled by the inheritance feature of OO (Adiga, 1993). For example if the application requires a representation of a lathe machine that is different from the one in the library supplied, a subclass Lathe can be created that inherits all the functionality of Lathe with additional methods to enhance its functionality. Similarly, an undesirable feature can be overridden through a re-implementation in the new subclass.

2.5.5 Complexity and variety

"Manufacturing systems are quite complex and varied in nature. It is impractical to imagine that a single solution or software package will address all the needs of all manufacturing firms. A practical approach to

designing software for managing CIM systems is to build generic solutions to the greatest possible extent, and then to customise them to the needs of each firm. Thus, generic software object class libraries, customisable through sub classing, provide a good starting point in the design of practical software." (Adiga, 1993)

2.6 Object Techniques for Modelling of Manufacturing Systems

One of the major aims of modelling a manufacturing system is to provide a view, or series of views, which can be interpreted by personnel at all levels of the system. At the shop floor level this may be the modelling of simple components, or groupings of components, which may perform a simple task. At the managerial level it may be necessary to examine the system from a higher level of abstraction, which ignores the detail that concerns lower levels.

A modular approach to manufacturing system design, therefore means a system can be viewed as a series of modules that can rapidly be combined to form a completely new manufacturing system. This offers many benefits to manufacturing system designers, most notably the utilisation of a series of well-designed reusable components can speed up the design process, ensure quality is kept at a maximum and can reduce the time it takes to design such systems.

Initial attempts at object-oriented modelling of manufacturing systems used entity relationship approaches to model systems such as that proposed by (Adiga and Gadre, 1990). However, the entity-relationship approach, while well suited to the modelling of static systems, has no facility for capturing the dynamic nature of such systems.

This problem persisted in other techniques such as the Object Oriented Modelling Process suggested by (Mize *et al*, 1992). Here the emphasis was placed upon the benefits of reuse, which are discussed later in this chapter. The software techniques in vogue at the time such as object modelling technique (OMT) and Object Oriented Analysis and Design (OOA/OOD) proposed by Coad and Yourdon (1990) all tend to have a software specific focus and do not lend themselves well to manufacturing systems without considerable modification. Such modification can lead to confusing and non-standardised designs which can actually slow down the development process and cause confusion amongst personnel involved with the system.

To improve the dynamic capabilities of object-oriented models many techniques have attempted to integrate state charts into OO modules However these improvised techniques do not allow the representation of some core aspects of object-orientation such as dynamic binding and polymorphism. (Wu, 2005).

2.7 The Unified Modelling Language for Manufacturing Systems

By their very nature manufacturing systems are extremely complex, with a wide range of interconnected objects and a myriad of messages passing between them. Designing manufacturing systems is further complicated by the individuality of each different system. It follows therefore, that manufacturing systems' needs cannot be met by 'off the shelf' packages. One solution is design generic solutions and then to customise them to the requirements of the company or application. The resultant generic object class libraries are customisable to the needs of the organisation through object-oriented techniques.

This abstraction of complex manufacturing systems into a series of objects is more intuitive than for many other systems as previously discussed. It is widely accepted that manufacturing systems need to be flexible, customisable and maintainable and this is effectively addressed in an object-oriented system where individual objects can be customised and updated using the key features of the technique (Adiga, 1993).

The Unified Modelling Language (UML) provides many elements which can aid manufacturing organisations including libraries of reusable classes and objects that can provide the 'building blocks' for new systems; inheritance which can simplify the development of new systems; and encapsulation through which a loosely coupled, modular approach can

reduce or even eliminate disruption to the rest of the system through incremental changes. The conceptual design of the system provides a further benefit to manufacturing managers in that it can allow for 'what-if analysis' to be carried out on a proposed system to establish viability, improve quality, or enhance production processes.

Ericksson *et al* (2000), describe the UML as an amalgamation of Grady Booch's, James Rumbaugh's and Ivar Jacobson's works standardised by the OMG (Object Management Group) in 1997. Later work by Ericksson *et al* (2004) extends the description of the UML to be a "free non proprietary language open to all but managed by the Object Management Group (OMG)". The purpose of the UML is to model systems in an object-oriented manner, bring together conceptual and executable artefacts and providing a language for human and machine. Erickson *et al* (2004) also go on to identify that the UML is an extensible language which can be modified to suit individuals or organisations.

Holt (2004), describes the UML as a general purpose modelling language originally developed for software development but which is also suitable for modelling other systems. The literature identifies thirteen UML diagrams classifying them into those that model what system is and those that show how a system behaves.

Bennett *et al* (2001), define the UML as a visual formal specification language used in the development of software systems and described the

language as having three rules which are: abstract syntax, well formedness and semantics which are expressed as diagrams. The work continues by explaining that the UML is not a programming language. Holt (2004) goes further by highlighting that the UML is not the answer to all modelling problems and that it is not a formal method.

From its definition, this work of Bennett *et al* (2004) suggests that UML is only for software which is contrary to other sources (such as Holt, 2004) which identify it as a systems modelling language. This is extended by Ericksson *et al* (2000) who describe how the UML has rules (syntax), meaning (semantics) but does not contain pragmatics (i.e. how to use it). This further reinforces the idea that the UML gives the system modeller the framework in which to develop but that it is not prescriptive and can be adapted.

The UML is a standardised modelling language consisting of a set of diagrams which have been developed to assist system developers accomplish the following: specification, visualisation, architectural design, construction, simulation, testing and documentation. The literature explains that the UML can model systems from different point of view utilising what the authors identified as twelve diagrams, categorised into static (structural), behavioural and interaction.

The UML has graphical elements which combined to form the models using some defined rules so serve the purpose of representing multiple

views of a system without showing how a system can be implemented. This author describes eight UML diagrams but further expresses that a system modeller does not have to use all of them in a given modelling problem, rather the modeller need to use only those that are required.

Maksimchuk *et al* (2005), describe the UML as being a standard visual modelling language for business process, work flow, sequence queries, application, database, and many more. Maksimchuk *et al* (2005) reiterate that the UML is a product of James Rumbaugh's Object Modelling Technique (OMT), Ivar Jacobson's Object Oriented Software Engineering Method and Gray Booch's Booch method, but adds that contributions were also made by many industry experts. He also goes on to praise how it has introduced commonality amongst professionals and goes further than other authors by stating that the UML can be used to model anything.

2.8 *Requirements for a CIM specific design methodology*

The following table outlines some of the issues from CIM implementation identified by Adiga (1993) along with their solutions in an OO design environment.

Problem	Solution
Staff involvement and co-operation	In order to achieve an accurate description of a system it is necessary for the analyst to obtain both bottom up and top down descriptions. The top down descriptions allow the analyst to perceive the system/sub-system under investigation in its wider context. The bottom down descriptions provide the necessary detail for an accurate model of the system/sub-system to be created.
Staff Training	As it is possible to model each subsystem and its external interfaces independently without regard to the detail of those interfaces, it is possible to incrementally implement changes. This means that staff training can also be implemented on an incremental basis.
Corporate Culture Not Right for CIM	Whilst the rigidity and inflexibility of many companies causes CIM to fail, the same rigidity can aid in the modelling of a system

	by providing a fixed system to model.
Human Resistance to change	The large involvement of users at all levels in the modelling of the system can lead to a feeling of "inclusion" on behalf of the staff. In general staff are more likely to wish to see a project, in which they have played a part, succeed.
Failure to encapsulate all departments	When modelling the detail of subsystems it is apparent which systems are immediately interactive with the current one. This means that it is unlikely that a system will be overlooked. In addition it may not be prudent or necessary to include all departments / systems in the new model. The idea of incremental implementation allows the selected implementation of changes. Therefore the system can either completely overlook departments or systems, or can allow for their inclusion at a later stage.
Attempting to implement using unsuitable methodologies and guidelines	The proposed guidelines and methodologies aim to overcome any failings with the more traditional areas of analysis and design in manufacturing.
Inability to implement conceptual design	The addition of an interface definition language (IDL) to the model will allow for automatic code generation of the control code for the component.
Failure to streamline	The need to understand and model the

processes	workings of even the most basic component in the system under review should lead to an optimised approach to the actions and operations of the system.
Inadequate planning and design	The integrated method proposed comprises a detailed set of diagrams which encompass all necessary planning and design aspects.
Inadequate analysis of user needs	The proposed method is very much user driven.
Time	The methodology allows for the whole system to be re-designed on a modular basis which allows for components to be upgraded as and when time permits.
Cost	As above.

Table 2-2: CIM implementation problems that can be overcome with OO design

2.9 *The Key Benefits of Object-Orientation*

The OO paradigm represents a different way of looking at the program modules. It defines program modules as a package of data and procedures named an 'object': i.e. an abstraction of private data and operations that are naturally associated together. This abstraction facility enables real-life factories and their complex interacting components to be represented as objects such as a machine or a part quite close to reality.

Data is stored in locations, i.e. instance variables, which cannot be directly accessed by other objects. Procedures are commonly known as 'methods'. Each procedure (or method) defines the behaviour expected of the object. One such behaviour is to change (or return) the data stored in its instance variables. Objects interact by sending one another messages. Typically, receipt of a message activates a corresponding method in the receiving object.

Sending a message to an object is similar to asking it to perform an operation on itself and return the result to some place or the requesting object.

2.9.1 Object Communication

Inter-object communication - the simplest model of communication between objects involves two objects where the sender of a message needs to know the identity of the receiving object, but not vice-versa.

However, the information flow may be bi-directional, i.e. the receiving object may return information of interest to the sender. This return may be in the form of a value or result sent automatically by the receiving object. Alternatively, the receiving object may send a reply message.

2.9.2 Concurrency and synchronisation

Concurrent systems consist of independent activities (or processes) that must communicate and synchronise in order to achieve some common goal. Two different methods are used in handling concurrency and synchronisation issues in OO approaches. In the first one, an object management system controls and synchronises access to objects. Thus individual objects be regarded as 'passive' objects. In the other approach, the objects are 'active'. With active objects, there is no need for an explicit synchronisation mechanism as the objects themselves decide when they are ready to receive a message. An active object is one that has an independent thread of control, i.e. it has control over the execution of computation required. It can monitor events that occur during an event and take action autonomously. This allows for asynchronous behaviour at the program level.

2.10 Object-Orientation in the Systems Life-Cycle

The property of encapsulation makes it possible to have software objects that directly correspond to the physical entities in a manufacturing system such as machines, operators, lots etc. Therefore, the entities that the users and software engineers discuss, when defining the requirements as part of the project, are the same entities designers build objects from and programmers work with during implementation of the requirements. This helps to make a smooth transition from requirements to design and implementation.

2.10.1 Implementation

Since program development follows the abstraction process, it allows software objects to be developed in parallel after the interfaces are defined. This is possible because the implementation details of one object are independent of other objects.

2.10.2 Testing

Since an OO application contains clearly defined and separately identifiable modules, these can be tested one at a time. Separate testing of individual objects before they are put into one system helps to localise errors.

2.10.3 Maintenance

Encapsulation restricts any undesired side effect from changing the contents of any object's data. Since all the data and procedures related to an object are located in one place, changes to be made are confined to one location. Apart from preventing any accidental corruption of the data, this feature helps in both the maintainability and the extendibility of software.

Use of explicit communication through messages and polymorphism allows the use of entirely new classes of objects in an existing application, as long as they follow the same message protocol as the application.

2.10.4 Prototyping and Software Evolution

The flexibility offered by an OO approach presents some special advantages for prototyping. Since one object can be treated like any other as long as the two have the same message protocol, we can build large complex systems from smaller interchangeable ones. But unlike the conventional approaches the initial prototype need not be thrown away; it can be 'grown' into the full production system.

2.10.5 Software Reuse

The basic OO concepts and implementation techniques support the development of software objects that can be reused in more than one application.

2.11 Meyer's Five Criteria for Modularity

Meyer's (1997) criteria are used in this work to evaluate the extent to which the method develop satisfies the requirements of modularity. Meyer (1997) states that "A software construction method is modular if it helps the designers produce software systems made of autonomous elements connected by a coherent, simple structure." A modular design should satisfy the following five fundamental requirements:

1. Decomposability
2. Composability
3. Understandability
4. Continuity
5. Protection

2.11.1 Modular Decomposability

A software construction method satisfies modular decomposability if it helps in the task of decomposing a software problem into a small number of less complex sub-problems, connected by a simple structure, and independent enough to allow further work to proceed separately on each of them. Once a system is decomposed into subsystems it should be possible to distribute work on these subsystems among different people or groups.

2.11.2 Modular Composability

A method satisfies Modular Composability if it favours the production of software elements which may then be freely combined with each other to produce new systems, possibly in an environment quite different from the one in which they were initially developed

2.11.3 Modular Understandability

A method favours Modular Understandability if it helps produce software in which a human reader can understand each module without having to know the others, or, at worst by having to examine only a few of the others. A method can hardly be modular if a user is unable to understand its elements separately.

2.11.4 Modular Continuity

A method satisfies Modular Continuity if, in the software architectures that it yields, a small change in a problem specification will trigger a change of just one module, or a small number of modules.

2.11.5 Modular Protection

A method satisfied Modular Protection if it yields architectures in which the effect of an abnormal condition occurring at run time in a module will remain confined to that module, or at worst will only propagate to a few neighbouring modules.

2.12 Five Rules for Modularity

From the preceding criteria, five rules follow which must be observed to ensure modularity:

1. Direct mapping
2. Few Interfaces
3. Small interfaces (weak coupling)
4. Explicit interfaces
5. Information hiding

2.12.1 Direct Mapping

Any software system attempts to address the needs of some problem domain. If you have a good model for describing that domain, you will find it desirable to keep a clear correspondence (mapping) between the structure of the solution, as provided by the software, and the structure of the problem, as described by the model. Hence the first rule: The modular structure devised in the process of building a software system should remain compatible with any modular structure devised in the process of modelling the problem domain.

2.12.2 Few Interfaces

Every module should communicate with as few others as possible.

2.12.3 Small Interfaces

If two modules communicate, they should exchange as little information as possible.

2.12.4 Explicit Interfaces

Whenever two modules A and B communicate, this must be obvious from the text of A or B or both. If a module is decomposed into several sub-modules or needs to be composed with other modules, any outside connection should be clearly visible. It should be easy to find out what elements a potential change may effect.

2.12.5 Information Hiding

The designer of every module must select a subset of the module's properties as the official information about the module, to be made available to authors of client modules. That is that the rest of the world through some official description or public properties knows every module.

2.13 Software Reuse

Software development with reuse is an approach to development which tries to maximise the reuse of existing software components. An obvious advantage of this approach is that overall development costs should be reduced. Fewer software components need to be specified, designed, implemented and validated. However, cost reduction is only one potential advantage of reuse. Systematic reuse in the development process offers further advantages (Sommerville, 2006):

- **System reliability is increased.** Reused components, which have been exercised in working systems, should be more reliable than new components. These components have been tested in operational systems and have therefore been exposed to realistic operating conditions;
- **Overall process risk is reduced.** If a component exists, there is less uncertainty in the costs of reusing that component than in the costs of development. This is an important factor for project management as it reduces the uncertainties in project cost estimation. This is particularly true when relatively large components such as sub-systems are reused;

- **Effective use can be made of specialists.** Instead of application specialists doing the same work on different projects, these specialists can develop reusable components which encapsulate their knowledge;
- **Organisational standards can be embodied in reusable components.** Some standards, such as user interface standards, can be implemented in a set of standard components. For example, reusable components may be developed to implement menus in a user interface. All applications present the same menu formats to users. The use of standard user interfaces improves reliability, as users are less likely to make mistakes when presented with familiar interfaces;
- **Software development time can be reduced.** Bringing a system to market as early as possible is often more important than overall development costs. Reusing components speeds up system production because both development and validation time should be reduced.

2.14 Chapter Summary

The methodology proposed in this work will utilise Hardware/Software Objects to enable manufacturing system designers to apply the concepts of software engineering.

The chapter has established that software systems are not perfect, indeed two out of eight software projects fail and fifty percent are over time and budget. However, the reasons for these failures have been established as primarily due to the lack of understanding at the user-requirements stage of the design methodology.

It is clear that the importance of selecting the most appropriate design methodology is paramount in any successful system implementation.

From the literature it has been established that a successful design methodology should:

- Accurately capture user requirements in a manner which can be understood by all stakeholders. Each stage in the design process must constantly and consistently cross reference user requirements to ensure they are fully met;
- The final system will fail if it does not adequately meet user requirements;

- Support iterative refinement of user requirements into low level technical detail for implementation. This can be achieved by a hybrid top-down/bottom-up approach;
- The methodology must endeavour to minimise resource wastage. Utilising off the shelf, pre-tested components from a library of objects can achieve this goal;
- The development technique must support loose coupling of objects and should allow for modular decomposition;
- Code needs to be well documented and adopt the relevant coding conventions to ensure subsequent maintenance, modification or upgrades are possible;
- Modules need to be tested individually and subsequently as part of the system.

The chapter also highlights some important benefits to manufacturing organisations for adopting an object-oriented design methodology:

- Manufacturing personnel already think of their systems in terms of objects and therefore an OO approach should prove to be intuitive;
- Simulation techniques are useful for validating control strategies and for generating software;
- Incremental development approaches reduce costs;
- OO systems utilising class libraries offer customisation opportunities and aid in system maintenance;

- Object class libraries that can be reused in other systems aid in breaking down the complexity of manufacturing system design.

These important points and benchmarks drawn from the literature will be used in this work to validate the original methodology devised in this work against the requirements of manufacturing organisations.

Many system modellers face the inimitable problem of having to cope with the recurrent need to become experts in a range of disciplines other than their own. For example, a computer system's analyst may need to analyse and design a software system for a petrochemical company, or an information system specialist may need to develop a new system for a supermarket chain. This implies the need for rapid personal knowledge expansion, however in reality the system modeller relies on an intuitive and highly detailed progression of models which enable them to overcome the barriers and bridge the gap between those with a dedicated knowledge of the system under consideration and those with the specialist skills needed to develop the new system. In short, system modellers need models which facilitate communication between the stakeholders at all levels within the system and those undertaking the development. These models need to be intuitive enough for all parties to understand and yet contain enough expressive power to enable the analysis and design of the system under consideration, in iteratively more complex levels of detail. System modelling tends to fall into two main camps, the slow and

methodical, but high quality methods such as SSADM, SDLC etc. and the fast, low cost and low quality methods, such as RAD, Code and fix etc. Some main points can be drawn in this chapter:

- ***Reduced costs inherent in an incremental implementation.*** Customising the solution over time means reduces costs by starting with the essential features and functionality and customising based on priority;

- ***Staged building.*** Building in stages allows the project to be broken into smaller, more manageable pieces giving staff the time to adapt to the new system and facilitating team development;

- ***Rapid Value.*** Implement small steps which have the most dramatic impact rather than redesign the whole system in one go;

- ***Stakeholder Involvement.*** Include staff at all levels in the development process as they have much to contribute and early and repeated involvement will ensure they take ownership;

- ***Results-based decisions.*** Make decisions for future enhancements based on actual results of previous phases;

- ***Milestone measurements.*** Goals and measurement criteria are should be defined prior to each new phase of implementation.

References

Adiga, S., & Gardre, M. 1989. Object-Oriented Modeling of Flexible Manufacturing Systems. *Journal of Intelligent and Robotic Systems.* **3(1)**, pp. 147-165.

Adiga, S. 1993. *Object-oriented software for manufacturing systems.* London: Chapman Hall. 0412397501.

Bennett, S., Skelton, J. and Lunn, Ken. 2004. *Schaum's Outline of UML. 2nd edn.* London: McGraw-Hill. 0077107411.

Booch, Grady. 1994. *Object-oriented analysis and design: with applications.* 2nd edn. Benjamin/Cummings: Menlo Park, Ca. 0805353402.

Booch, G., Rumbaugh, J. &Jacobson, I. 2005. *The Unified Modeling Language User Guide. 2nd edn.* USA: Addison Wesley Longman. 0321267974

Cassandras, C.G. and Lafortune, S., 2007. *Introduction to Discrete Event Systems.* 2nd edn. London: Springer. 0387333320

Coad, P. and Yourdon, E., 1990. *Object-Oriented Analysis.* 2nd edn. Michigan: Prentice Hall. 0387333320

Eriksson, Hans-Erik and Penker, Magnus. 2000. *Business Modelling with UML.* London: Wiley Computer Publishing. 0471295515

Eriksson, Hans-Erik, Penker, Magnus, Lyon, Brian and Fado, David. 2004. *UML 2 Toolkit.* London: Wiley Publishing. 0471463612

Finkelstein, C. 1989. *An Introduction to Information Engineering.* Wokingham, UK: Addison-Wesley. 0-201-41654-9.

Fisher, A. 1991. *CASE Using Software Development Tools.* 2nd end. Chichester, UK: John Wiley & Sons. 0-471-53042-5.

Glassey, C.R. & Adiga, Sadashiv. 1989. Conceptual Design of a Software Object Library for Simulation of Semiconductor Manufacturing Systems. *Journal of Object Oriented Programming.* **11**(1) pp. 39-42.

Holloway, S., & Bidgood, T. 1991. *Case Handbook for Information Managers.* Aldershot, UK: Avebury Technical Academic Publishing Group. 1-85628-189-2.

Holsing, N. F., & Yen, D. C. 1997. Integrating Computer-Aided Software Engineering and Object-Oriented Systems: A Preliminary Analysis. *International Journal of Information Management.* **17**(2), pp. 95-113.

Holt, John .2004. *UML for Systems Engineering: Watching the Wheels.* 2nd edn. Institute of Electrical Engineering, London. 0863413544

Maksimchuk, Robert A., Naiburg, Eric J. and Brown, Alan. 2005. *UML for Mere Mortals.* London: Pearson Education. 0321246241.

Meyer, Bertrand. 1997. *Object-oriented Software Construction*. 2nd edn. London: Prentice-Hall. 0136291554.

Mize, J. H., Bhuskute, H. C., Pratt, D. B. and Kamath, M. 1992. Modeling of Integrated Manufacturing Systems Using an Object Oriented Approach. *IIE Transactions.* **24**(3), pp. 14-26.

Pressman, Roger S. 2004. *Software Engineering a practitioner's approach*. 6th Ed. London: McGraw-Hill. 0071238409.

Sodhi, J. 1991. *Software Engineering Methods, Management and CASE tools*. USA: TAB Professional and Reference Books. 0-8306-3442-8.

Sommerville, I. 2006. *Software Engineering*. 8th end. London, UK: Addison-Wesley. 03211313798.

Stevens, W. 1991. *Software Design Concepts and Methods*. London, UK: Prentice-Hall. 0-13-820242-7.

Wong, S. T. W., Mak, K. L. and Lau, H. Y. K. 1999. An Object-Oriented Model for the Specification of Manufacturing Systems. Computing & Industrial Engineering, **36**(1). pp. 655-671.

Chapter 3
Petri Nets for Functional Modelling

The aim of this chapter is to analyse, describe and evaluate Petri net graphs along with the various techniques for extending their modelling capabilities. This will enable the identification of the optimum model for this work which offers the best combination of analysis and modelling power along with visual simplicity. The Petri net model thus identified will then be used as the basis of the combined Petri net/object-oriented modelling technique presented in Chapter 4. The chapter begins by outlining the fundamental concepts underlying Petri net theory and demonstrates their applicability to the modelling of asynchronous concurrent systems. The various attempts at extending the analysis and modelling power of Petri nets are presented in a comprehensive literature review before the chapter concludes with a summary of the strengths and weaknesses of each technique and a justification of the chosen technique. This chapter offers a minor contribution to knowledge by presenting a comprehensive and up-to-date literature review of the research activity into Petri net theory.

3.1 Introduction

A Petri net can be described as an abstract, formal model of information flow within a system, particularly those that exhibit asynchronous and concurrent behaviour (Peterson, 1981). Research into this field is primarily concerned with the search for simple, yet powerful, methods for describing and analysing the flow and control of information in such systems. In addition to being suitable for the description of the dynamic changes within a system, Petri nets can also describe the state of individual components within that system at any period of time, though time is generally not explicitly modelled with Petri nets. This concept of modelling state changes with dynamic events can be thought of as allowing a description, *via* models, of the behaviour of the system. As systems increase in their complexity the problem of multiple parallel or concurrent activities needs to be considered and these can be effectively addressed in a Petri net based model as will be demonstrated in the remainder of this chapter.

3.2 Petri net graphs for modelling static systems

A Petri net graph allows the static properties of a system to be modelled by using two types of nodes: places (represented by circles) and transitions (represented by solid bars). The connection between nodes is made by directed arcs, such that a directed arc can link a place to a transition or *vice-versa*. If a node is directed from node *i* to node *j*, then *i* is an input to *j*, and *j* is an output of *i*. Figure 3-1 shows a simple Petri net graph where P_1 is an input to t_2, and t_2 is the output of P_1.

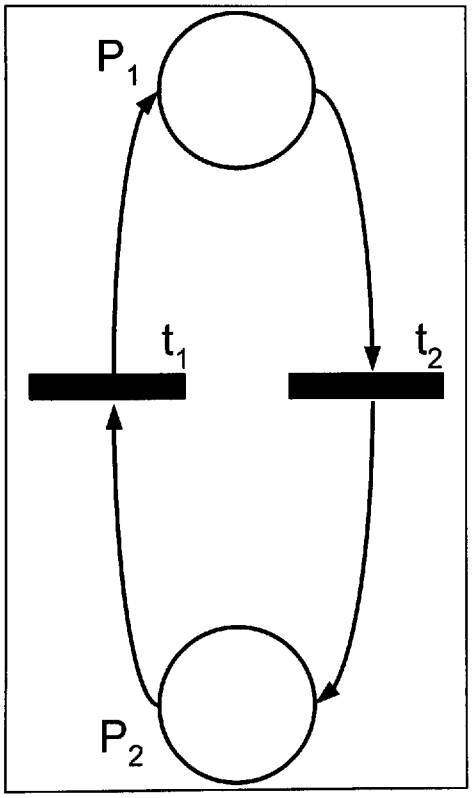


Figure 3-1: A simple Petri net graph

3.3 Marked Petri nets for modelling system behaviour and dynamics

In addition to the static properties represented by the graph, a Petri net has dynamic properties that result from its execution. This execution is controlled by the movement of tokens within the Petri net, thereby modelling the changes of state in the system. Tokens are represented by black dots which reside in the places of the net. A Petri net which has tokens is described as a marked Petri net (Peterson, 1977).

The execution of a transition is called firing and is facilitated by its input places being marked with tokens. If all the input places to a transition are marked, the transition is said to be enabled. In Figure 3-2 (a), transition t_2 is enabled as its only input (P_1) is marked.

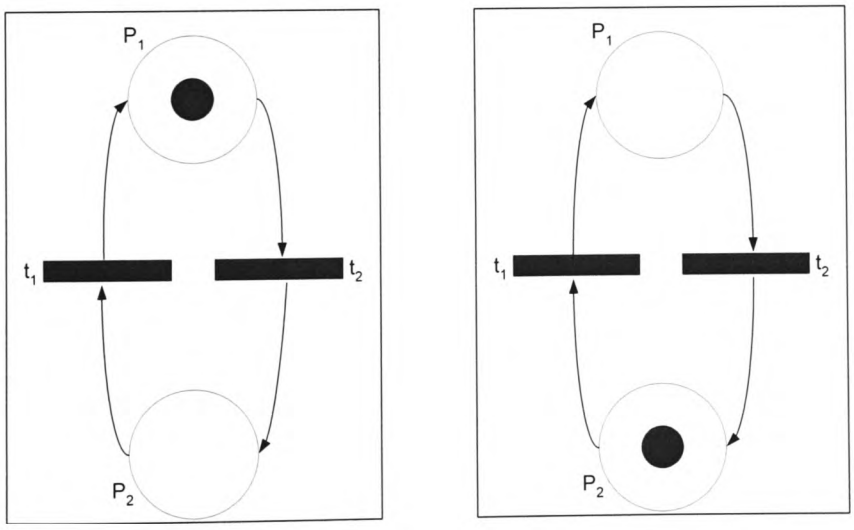


Figure 3-2: (a) A marked Petri net (b) The result of t1 firing

Upon firing an enabled transition moves the tokens from its input places and deposits them in its output places as shown in Figure 3-2 (b). The distribution of the tokens in a marked Petri net defines the state of the net and is called its marking.

3.4 Conflict

A conflict occurs whenever the firing of one transition disables another. Figure 3-3 from (Peterson, 1977), demonstrates a conflict. Transitions t_3 and t_5 are enabled, therefore either one can fire, the choice as to which should fire first is arbitrary. However, if transition t_3 fires then transition t_5 will no longer be enabled conversely if transition t_5 fires then transition t_3 will be disabled.

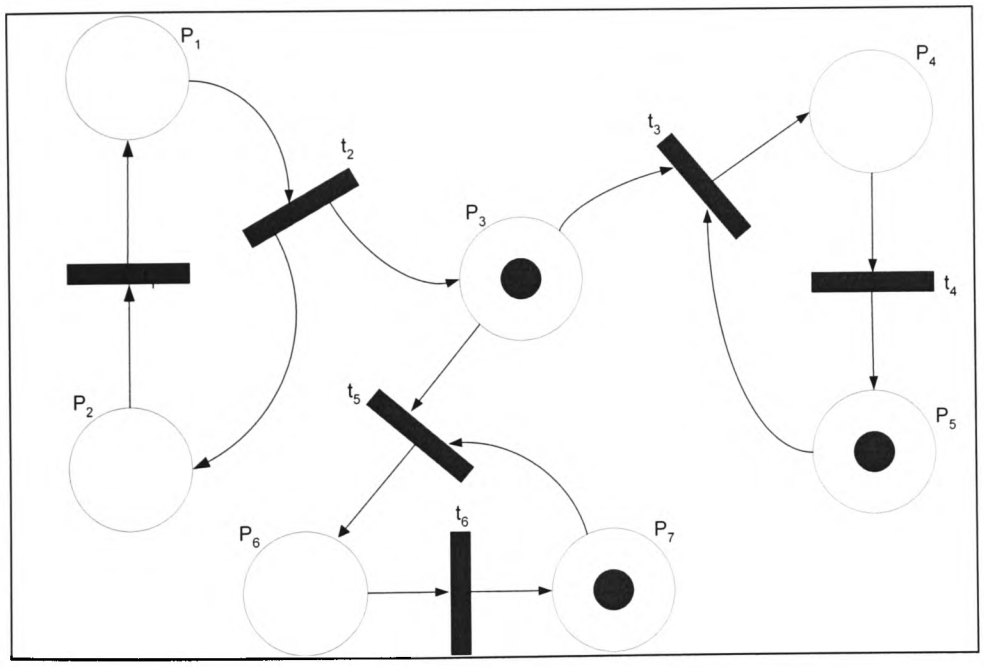


Figure 3-3: A Marked Petri net with conflict

3.5 Modelling with Petri net graphs

3.5.1 Uninterpreted models

A Petri net is considered to be an uninterpreted model however, in order to make the Petri net graph more intuitive it is sometimes useful to label the nodes with system specific information. The labels have no effect upon the execution of the net and their purpose is simply to make the models more visually intuitive to the reader.

3.5.2 Sinks and Sources

An example is given, in Figure 3-4 of a system whose purpose is to provide raw material from an external source to a likewise external sink. Raw materials enter the system from the source and enter a storage unit. When the raw material and the arm are both ready, the material is moved from storage and placed onto the waiting conveyer belt by the arm. Once the conveyer is loaded it is moved along whereupon the material arrives at its source and the arm once again becomes ready.

A transition can be defined as a source if it has an output place but no input. The sink transition is the reverse in that it has an input place but no output. The concepts of sink and source transitions are intuitive and therefore no further detail will be described here, however, the concept of a transition being, fired from or firing, an external entity is important to note as it provides the basis of the modular concept presented in this thesis.

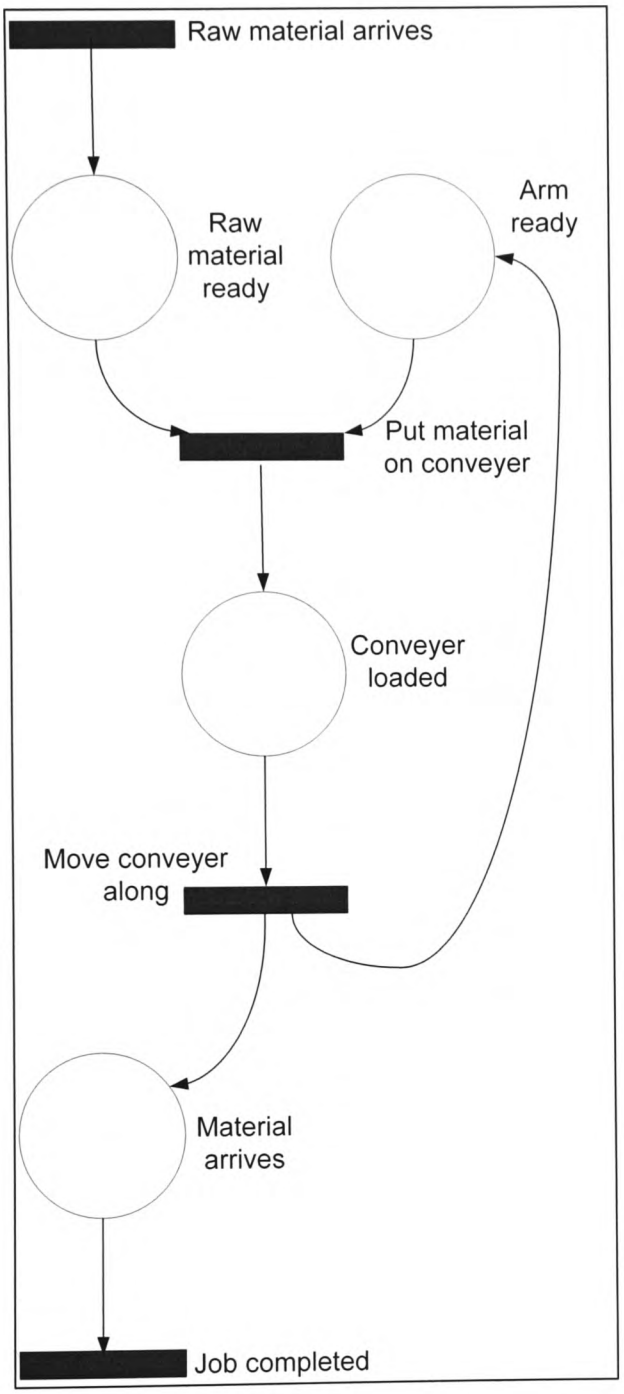


Figure 3-4: Modelling concurrency with Petri net graphs

3.5.3 Concurrency

Figure 3-4 also demonstrates the concurrency inherent in the system which is composed of two main independent entities, the raw material and the robot arm. If required, it would be possible to model the events which relate solely to the one or the other. Raw materials may enter or leave the system independent of the action of the robot arm, there is no need to synchronise these two entities. However, in order to begin the process of loading the conveyer belt with raw material, both entities need to be available and this can also be modelled. Thus a Petri net is ideal for modelling systems of distributed control with multiple processes occurring concurrently.

3.5.4 Asynchronicity

The basic Petri net graph contains no inherent measure of time which reflects the philosophy that the only important property of time, from a logical point of view, is in defining a partial ordering of the occurrence of events. In real life situations, events will take a variable amount of time and this is reflected in Petri net models by not depending upon a notion of time to control the sequence of events. Instead a Petri net structure contains all the information necessary to define the possible sequences of events of a modelled system (Peterson, 1977).

3.5.5 Non-Determinism

A Petri net, in common with the systems modelled by them, can be viewed as a series of discrete events. Any particular order of occurrence modelled is generally one of many possible allowed by the basic structure. This leads to non-determinism in the execution of a Petri net, whereby if at any time more than one enabled transition may fire, the choice as to which to fire is made in a nondeterministic (randomly or by un-modelled forces) manner (Peterson, 1977). It is important to model all event sequences which are possible in real life with no regard to those that whilst possible, are not available in the real system under consideration. This is to ensure that account is taken in the modelling process of forbidden states, that is states that the system should never achieve. Knowledge of these forbidden states is crucial to the novel concept of behavioural constraints which have been developed in this thesis.

The firing of a transition is considered to take zero time, i.e. to be instantaneous and since time is a continuous variable, the probability of any two or more events happening simultaneously is zero, therefore in reality two transitions are unlikely to fire simultaneously (Peterson, 1977).

3.6 Petri Net Analysis

Petri nets are composed of places *P* and transitions, *T*. Places are connected to transitions *via* their input, *I* and outputs, *O*. An input and output for a place is always a transition whilst conversely the input and output for a transition is always a place. A set of inputs is defined for each transition $I(t_j)$, accordingly the set of outputs is also defined for each transition $O(t_j)$. A Petri net (*C*) is formally defined as a four tuple where $C=(P, T, I, O)$.

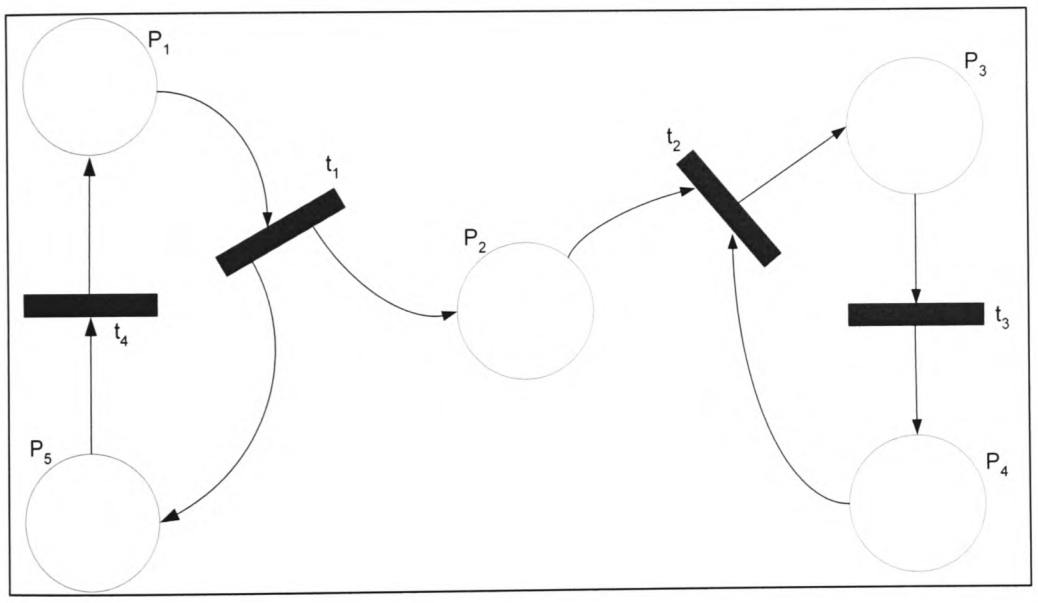


Figure 3-5: Petri net graph

The Petri net graph in Figure 3-5 can be structurally defined as:

$P = \{p_1, p_2, p_3, p_4, p_5\}$

$T = \{t_1, t_2, t_3, t_4\}$

The inputs are defined as:

$I(t_1) = \{p_1\}$

$I(t_2) = \{p_2, p_4\}$

$I(t_3) = \{p_3\}$

$I(t_4) = \{p_5\}$

The outputs are defined as:

$O(t_1) = \{p_2, p_5\}$

$O(t_2) = \{p_3\}$

$O(t_3) = \{p_4\}$

$O(t_4) = \{p_1\}$

A number of features enable a Petri net graph to be analysed and each of these is briefly reviewed below. For a more detailed analysis readers are referred to the work of Peterson (1977).

3.6.1 Marking

Markings are used to show the distribution of tokens in a Petri net graph and are represented by μ. Markings are visualised utilising binary representations which aid in establishing the current state of places in the system. Referring to Figure 3-3 it can be seen that the system marking can be represented as: $\mu = \{0,0,1,0,0,0,0\}$ which represents of the six available places only P_3 is enabled (marked) whilst the other five places are disabled. A marked Petri net would modify the structure defined in section 3.6 to: $M = (P, T, I, O, \mu)$.

The stage of marking of a Petri net graph is shown by $\mu_0 \ldots \mu_n$ with μ_0 representing the initial marking. For Figure 3-5 the marking tree would read as follows:

$\mu_0 = \{1,0,0,1,0\}$

$\mu_1 = \{0,1,0,1,1\}$

$\mu_2 = \{1,0,1,0,0\}$

$\mu_3 = \{0,1,0,1,1\}$

The system would now continually loop through its actions.

Understanding the state of an object is useful for aspect to the modelling of behavioural constraints and is discussed in more detail in Chapter 4.

3.6.2 Reachability

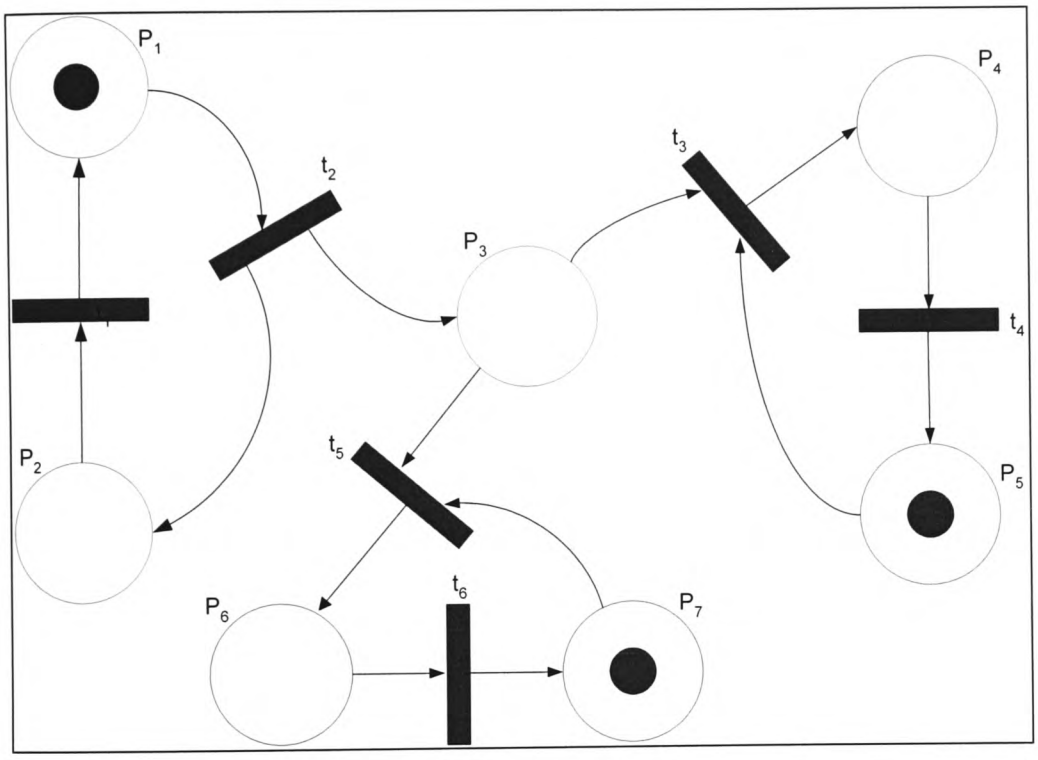


Figure 3-6: A Petri net graph in its μ_0 marking

The marking for Figure 3-6 demonstrates how the reachability tree can branch depending upon which transition fires in a Petri net with conflicts. At its μ_1 marking the graph provides two alternatives a) to fire t_5 or b) to fire t_3 as depicted in Table 3-1.

	Marking	Alternative A (fire t_5)	Alternative B (fire t_3)
μ_0	{1,0,0,0,1,0,1}		
μ_1	{0,1,1,0,1,0,1}		
μ_3		{1,0,0,0,1,1,0}	{1,0,0,1,0,0,1}
		Return to μ_1	Return to μ_1

Table 3-1: Two alternatives for a Petri net graph with conflicts

3.6.3 Boundedness and Safe nets

A Petri net in which any place can only have one token is called a safe net (Srihari *et al*, 1990). Boundedness refers to the maximum number of tokens that a place can hold. In the context of a bounded-ness a safe net is considered a 1-bounded net (Peterson, 1977).

3.6.4 Conservativeness

Conservative Petri nets are useful for modelling situations where tokens represent resources. In a conservative net tokens are neither created

nor destroyed. This means that the number of inputs to a transaction is equal to the number of outputs (Peterson, 1977).

3.6.5 Liveness

A transition can have three possible firing states. It is live if it can be fired in all reachable markings; it is potentially fireable if at least one marking enables it; and it is dead if no markings can enable it. A Petri net is live if "all transitions in the net are enabled by a single cycle of token movement" (Srihari *et al*, 1990).

3.6.6 Properness

A Petri net is said to be proper if the initial marking is reachable from all the markings in the reachability set (Cecil *et al*, 1992).

3.6.7 Decision Free

A Petri net is decision free if there is a single input arc and a single output arc from each place.

3.6.8 Timed transitions

A timed transition is represented by a rectangular box.

3.6.9 Inhibitor arcs

An inhibitor arc will not fire if the place to which it is linked contains a token.

3.6.10 Weighted arcs

A transition associated with a weighted arc will only fire if the number of tokens in the place associated with the arc is equal to or greater than the weighted value. Upon firing, the weighted number of tokens is removed from each associated input place. When weighted arcs are used to link output places and transitions, tokens corresponding to the weighted value are added to the respective output places (Cecil *et al*, 1992).

3.6.11 Deadlock

Deadlock occurs when a net reaches a marking from which no transition can fire (Agerwala, 1979).

3.7 Petri Net Analysis Methods

Petri nets can be analysed in numerous ways but generally their analysis falls into three specific categories:

3.7.1 Reduction or decomposition methods

To aid in the analysis of large scale, complex systems, techniques are employed to reduce this complexity and size while preserving the properties of the system that are required for analysis. Murata (1989) indicates that "reduction techniques are powerful but only applicable to special subclasses of Petri nets." This would imply that they are particularly useful for decomposing more abstract nets, or removing the non-essential properties for specific criteria investigation, but are generally not applicable to all types of net or the entire net.

3.7.2 Matrix equations

Matrix equations employ mathematical equations or algebraic expressions to study the dynamic behaviour of the net. Murata (1989) states that "matrix equations govern the behaviour of concurrent systems modelled by Petri nets". However, he recognises the limitations of these techniques and maintains that "the solvability of these equations is limited due to the inherent non-deterministic nature of Petri net modules". The text also points out that, as with reduction

techniques, these methods are not employable with all types of net and are more useful on smaller subclasses of net.

3.7.3 Reachability tree method

The reachability tree method begins from the initial marking of the net and for each firing of a transition produces a tree diagram of all possible markings within the net.

This type of analysis is useful for a number of reasons. Firstly, it will allow identification of unreachable places within the net which, in turn, identifies errors in the model. Other errors in the model will also be identified through the identification of transitions where the pre-requisite conditions for enabling them can never occur. Secondly, the analysis can aid in the removal of conflict and confusion in the net as the basic tree can only be generated based upon the assumption that the net is pure and free from conflict.

Usefully, in an ordinary net, where the arc weightings are all one, this tree structure could possibly be use as a model for a decision structure when the net is used to module software. Each branch may represent an outcome of a decision loop and the resultant marking used to represent Boolean statements executed on the condition of the branch.

3.8 Literature review of Petri net extensions

3.8.1 A Brief history of the development of Petri nets

Petri nets were developed by Dr Carl Adam Petri (Petri, 1962) as a method for the design and programming of information processing machines. Petri's work was subsequently adopted by the Massachusetts Institute of technology (Peterson, 1977) where it became known as Petri net theory. Stochastic Petri nets are nets in which random firing delays associated with transitions. They are a mathematical model for description of phenomena with a probabilistic nature that usually is time related (Marsan, 1989).Much work has been undertaken to use Petri nets for the design and implementation of a number of types of systems such as Flexible Manufacturing Systems (Chaillet *et al*, 1993)).

3.8.2 Application to Manufacturing Systems

Peterson (1981) describes Petri nets as "a tool for the study of systems". Cecil *et al* (1992) note that the "ever increasing application of Petri nets in the modelling of manufacturing systems testifies to their research potential and modelling capabilities".

The earliest application of Petri nets was as part of the project MAC a Masters thesis (Hack, 1972) which dealt with the analysis of production systems. A survey paper (Silva and Valette, 1990) cites a number of papers in French from 1978 and 1979. Industrial process control is

cited as one of the applications of Petri nets in (Johnsonbaugh and Murata, 1982) where much of the effort was in developing hardware implementations of Petri nets.

The mid 1980's saw the publication of more French papers dealing with Petri net controllers for flexible manufacturing systems (Silva and Velilla, 1982) and (Valette *et al*, 1985).

Many of the approaches described attempted to incorporate a modular approach in order to reduce the size and complexity of models for large systems.

The application of Petri nets to manufacturing systems is a very rich domain. States and events are represented explicitly. Petri nets represent an important aid for integrating the whole system. From scheduling to real time control Petri net theory offers solutions for design, performance evaluation and implementation.

More work needed to produce efficient and distributed code for control purposes (Silva, 1983). The modelling of large concurrent manufacturing systems requires some form of modularisation to break down the complexity (Reisig, 1986)

The amount of applications to manufacturing systems can be attributed to the fact that they can analyse behavioural properties, can be used for

performance evaluation, simulation and be used to develop controllers (Zurawksi and Zhou (1994).

The late 1980's saw the introduction of Controlled Petri nets in (Krogh, 1987) and (Holloway and Krogh, 1990), which were applied to the supervisory control of discrete event dynamic systems.

In (Zurawski and Zhou, 1994) a tutorial is presented with an introduction to industrial applications of Petri nets and an up to date bibliography. In the late 1990's there is a large concentration on more high level Petri net models which incorporate other techniques such as fuzzy logic (Hanna *et al*, 1994) or object oriented methods.

3.8.3 Petri Nets for Control

Controlled Petri nets developed by Krogh (1987) allow state transitions to be influenced by external control inputs. They help to reduce computational complexity in a system (Holloway and Krogh, 1990) but are based more on the mathematical rather than the visual aspects of net theory.

Chaillet *et al* (1993) merged Petri nets with a database to control and monitor Flexible manufacturing systems however it provides no OO capabilities and is constricted by having a single control net and which controls individual modules. The work was extended by Villarroel and Muro-Medrano (1994) with their Knowledge Representation Oriented

Nets which expand the idea proposed by Chaillet to include a coordinator between the main and module level controllers.

Automation Petri nets (Uzam *et al* 2000) extended the basic Petri net structure to accommodate sensor signals that help to avoid forbidden state problems by utilising inhibitor arcs.

Manufacturing control can be either centralised or decentralised. According to (Silva and Valette, 1990), centralised control requires a co-ordinator (or manager) and a set of tasks. The co-ordinator plays the 'token game' on the net model. The tasks are attached to fired transitions.

Performance analysis tasks such as measuring throughput or scheduling exercises are performed using timed Petri nets (Murata, 1989), or stochastic Petri nets, (Marsan, 1989).

Complex simulation of flexible manufacturing systems is performed by higher level nets such as coloured Petri nets (Jenson, 1997) and attempts at Object-Oriented Petri nets (Adamou *et al*, 1998).

3.8.4 Object-Oriented Petri nets

Petri Net theory has been a major research topic for some time and several attempts have been made to integrate Petri nets and object-oriented (Delatour and Paludetto, 1998), (Venkatesh and Zhou, 1998).

Other researchers have extended the basic Petri net formalism to incorporate object-oriented concepts such as the Hierarchical Object Oriented Design (HOOD). Hierarchical Object Oriented Design (HOOD) was designed as a software development process for the European Space Agency. Petri nets were added later to provide a formal verification method. (Giovanni, 1991). HOOD generally utilises the principles of Object Orientation and adds Petri nets for functional modelling but does not enjoyed the standardisation of the UML. HOOD is also designed with a focus on ADA applications. However, it does have a form of communication mechanism between modules using the concept of a buffer (Giovanni, 1991).

However, these approaches have led to extremely complex models where the link between Petri nets and object-oriented systems design is at best tentative. In addition the techniques do not fully capture all the benefits of a true object-oriented approach.

A language for Object-Oriented Petri nets (LOOPN) is an attempt to modularise Petri nets (Lakos, 1991). LOOPN are an extension of coloured Petri nets which support modularisation, flexible token visibility, simulated notion of time and some OO features (Lakos, 1991). The main problem with LOOPN is that it is concerned with mapping of code rather than actually visualising a system.

Net Oriented Analysis and Design (NOAD) = OOA/OOD + Net theory. (Honiden and Uchihira, 1992). Attempted to integrate object-orientation with Petri nets by using the OOA/OOD approach. However, the technique is based heavily on data flow diagrams and does not provide a complete object-oriented model.

Cooperative Objects use an object control structure which defines the inner control structure of each object and is modelled with a Petri net. (Bastide, 1993). As with LOOPN this technique is based around programme code rather than the visual depiction of systems.

Object-oriented Petri nets (OPN) adds some modularity to Petri net graphs but it is not a true OO technique as it does not support classes or inheritance (Wang, 1996). This hierarchical approach allows individual places to represent entire sub-nets.

Hsiung *et al* (1997) developed MOBnet: Multiple Token Object Oriented Bi directional Net. This uses multiple tokens to represent data and introduces OO places and bidirectional arcs. Whilst it is a good technique which introduces classes and inheritance it suffers greatly from the complexity of the models.

CoOperative Objects (COO) developed by (Sibertin-Blanc, 1997) is based on C++ and is not a visual modelling but a language mapping process.

Object Petri Net Language uses high level Petri nets where tokens represent data and transitions contain time intervals and functions. It has no OO standards and is mainly designed for embedded systems (Esser, 1997).

The HOOD approach was expanded by Chen and Lu (1997) to incorporate Petri nets in their Petri-net and entity-relationship diagram based object-oriented design method (PEBOOD). PEBOOD integrates IDEF0, entity relationship diagrams and Petri nets into a suite of models. Whilst it provides some OO capabilities it lacks the standardisation of UML (Chen and Lu, 1997)

Azzopardi and Holding (1997) attempted to use OMT for modelling the static system and Petri net for the dynamics. However the resultant models are not integrated with each other and the technique leads to a number of unconnected modules

G-CPN (Serey *et al*, 1997) uses modules for grouping Petri nets but has real OO capabilities. The G-CPN method proposes no method of communication between modules.

Elementary Object Nets Uses tokens as objects (Valk, 1998) but this is only a useful technique where the tokens represent resources that move around the system.,

Object-oriented Predicate/Transition nets (OOPr/T nets) (Philippi, 1998) uses Petri nets to diagrammatically represent programming code.

State based object PN (SBOPN) model only states in an object not functionality. These use source and sinks for communication (Newman *et al*, 1998)

(Baldassari and Bruno, 1988) first proposed the idea of reducing the complexity of Petri net models by integrating them within an object-oriented framework where each object is an autonomous net exchanging messages *via* tokens.

Extended Object Oriented Petri-nets (EOPNs) were developed by Liu *et al* (2004) for coping with the complexity of wafer fabrication systems. The models use a hierarchical structure to model systems using work areas for resources, machines and functions. However, the technique requires the separation of these entities and results in a model that is not a direct representation of the system and which can be confusing for stakeholders.

The disadvantages of high level Petri nets as identified by Villani (2004) are that they are not useful for representing data and there is no notion of a hierarchy. High level Petri nets (Villani, 2004) attempted to overcome these problems but are not fully supportive of object oriented techniques.

Stanton *et al* (1996), discussed Petri Nets in relation to the specification and design of control code. The work shows how control code can be specified and designed for a manufacturing system using hierarchical Petri Nets. Structured Petri nets (Stanton, 1999), allows a direct linkage with system inputs and outputs to be modelled.

The main problem to be overcome with Petri net graphs representing even quite simple systems is 'state space explosion'. This describes the complexity caused by the number of graphical elements required to represent even a relatively simple system.

3.9 Chapter Summary

This chapter has demonstrated how a Petri net can describe a manufacturing system graphically allowing system users and designers to gain a better understanding of the complex interactions within the system.

The basic structure of a Petri net graph allows system modellers to identify and visually describe the events present in a system (via transitions) and its behaviour (*via* places).

The use of tokens in a marked n*et* allow the representation of the sequence of transition firing and subsequent changes in behaviour as the system moves through the sequence of events required to achieve its goal.

Using a token player it is possible to simulate a system hypothesis and the Petri net graph's simplicity means that it is intuitive to modify the net to carry out 'what if' analysis on the proposed system.

The analysis of Petri net graphs (via reachability marking) provides manufacturing system's analysts with a method of mathematically proving designs.

The inhibitor arc allows for the implementation of safety features within the design along with the mathematical proof this is especially important for safety or missing critical systems. The models allow for the

specification of systems which display properties of synchronicity and concurrency and these properties are highly relevant for manufacturing systems.

The use of source and sinks enables a modular approach to system design to be adopted and this is further enhanced by the ability to iteratively refine Petri net graphs at different levels of abstraction. The myriad of proposed extensions to the original Petri net formalism allow for a range of scenarios to be modelled and this has been demonstrated by the diversity of systems which have been successfully modelled.

There are drawbacks to the technique, and these have been highlighted in this chapter. The main problem to overcome is the resultant *state space explosion* resulting from the sheer number of places, transitions and arcs required to model even a relatively simple system.

Whilst some attempts have been made to modularise the nets, full object-orientation has yet to be achieved and this is addressed in this thesis by integrating Petri net graphs into UML diagrams, as discussed in chapter 4.

References

Adamou, M., Zerhouni, S. N. and Bourjault, A., 1998, Hierarchical modelling and control of flexible assembly systems using object-oriented Petri nets. *International Journal of Computer Integrated Manufacturing*, **11**, pp. 18-33.

Agerwala, T. 1979. Putting Petri Nets to Work. *IEEE Computer Society: Computer.* **12**(12), pp. 85-94.

Azzopardi, D. and Holding, D. J. 1997. Petri nets and MOT for Modelling and Analysis of DEDS. *Control Engineering Practice.* **5**(10), pp. 1407-1415.

Baldassari, M. and Bruno, G. 1988. An Environment for Object-Oriented Conceptual Programming Based on PROT Nets. *Lecture Notes in Computer Science.* **340**, pp. 1-19.

Buldizzi Fabio, Gilla Allessandro and Seatzu Carla. 2001. Modelling and Simulation of Manufacturing System Using First Order Petri Net. *International Journal of Production Research.* **39**(2), pp. 255-282.

Cecil, J. A., Srihari, K. and Emerson, C. R. 1992. A Review of Petri net Applications in Manufacturing. *The International Journal of Advanced Manufacturing Technology.* **7**(3) pp. 168-177.

Chen, K. and Lu, S. 1997. A Petri-net and entity-relationship diagram based object-oriented design method for manufacturing systems control. *International Journal of Computer Integrated Manufacturing*. **10**(1-4), pp. 17-28.

Chaillet, A., Courvoisier, M., Combacau, M. and deBonneval, A. 1993. Merging Petri Nets and Database Models for Control and Monitoring Requirements in FMS. *The International Journal of Systems, Man and Cybernetics*. **1**, pp. 42- 47.

Delatour, J. and Paludetto, M. 1998. UML/PNO: A way to Merge UML and Petri Net Objects for the Analysis of Real Time Systems. 1998. European Conference on Object-Oriented Programming. *Lecture Notes in Computer Science*. **1543**, pp 511-514.

Esser, R. 1997. An Object-Oriented Petri Net Language for Embedded Systems Design. *Proceedings of the 8th International Workshop on Software Technology and Engineering Practice*. pp. 216-223. 0-8186-7840-2

Giovanni, Raffaele. 1991. Hood Nets. *Lecture Notes in Computer Science*. **524**, pp. 140-160.

Hack, M. 1972. Masters thesis: Analysis of production schemata by Petri nets. Massachusetts Institute of Technology.

Hanna, M., 1994, "Determination of product quality from an FMS cell using Fuzzy Petri nets." In *Proc. IEEE International Conference on Systems, Man and Cybernetics,* San Antonio, TX, USA, pp. 2002-2007.

Holloway, L. E. and Krogh, B. H., 1990, "Synthesis of feedback control logic for a class of controlled Petri nets". *IEEE Transactions on Automatic Control,* **35**, pp. 514-523.

Honiden, S., and Uchihira, N. 1992. *Net* Oriented Analysis and Design. *IEICE Trans. Fundamentals.* **E75-A**(10), pp. 1317- 1324.

Hsiung, P., Lee, T., Chen, S. 1997. *MOBnet: An Extended Petri Net Model for the Concurrent Object-Oriented System-Level Synthesis of Multiprocessor.* IECE Trans. Info. & Syst. **E80-D(**2). pp. 232-242.

Jensen, K., 1997, *Coloured Petri-Nets: Basic Concepts, Analysis Methods and Practical Use, Vol. 1*, London: Springer-Verlag. 0387582762

Johnsonbaugh, R. and Murata, T. 1982. Petri nets and marked graphs - Mathematical models of concurrent computation. *In the Math Association of America, The American Math Monthly,* **89**(8), pp. 552-566.

Krogh, B. H., 1987, "Controlled Petri nets and maximally permissive feedback logic." In *Proc. 25th Annual Allerton Conference,* University of Illinois, USA, pp. 317-326.

Kichang Lee, Hanil Jeong, Chankwon Park and Jinwoo Park. 2004. Construction and Performance Analysis of A Petri Net Model Based Functional Model in Computer Integrated Manufacturing (CIM) System. *International Journal of Advanced Manufacturing Technology*. **23**(1-2), pp. 139-147.

Llewellyn, E.W., Stanton, M.J., Roberts, G.N. 2000. *Towards the implementation of the Unified Modelling Language (UML) into a Computer Integrated Manufacturing (CIM) environment.* Fourteenth International Conference on Systems Engineering. 12th – 14th September 2000. Coventry, UK. pp 398 – 403.

Llewellyn, E.W., Stanton, M.J., Roberts, G.N. 2001. Discrete event systems design based upon the UML and Petri net objects. *3rd Workshop on European Scientific and Industrial Collaboration.* 27th – 29th June 2001. Twente, The Netherlands, pp. 211-219

Llewellyn, E.W., Stanton, M.J., Roberts, G.N. 2003. A combined object-oriented and structured Petri net approach for discrete event systems' design. 4th Workshop on European Scientific and Industrial Collaboration. 28th – 30th May 2003. Miskolc, Hungary, pp. 398-403.

Liu, Huiran, Fung, Richard Y. K. and Jiang, Zhibin Dr. 2005. Modelling of semiconductor wafer fabrication systems by extended object-oriented Petri nets. *International Journal of Production Research.* **43**(3), pp. 471 – 495.

Marsan, M. A. 1990. Stochastic Petri nets: An elementary introduction. *Lecture Notes in Computer Science.* **424**, pp. 1-29.

Murata, T. 1989. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE.* **77**(4), pp. 541-581.

Newman, A., Shatz, S. M. and Xie, X. 1998. An Approach to Object System Modelling by State-Based Object Petri Nets. *Journal of Circuits, Systems and Computers.* **8**(1), pp. 1-20.

Peterson, J. L. 1977. Petri nets. *Computing Surveys.* **9**, pp. 223-252.

Peterson, James. L. 1981. *Petri net theory and the modelling of systems.* London: Prentice-Hall. 0136619835.

Petri, C. A., 1966, Communication with automata, *English translation of 'Kommunikation mit Automaten',* Griffiss Air Force Base Technical Report RADC-TR-65-377 **1**(1).

Philippi, Stephen. Modularization of Petri nets using Object-Oriented Concepts. *IEEE Journal of Systems, Man and Cybernetics.* **1**(1), pp. 84-89.

Ribeiro, A., Costa, E. and Lima Eduardo. 2008. Flexible Manufacturing Systems Modelling Using High Level Petri Nets. *ABCM Symposium Series in Mechatronics.* **3**, pp. 405-413.

Saldhana, John Anil, Shatz, Sol M. and Hu, Zhaoxia. 2001, Formalisation of Object Behavior and Interactions from UML Models, *International Journal of Software Engineering and Knowledge Engineering.* **11**(6), pp. 643-673.

Serey, D., Fernandes, J. P., Perkuisich, A. and Figueiredo, J. 1997. G-Nets: A Petri net based approach for logical and timing analysis of complex software systems. *Journal of Systems and Software.* **39**(1), pp. 39-59.

Sibertin-Blanc, C. 1997. Concurrency in CoOperative Objects. *Proceedings of the IEEE, **77**, pp. 35-44.*

Silva, M. and Velilla, S., 1982, Programmable logic controllers and Petri nets: A comparative study. In *Proc. IFAC Conference on Software for Computer Control,* Madrid, Spain, pp. 83-88.

Silva, M. and Valette, R. 1990. Petri nets and flexible manufacturing." *Lecture Notes in Computer Science.* **424**, pp. 374-417.

Srihari, K., Emerson, C. R. and Cecil, J.A. 1990, Modelling Manufacturing with Petri Nets. *The Journal of Computer Integrated Manufacturing.* **6**(3), pp.15-21.

Stanton, M.J., Arnold, W.F. and Buck, A.A. 1996. Modelling and Control of Manufacturing Systems Using Petri Net. *In proceedings of the 13th IFAC World Congress, San Francisco, USA*, vol J, pp. 324-329.

Stanton, M. J. 1999 .*Doctoral Thesis: Structured Petri Nets for the Design and Implementation of Manufacturing Control Software with Fault Monitoring Capabilities.* University of Wales College, Newport.

Uzam, M., Jones, A. H., Yucel, I. 2000. Using a Petri net Based Approach for the Real Time Supervisory Control of an Experimental Manufacturing System. The International Journal of Advanced Manufacturing Technology. **16**. pp. 498-515.

Valette, R., Courvoisier, M., Demmou, H., Bigou, J. M. and Desclaux, C., 1985, "Putting Petri nets to work for controlling flexible manufacturing systems." *In Proc. International Symposium on Circuits and Systems,* Kyoto, Japan, pp. 929-932.

Valk, R. 1998. Petri Nets as Token Objects. *Lecture Notes in Computer Science.* **1420**, pp. 1-25.

Venkatesh, K. and Zhou, M. 1998. Object Oriented Design of FMS Control Software Based on Object Modeling Technique Diagrams and Petri Nets. *International Journal of Manufacturing Systems.* **17**(2), pp. 118-136.

Villarroel, J. L., and Muro-Medrano, P. R. 1994. Using Petri net Models at the Coordination Level for Manufacturing Systems Control. *Robotics & Computer Integrated Manufacturing*. **11**(4), pp. 41-50.

Wang, L. 1996, Object Oriented Petri nets for Modelling and Analysis of Automated Manufacturing Systems. *Computer Integrated Manufacturing Systems*. **26**(2), pp. 111-125.

Wu, T. and O'grady, P.1999. Concurrent engineering approach to design for assembly. *Concurrent Engineering Research and Applications*. **7**(3), 231–243.

Wu, Z. M. 2005. Modelling and simulation of an intelligent flexible manufacturing system via high-level object Petri nets (HLOPN). *International Journal of Production Research*. **43**(7), pp. 443 -1463.

Zimmermann, Armin, Freiheit, Jorn and Huck, Alexander. 2001. A Petri Net Based Design Engine for Manufacturing. *International Journal of Production Research*. **39**(2), pp. 225-253.

Chapter 4
A Three Level Control Structure with Behavioural Constraints

This chapter discusses the Functionally Encapsulated Modules (FEMs), developed in this work, in further detail and introduces a novel methodology for their use within manufacturing systems. The chapter outlines a technique for combining the Unified Modelling Language (UML) and Structured Petri Nets (Stanton *et al*, 1999) for the modelling of manufacturing systems. The method presented identifies three levels of control in each system and this chapter describes how these control levels are decomposed down to a functional model that can intuitively be implemented. A top down design methodology is presented which maximises the loose coupling, and therefore the reuse capabilities of the system as each level is clearly modularised. The novel concept of behavioural objects is discussed as a mechanism for further ensuring the maximum reuse capability of each object in the system is achieved. A case study is presented based on a manufacturing system developed at the University of Wales, Newport and the chapter demonstrates how the full methodology created during this research work is applied to a working system. Finally an original technique for the automated generation of control code is presented.

4.1 Introduction

Manufacturing systems are complex and varied in nature and therefore their software needs cannot readily be met by general purpose 'off the shelf' packages. The approach generally adopted by software engineering practitioners is to design generic solutions, which can be customised to the specific requirements of the system. The resultant generic object class libraries are customisable through object-oriented (OO) techniques, and provide a good starting point for the design of practical control software. The abstraction of complex manufacturing systems into a series of objects is more intuitive because manufacturing end users already consider their systems in terms of objects, i.e. parts, conveyors, lathes, drilling machines etc. (Adiga, 1993). The Unified Modelling Language (UML) has become the *de facto* standard for object-oriented analysis and design and its application to manufacturing systems has already been demonstrated by the author (Llewellyn *et al*, 2000, 2001, 2003). Object-oriented modelling as a method of designing manufacturing systems has already been proposed in (Adiga and Gadre, 1990), (Adiga, 1993). The idea has further been expanded to take account of the increasing use of robots (Lin *et al*, 1994). Much of the early work in this area was based around the object-oriented methods described by (Coad and Yourdon, 1991), (Yourdon, 1994). Booch, Jacobson and Rumbaugh amalgamated the early ideas (Booch *et al*, 1999) into the Unified Modelling Language (UML) which has

become the *de facto* standard for object-oriented modelling and which has been used as the object-oriented modelling technique for this work.

In the discipline of manufacturing, Petri nets are widely used to model discrete event systems (DES) and Discrete Event Dynamic Systems (DEDS). Petri nets provide a model which is mathematically provable and, using a token player, one that also functions as a simulation tool. A number of works have also considered their ability to map against various types of control code.

The challenge facing manufacturing organisations wishing to remain competitive in a global economy is to reduce the time from product conception to market whilst retaining high quality, low cost goods. This situation is complicated by the variety and complexity of manufacturing systems and its software, which cannot readily be met by general purpose 'off the shelf' packages.

The rapid growth in the development of powerful, low cost computers has seen many attempts to integrate computers into manufacturing organisations under the umbrella term of Computer Integrated Manufacturing (CIM). However, a fundamental flaw of the *ad hoc* integration of computer technology into manufacturing organisations is the resultant 'islands of automation' (Hannam, 1997). These islands arise as

the result of implementing computers into parts of an organisation with no thought as to how these individual parts may be linked together at some future point in time. Despite this problem, computerisation offers many benefits to manufacturing organisations. A computer can do things quicker than a human and is less prone to mistakes, especially when working in hazardous environments or long, unsociable hours. The computer also provides tools to integrate the whole process from concept to market. For instance, it is possible to use a computer aided design tool to produce a first draft of the idea, the computer would then be used to aid in requirements gathering, to simulate the production processes and ultimately as a controller for the finished system. One solution to the islands of automation problem, posed by Pressman (2004), is 'loose coupling'. A loosely coupled system fully encapsulates the minute detail of an object behind a well-defined, publicly accessible interface. For example, a manufacturing cell embodying a loosely coupled design approach that utilises a pneumatic manipulator could intuitively facilitate the replacement of the manipulator with a robot arm. Such a change would cause minimal disruption to the remainder of the software controlling the system, despite the distinct differences in how the two components actually operate. This work has developed a novel approach to combining the UML and Petri nets to capture the benefits of both techniques and to overcome their shortcomings in a manufacturing environment.

Manufacturing systems are made up of a wide range of inter-related hardware and software with the associated communications infrastructure to link them together. Whilst the hardware itself is complex, it is the software which provides the intelligence to enable a machine to perform its operations. This work utilises the concept of a hardware/software object (HSO) to visualise and model machinery within the system and its associated control software.

4.2 Functionally Encapsulated Modules – Merging the UML and Petri nets.

The UML uses sequence and state charts for modelling the message flow and states of the system respectively. This results in two separate models, neither of which is mathematically provable. It is proposed that Petri net graphs can be used to capture both the message passing and states of a system in one graph. Merging the UML and the structured Petri net modules, developed in Stanton (Stanton, 1999), produces graphical models which take full advantage of current object-oriented software engineering techniques and which aid in the reduction of the state space explosion problem inherent in Petri nets. The models are also mathematically provable (Delatour and Paludetto, 1998) and allow the modelling of concurrent and non-deterministic systems (Zapf and Heinzl, 2000). However, one of the main drawbacks of Petri net graphs is their inherent complexity, even on relatively simple systems. In the UML, operations are used to access and alter the internal state of the object. The proposed technique uses Petri nets to model these operations and their resultant behaviour changes. By modelling only the limited range of states and operations within a single object the complexity of the graphs is reduced considerably. As well as capturing the static, dynamic and behavioural attributes of the system, the resultant models help in the identification of user requirements, are understandable to a wider range of

users, are extendable and reusable, and provide enough low level detail for the automatic generation of control code (Stanton, 1999). Structured Petri nets enable control and feedback places to be added to a basic Petri net structure and the encapsulation of these places into the object's interface enables a full Petri net diagram to be created if required. Breaking Petri nets down into smaller manageable sections during development enables system designers to develop modular systems akin to those developed by software engineers.

4.3 Applying Constraints

Once the classes have been designed and their operations and attributes established and modelled, the resultant object is highly generic and can be applied to a range of applications. However, in order to utilise the object, strict control must be placed over the actions it is allowed to perform. For example, a manipulator may be able to move left and right, up and down, back and forth and the gripper may open and close. When applied to a specific system the manipulator may not be able to move right due to an impeding obstacle and therefore the controller must always raise the object, move it right and lower it, in order for it to achieve the required action of moving right. If this constraint is built into the object then it becomes system specific and loses some of its genericity. This chapter proposes a method of applying a constraint object to the class in order to meet the system requirements whilst not affecting the genericity of the class itself.

The UML uses the Object Constraint Language (Warmer and Kleppe, 1999) in order to apply constraints to the model. However, these are little more than comments with no direct code conversion possible. The forbidden state problem is an is an area widely researched in Petri net theory and the work of Holloway and Krogh (1990) in applying constraints to controlled marked graphs has been adapted to fit the Petri net/UML approached presented in this thesis.

4.4 A Three Level Control Architecture

This work suggests that a three level architecture be utilised when considering the design of manufacturing systems as outlined below:

4.4.1 Goal Control

In a typical manufacturing system it is possible to identify three levels of control. The first, commonly known as supervisory control is 'goal oriented'. This primary level of control is concerned with aiding the system in the achievement of its main goal(s). Typically this will involve a single controller co-ordinating several sub-modules. It is possible that the sub-modules each have their own goal-oriented controllers. At this level the controller is concerned with achieving the specific goal of the system or sub-system. Three questions need to be answered in order to define a goal controller:

- What does the system do (goal)?
- What does it need to do it (inputs)?
- What does it do with its output (if any) (outputs)?

In order to model the levels of control in the system it is important to identify the overall goal. This can be described as the main purpose or function of the complete system under consideration. The system's goal does not take into consideration anything which is not directly within the scope of the system being considered. Anything which provides to the

target system is considered as an input, whilst anything which takes from the system is considered as an output. At this stage the functional behaviour of the system is not considered. Simply stated the first stage considers *what* the system does rather than *how* it does it. The goal controller of a system can be thought of as a sequencer, its role is to initiate the modules or objects under its control to fulfil their tasks in the required sequence in order to achieve the system goal. The goal controller in a system will generally be highly system specific and subject to major changes if at some later point in time the goal of the system changes. Therefore it is important to reduce the complexity within this level of control to a minimum whilst ensuring it maintains loose coupling with modules or objects it controls. In effect it can be thought that the goal controller takes responsibility for the aim of the system.

Definition: A goal controller is control software which co-ordinates task controllers in order to achieve the goal of the system.

4.4.2 Task Control

Once the goal control of the system is identified the individual tasks required to complete the goal are identified. The task control level is designed to operate the sub-systems under the guidance of the goal controller. The task controllers will activate upon receipt of a signal from

the goal controller, carry out their task and then [usually] report to the goal controller that the task is achieved. When developing task controllers it is imperative that their invocation commands are embedded within a public interface and that they communicate only with the goal controller *via* this interface. Direct communication with other sub-systems would violate the concept of loose coupling and in the event of the internal software or hardware of a sub-system changing, would possibly require knock-on changes to other sub-systems.

An example of a task or sub-system level controller is a programmable logical controller (PLC) that is responsible for coordinating a series of pneumatic actuators that together form a manipulator. Or, the task level controller could be required to co-ordinate a number of manipulators to achieve some specified task.

At this level of control there is no concern for any goal, the object is simply to allow the object(s), i.e. in this case the manipulator, to perform some task(s). The task controllers are initiated and co-ordinated by the goal controller. Generally there will be a task controller for each module with a specific tasks or objective which takes care of an element in the sequence of steps needed for the system to reach its goal. Once again the modules are examined individually in order to establish their inputs and outputs within the rest of the system, however they differ from goal controllers in

that they are within the scope of the system. Generally a task controller will need to co-ordinate with the goal controller above it, and the controllers beneath it. The interface between the top level should be made via uncomplicated public interfaces and will generally be limited to receiving initiate commands and sending feedback signals.

Definition: A task controller is the software which controls a sub-system. It receives a single command from the goal controller and then carries out a complete task which aids in achieving the goal of the system.

4.4.3 Object Control

The final level of control in a system is the object controllers themselves. Carrying on with the example given in the task level control section, an object level controller may be responsible for the functions of a single actuator. This level of control is highly specific to the object and communications, *via* a public interface, only with the task level controller.

Definition: An object controller is the software which enables a single object to perform its function

4.4.4 Behavioural Constraints

In order to maximise the usefulness of an object in an object-oriented system, it is important to design objects that are as generic as possible. This applies not only to the objects themselves but also to the links between them. Ideally in a system the objects should be as distinct as possible from one another, and this idea should follow through to the linkages between sub-modules. This loose coupling between objects and modules means that changes to elements of a system have little, if any, impact upon the rest of the system. However, in practice this is difficult to implement as any solution will be designed for a specific system and will have to accommodate the peculiarities and constraints of that system. System specific design will take place at the expense of the genericity of the objects and the overall design. The addition of a constraint object, as discussed later in this chapter (section 4.11), allows the designer to keep the system specifics separate from the generic objects.

4.5 A Methodology for Implementation: Analysis and Design

The initial stage in the methodology is the analysis and design stage. At this stage it is necessary to model the existing or proposed system at all levels of detail. The first stages aid in identifying the boundaries of the system under consideration including its interactions with other systems. The system is then broken down into a number of modules which are each analysed and modelled individually. Each module is then broken down into its component objects before the system is finally redeveloped.

4.5.1 Step 1: Identify System Boundaries and Interactions

The first step is to identify the boundary of the whole system and describe any interactions it has with other systems or external entities. At this stage the inputs and outputs of the system are identified and its overall goal is determined. Use cases (models and scenarios) are used to describe the top-level of the system and should be intuitive for all stakeholders to understand.

4.5.2 Step 2: Identify Sub-Systems, Boundaries and Interactions

Having established the system and its goal the sub-systems that make up the whole are identified. This is achieved by identifying the distinct tasks that must be achieved in order to meet the goal. This stage also establishes the boundary of each sub system and its inputs from and outputs to the rest of the system, where appropriate. The inputs and

outputs of sub-systems will ultimately become the control and feedback signals passed between the system's goal and task controllers. Use cases models and scenarios are once again utilised to capture the workings at this level. Class diagrams are also utilised to ensure that the make-up of conceptual groupings of components is identified.

4.5.3 Step 3: Identify Modules and their Interactions

Each sub-system is analysed individually to establish the modules that form that part of the system. Each module is modelled with use cases to establish its task and how it co-operates with other modules within the sub-system. Class diagrams are used to capture the composition of each module.

4.5.4 Step 4: Identify Objects and their Functionality

The final step in the analysis and design stage is to identify and model the objects that make up each module. This is achieved using Functionally Encapsulated Modules (FEMs). At this stage all functionality of the object is captured and modelled. The modelling allows the system designer to capture all possible operations that the object can perform, irrespective of the current system. Class diagrams are also utilised to capture the inheritance that may exist between system objects.

4.6 A Methodology for Implementation: Development

The second stage in the methodology is concerned with the development of the control software at each level, i.e. – object, task, behavioural and goal.

4.6.1 Step 5: Develop Object Controllers

The Object Controllers are developed to ensure that each object is capable of performing all possible tasks which it is able to undertake. This ensures that the object is fully reusable and can form part of a class library of pre-built and tested components for use in other systems.

4.6.2 Step 6: Develop Task Controllers

The task controller ensures that a sub system is able to work together to form a specific tasks within the system. In many cases the grouping of such objects to perform a task provides a logical module which can also be reused in other systems. For example the controller for a group of actuator objects which forms a pneumatic manipulator can be reused as a complete unit.

4.6.3 Step 7: Develop Behavioural Constraints

In order for modules to be reused in similar systems, behavioural constraints are developed to ensure that the module meets the needs of the particular system under consideration without having to remove any of

the generic nature of the control code. The behavioural constraints act as an intermediary between task and object level controllers.

4.6.4 Step 8: Develop the Goal Controller

The goal controller acts as a sequencer which co-ordinates all the sub-systems under its control in order to meet the goal of the system. This level of control is inevitably system specific and should be designed to be as intuitive and non complex as possible. The goal controller is established specifically for the current system and will generally not be useable in other scenarios.

4.7 A Methodology for Implementation: Testing

At this stage simulations are run on the system to ensure it fully complies with user requirements and possibly to evaluate alternative design implementations. The models should be reconfigured to ensure the system works to optimum capacity. Simulation can be undertaken at object, task and goal level control. Each object can be tested to ensure it is capable of achieving its full potential. Task controllers can be tested to ensure they fully meet their given task. Goal controllers can be tested to ensure the whole system achieves its goal.

4.8 A Methodology for Implementation: Implementation

The final stage is to implement the system by automatically generating code from the FEMs designed at earlier stages. This work presents a

method of generating pseudo-code which can be translated into any language(s) required by the hardware controllers.

4.9 Functionally Encapsulated Modules

Functionally Encapsulated Modules (FEM) are a method of obtaining full object-orientation by using the powerful static modelling capabilities of the UML with the dynamic capabilities of a Petri net graph.

To demonstrate how FEM work a pneumatic manipulator will be used as an example. The actuator is a composition of four actuators and four sensors. The sensors are an inherent part of the actuator and enable the current state of that part to be ascertained. Figure 4-1 shows the graphic representation of class Manipulator using the UML's notation.

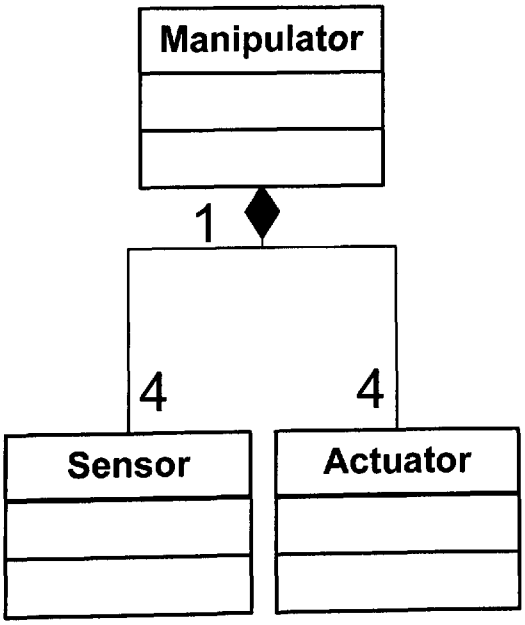


Figure 4-1: A pneumatic manipulator

An actuator can be in one of two states, that is it can be actuated or de-actuated. Its inherent sensor is aware of these two states (see Figure 4-2).

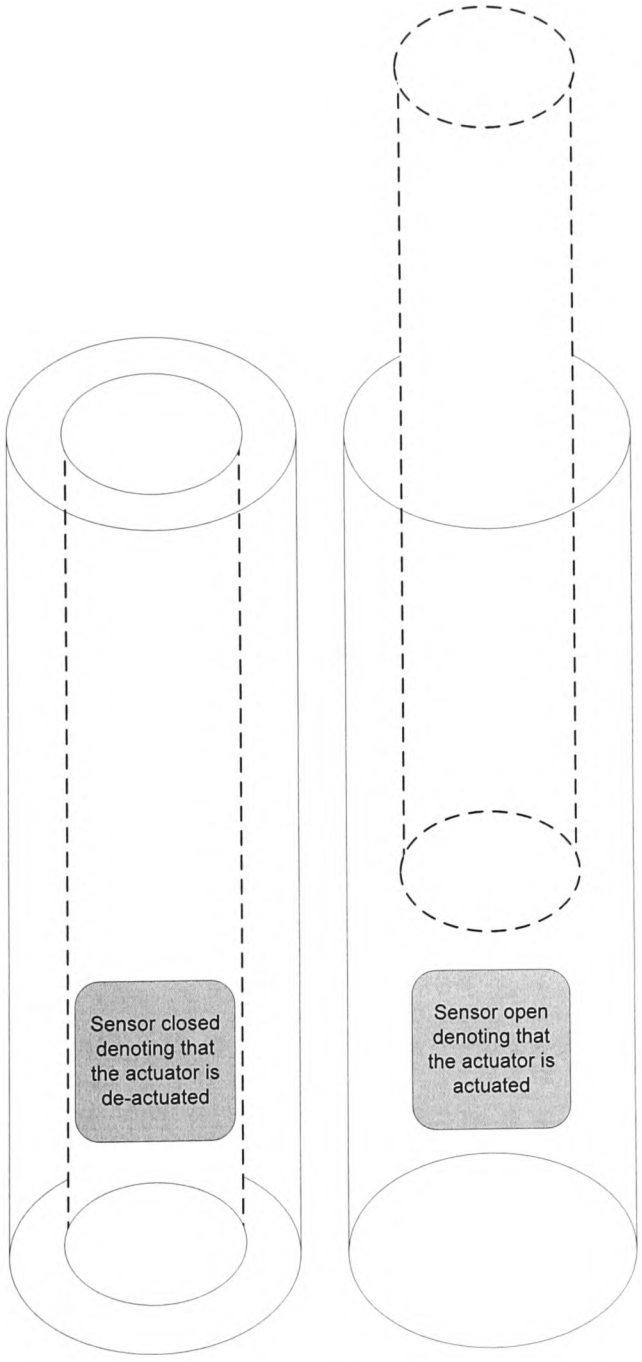


Figure 4-2: The two states of a pneumatic actuator

The actuators that make up the pneumatic manipulator each have their own characteristics when assembled into the unit. One is responsible for moving left or right; one moves up and down; one moves forward and backwards and finally one is used as a gripper which can be opened and closed (see Figure 4-3).

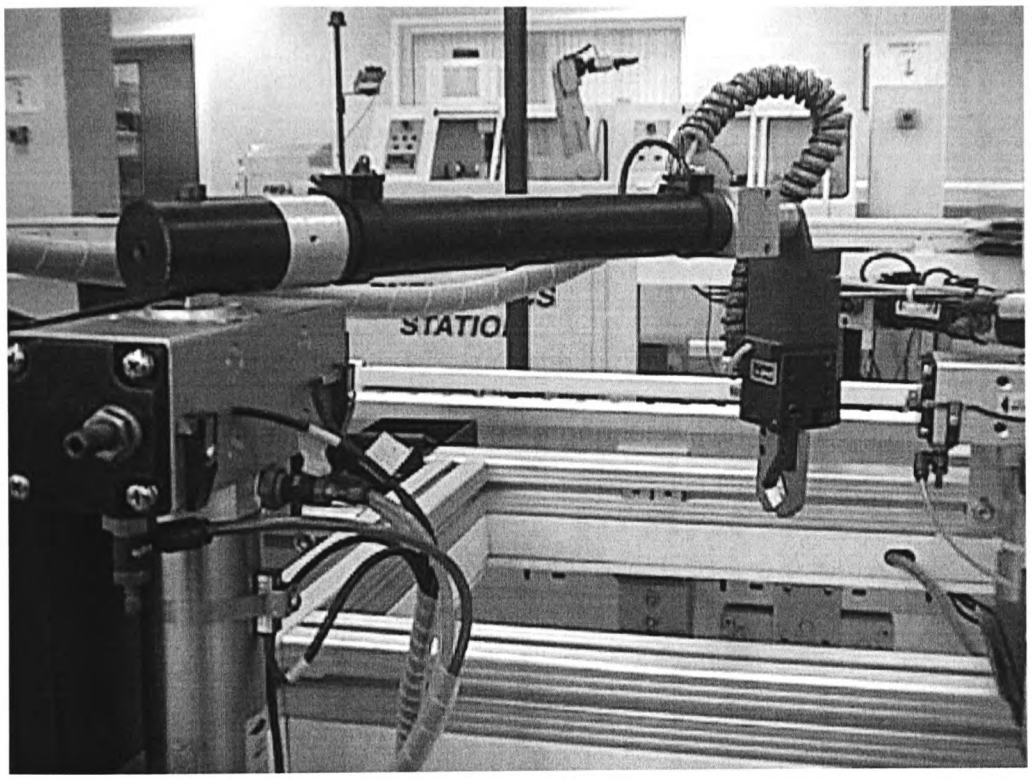


Figure 4-3: A Pneumatic Manipulator

The Manipulator class can undertake eight operations – move forward, backwards, up, down, left, right, open and close [gripper]. Each actuator can only understand two commands – actuate and de-actuate.

A bottom-up approach is taken to model the capabilities of the actuator functions using a Petri net graph as shown in Figure 4-4.

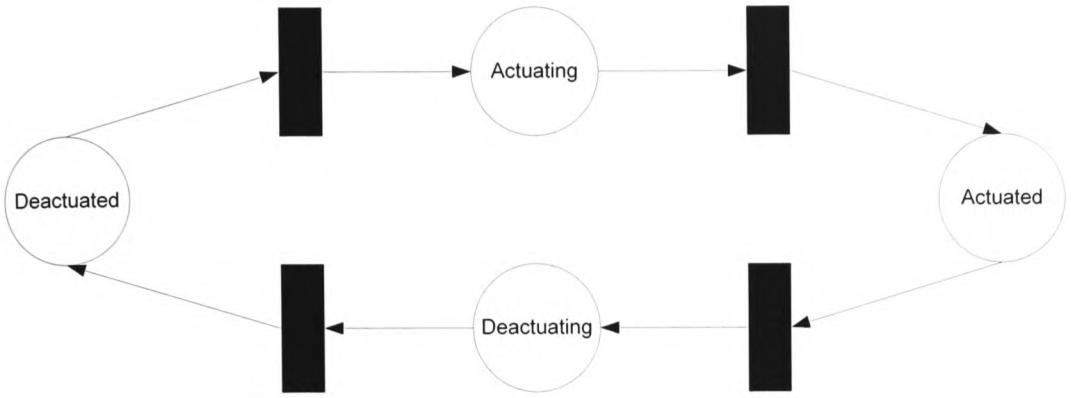


Figure 4-4: The functionality of an actuator

However in addition to the physical states of the actuator it is necessary to model its interaction with the sensors to model those states and the control signals to initialise them.

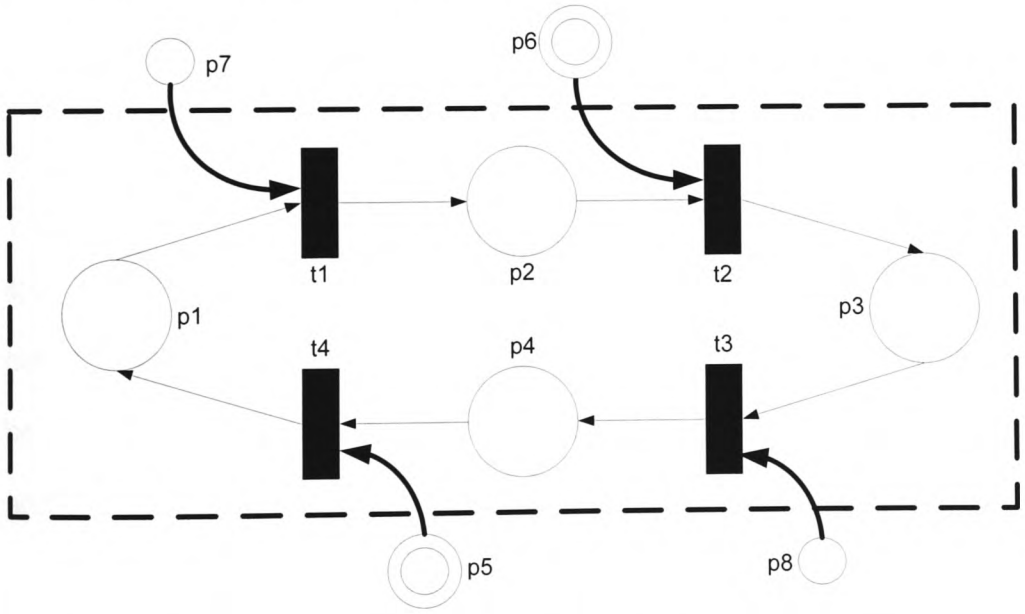


Figure 4-5: A pneumatic actuator showing control and feedback places

Figure 4-5 shows the control (p_7, p_8) and feedback places (p_5, p_6). The dotted line represents the interface to the object. Only p_5 to p_8 are

accessible from outside of the object to ensure the properties of encapsulation. The places correspond to the actions show in Table 4-1

Place	Function
p_1	Actuator is deactuated
p_2	Actuator is actuating
p_3	Actuator is actuated
p_4	Actuator is deactuating
p_5	Sensor is closed showing the actuator is deactuated
p_6	Sensor is open showing that the actuator is deactuated
p_7	Command to begin actuating
p_8	Command to being deactuating

Table 4-1: The mapping of places to functions

The Manipulator is made up of four actuator objects as shown in Figure 4-6, each with their own attributes and operations. No consideration needs to be given as to how these operations are performed as they will be accessed *via* their public interface only. The state attribute is private and can only be accessed through the getState operation.

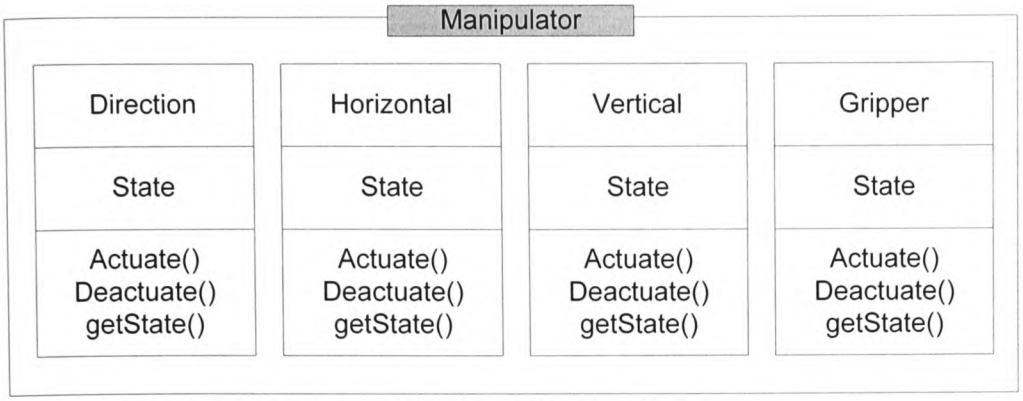


Figure 4-6: The composition of the manipulator

4.10 Controlling a Functionally Encapsulated Module

The FEM is controlled *via* its Petri net representation from a control object. The control object itself is also represented with a structured Petri net. As an example the manipulator may wish to pick up a component and place it on a loading area. The sequence of actions is shown in Table 4-2.

Action	Function Call	Result
move up	Vertical.actuate	Moves the arm up
move forward	Horizontal.actuate	Moves the arm forward
open the gripper	Gripper.actuate	Opens the gripper
move down	Vertical.deactuate	Moves the arm down
close the gripper	Gripper.deactuate	Close the gripper (picking up the component)
move up	Vertical.actuate	Moves the arm up
move right	Direction.actuate	Moves the arm right
move down	Vertical.deactuate	Moves the arm down
open the gripper	Gripper.actuate	Opens the gripper (releasing the component)
move up	Vertical.actuate	Moves the arm up
move left	Direction.deactuate	Moves the arm left
move back	Horizontal.deactuate	Moves the arm back
close the gripper	Gripper.deactuate	Close the gripper

Table 4-2: The sequence of operations for picking up an object

For safety purposes it is vital that the system is in a safe state. It is important, therefore to establish a safe state for the object being controlled.

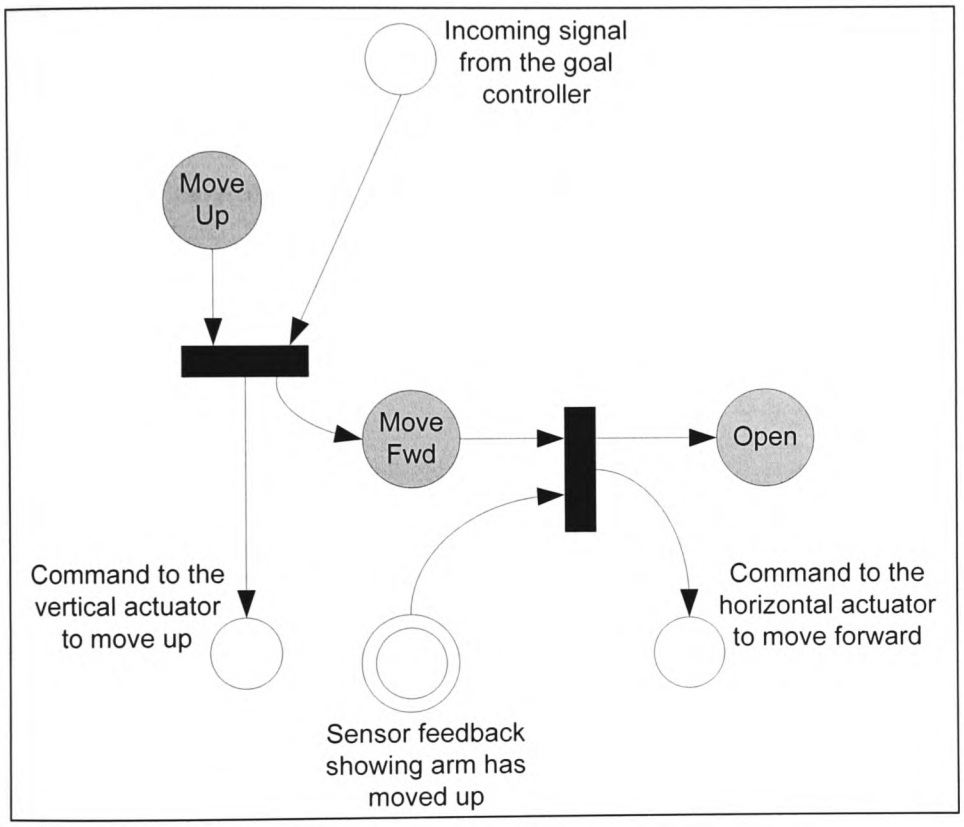


Figure 4-7: Part of the control net for the manipulator

Figure 4-7 shows part of the control net for the manipulator example. An incoming signal from the goal controller would request that the manipulator carry out its task. This is the invocation for the sub-system to carry out a complete cycle thereby performing its task. The sequence of events is represented by the grey circles and shows the first three steps being

carried out. Control places emanate from each stage to the relevant object. Feedback from the object is used to ensure that the one stage is completed before moving onto the next. It would be possible, based on user requirements to allow several actions to occur simultaneously, possibly then waiting for a number of feedback signals to occur before the system proceeds. For example the arm could move up and forward and open the gripper simultaneously before waiting for feedback to ensure it is in the correct state before lowering to pick up the object.

4.11 Behavioural Constraints

The actuator class has been designed to be as generic as possible, as indeed is the resultant manipulator. It can be seen that this object can be reused in any application. To ensure the object remains as general purpose as possible the environment specific constraints are built into a separate object which acts as an intermediary between the controller, which is goal specific and the manipulator object itself. In the system under consideration, the only constraint for the raw materials manipulator is that the gripper cannot be opened when the arm is raised. Imagining the cylinders to be quite heavy, doing so could amount in considerable damage to the other objects in the system and possibly the cylinder itself.

Figure 4-8 shows a constrained object being used. The controller object sends a message to the manipulator *via* its constraint. The constraint validates the request based on the current state of the object it is constraining, and depending upon the outcome either sends the message on to the object for implementation or returns an error message to the controller.

The constraint here is the intermediary between the controller and manipulator object, in other cases the constraint could be constraining a combination of objects where it is ensuring there are no conflicts between objects operating in the same environment.

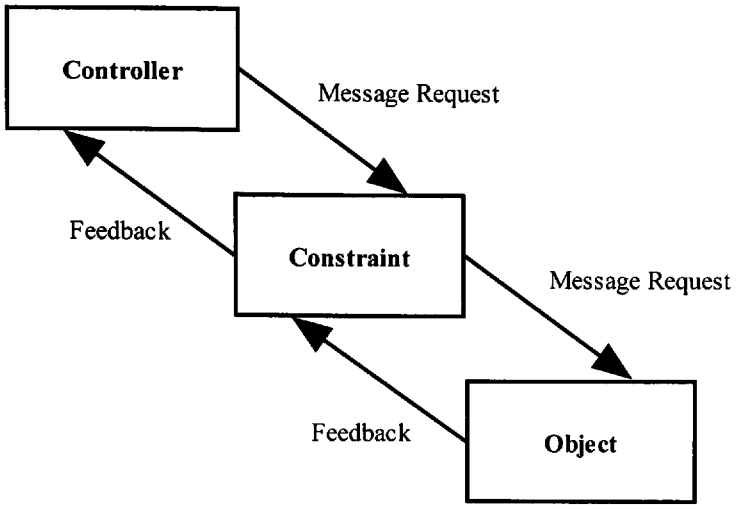


Figure 4-8: A constrained object

The actuator class has been designed to be as generic as possible, as indeed is the resultant manipulator, which can still utilise its full six degrees of freedom, meaning that they can be reused in any application. To ensure the object remains as general purpose as possible the environment specific constraints are built into a separate object that acts as an intermediary between the controller, which is goal specific, and the manipulator object itself, which is task specific.

4.12 Chapter Summary

This chapter has presented a novel methodology for a combined object-oriented and Petri net approach to the development of manufacturing systems. The technique designed in this work, entitled Functionally Encapsulated Modules, utilises Petri net graphs to model the functions of each object. This allows system designers to capture both the state and dynamics of an object in a single visual representation. It also allows for each module to be simulated for testing purposes. Using structured Petri nets which allow for modelling of control and feedback signals considerably reduces the complexity of the resultant Petri net graphs. This goes some way to reducing the state space explosion problem inherent in large complex systems.

The technique outlined in this chapter addresses many of the methodology issues highlighted in chapter 2:

- User requirements are iteratively captured using a series of use case diagrams and scenarios. A top down, abstracted view of the system from the perspective of its goals is initially taken. This view is then refined to establish more and more detail about the system. The use case models are intuitive for all stakeholders and ensure clear communication between technical and non-technical personnel. The use cases can be cross referenced at each stage of

the design process to ensure that the system adheres to the user requirements;

- Once the system has been modularised a bottom up approach is taken to capture the capabilities of each system object. Viewing the objects as independent entities ensures their full functionality is captured. Object controllers are developed for individual objects or groups of objects which are inter-dependant. This facilitates the building of a library of generic and reusable classes which can be utilised in other systems or later in redesign processes;
- Communication between objects is only undertaken *via* public interfaces in the objects. This is facilitated by control and feedback places in the Petri net structure. At implementation stages the control and feedback places are coded as public operations. This feature ensures that systems are loosely coupled. Loose coupling in this case will ensure that changes to objects in the system have a minimal impact on other objects. Objects can be used based on what they do rather than how they do it;
- Objects and modules can be individually tested using the token player facilities of Petri net graphs. Upon system integration the entire system can be simulated using the same method;
- The well defined interfaces presented by FEMs enable system designers to incrementally upgrade parts or all of a system.

References

Adiga, S. 1993. *Object-oriented Software for Manufacturing Systems.* London, UK: Chapman & Hall. 0412397501.

Adiga, S. &Gadre, M. 1990. Object-Oriented Software Modeling of a Flexible Manufacturing System. *Journal of Intelligent and Robotic Systems*, **3**, pp. 147-165.

Bittner, K. 2003. *Use Case Modelling*. Boston, USA: Addison Wesley. 0201709139.

Booch, G., Rumbaugh, J. & Jacobson, I. 1999. *The Unified Modeling Language User Guide.* USA: Addison Wesley Longman. 0321267974.

Chen, K. & Lu, S. 1997. A Petri-net and entity-relationship diagram based object-oriented design method for manufacturing systems control. *International Journal of Computer Integrated Manufacturing.* **10**(1-4), pp. 17-28.

Coad, P. and Yourdon, E., 1990. *Object-Oriented Analysis*. 2nd edn. Michigan: Prentice Hall. 0387333320

Delatour, J. &Paludetto, M. 1998. UML/PNO: A Way to Merge UML and Petri Net Objects for the Analysis of Real-Time Systems. *Lecture Notes in Computer Science*, **15** (43), pp. 511-514.

Di Giovanni, R. 1991. Hood Nets. *Lecture Notes in Computer Science.* **524**, pp. 140-160.

Hannam, Roger. 1997. Computer Integrated Manufacturing: from concepts to realisation. Harlow: Addison-Wesley. 0201175460.

Lin, L., Wakabayashi, M. & Adiga, S. 1994. Object-oriented modelling and implementation of control software for a robotic flexible manufacturing cell. *Robotics & Computer-Integrated Manufacturing*, **11**(1), pp. 1-12.

Llewellyn, E.W., Stanton, M.J., Roberts, G.N. 2000. *Towards the implementation of the Unified Modelling Language (UML) into a Computer Integrated Manufacturing (CIM) environment.* Fourteenth International Conference on Systems Engineering. 12th – 14th September 2000. Coventry, UK, pp 398 – 403.

Llewellyn, E.W., Stanton, M.J., Roberts, G.N. 2001. Discrete event systems design based upon the UML and Petri net objects. 3rd Workshop on European Scientific and Industrial Collaboration. 27th – 29th June 2001. Twente, The Netherlands, pp. 211-219

Llewellyn, E.W., Stanton, M.J., Roberts, G.N. 2003. A combined object-oriented and structured Petri net approach for discrete event systems' design. 4th Workshop on European Scientific and Industrial Collaboration. 28th – 30th May 2003. Miskolc, Hungary, pp. 398-403.

Meyer, Bertrand. 1997. Object-oriented Software Construction. 2nd edn. London: Prentice-Hall. 0136291554.

Narisawa, F., Naya, H. &Yokoyama, T. 1998. A Code Generator with Application-Oriented Size Optimization for Object-Oriented Embedded Control Software. *Lecture Notes in Computer Science*, **15**(43), pp. 511-514.

Pressman, Roger S. 2004. Software Engineering a practitioner's approach. 6th Ed. London: McGraw-Hill. 0071238409.

Stanton, M. J. 1999 .Doctoral Thesis: Structured Petri Nets for the Design and Implementation of Manufacturing Control Software with Fault Monitoring Capabilities. University of Wales College, Newport.

Venkatesh, K. & Zhou, M. 1998. Object-oriented design of FMS control software based on object modeling technique diagrams and Petri nets. *Journal of Manufacturing Systems*, **17**(2), pp. 118-136.

Warmer, J. & Kleppe, A. 1999. *The Object Constraint Language.* Reading, USA: Addison-Wesley. 0201379406.

Wu, B. 1995. Object-oriented systems analysis and definition of manufacturing operations. *International Journal of Production Resources*, **33**(4), pp. 955-974.

Yourdon, E. 1994. *Object-Oriented Systems Design.* New Jersey, USA: Prentice-Hall. 0136363253.

Zapf, M. & Heinzl, A. 2000. Approaches to integrate Petri nets and object-oriented concepts. *Translated from WIRTSCHAFTS 'INFORMATIK* . **42**(1), pp. 36-48.

Chapter 5

Application of Functionally Encapsulated Modules to a Manufacturing System

This chapter demonstrates a unique technique, called Functionally Encapsulated Modules (FEMs). FEMs form one of the contributions of this research work by combining the Unified Modelling Language (UML) and Structured Petri nets (Stanton, 1999) for modelling manufacturing systems. The chapter highlights how the three level control architecture, described in Chapter 4, is applied to a manufacturing system where the resultant models provide full control of the system. A further contribution, entitled behavioural objects, is applied as a mechanism for ensuring the maximum reuse capability of each object in the system is achieved. The benefits of the loosely coupled and highly reusable systems produced by this original methodology are clearly demonstrated. The chapter presents a case study based on a manufacturing system developed at the University of Wales, Newport and the chapter demonstrates how the full methodology created during this research work is applied to a working system. Finally an original technique for the automated generation of pseudo-control code is presented.

5.1 Application

The technique described in this paper is demonstrated by applying it to the University of Wales, Newport's Computer Integrated Manufacturing (CIM) system. Initially user centric views of the system are modelled (Bittner, 2003) using use case scenarios and their resultant diagrams. Next the classes in the system are identified and their attributes and operations captured. The attributes (or states) and operations are modelled using Petri net graphs, where one graph is used to model all operations for a particular class. Output places (Stanton, 1999) are used to represent message passing between objects. Finally the system constraints are identified and placed in a constraint class for each object.

The system (shown in Figure 5-1 and Figure 5-2) has been designed as an example of a modern CIM system and it incorporates a number of "modules" that interact in order to produce two end products. The end products are a milled block and a lathed cylinder which can be manually combined to produce a paper-weight. The raw materials used by the system are a Perspex block and a metal cylinder. The block and cylinder originate from the Raw Materials Station (RMS) and are placed into trays on a conveyor belt for transportation around the system. The system performs two main functions – the block is milled and the cylinder lathed so that the two items can be fitted together. Finally the finished product is stored in the Automated Storage and Retrieval System (ASRS).

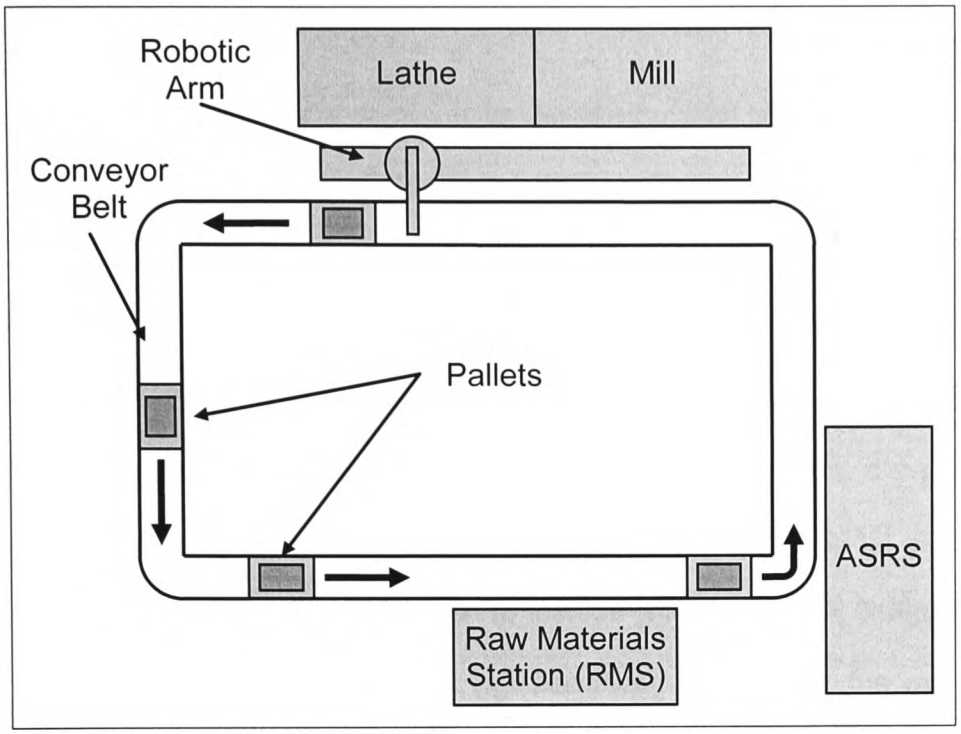


Figure 5-1: A Schematic of the CIM System (adapted from Stanton, 1999)

Figure 5-2: The University of Wales, Newport CIM System

5.2 Definition of System Goal and Boundaries

The first stage in the methodology is to identify the goal of the system and define its boundaries. In order to complete this stage the system must be analysed at its highest level of abstraction.

The system begins its process when an employee initiates a start sequence. In operation the system processes two raw materials, a cylinder and a block which are lathed and milled respectively before being placed in an ASRS. At some later stage an employee will remove the completed components for manual assembly. A use-case diagram for the overall goal of the system is shown in Figure 5-3:

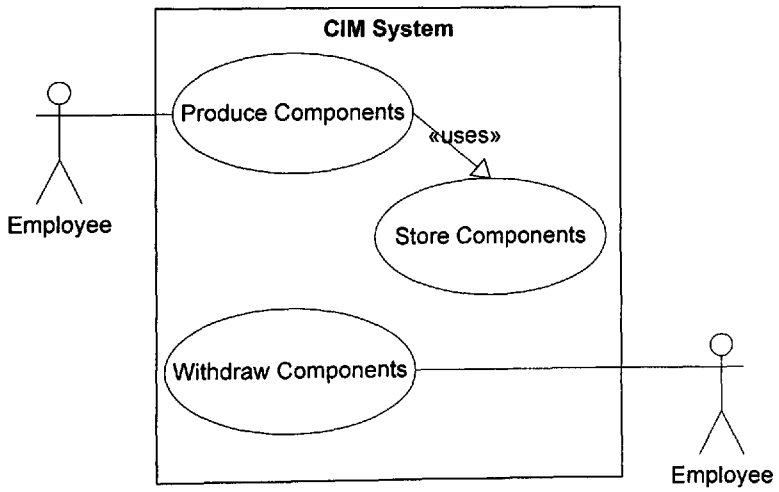


Figure 5-3: A use-case diagram depicting the CIM system

The associated scenario would be:

1. An Employee starts the system by pressing the start button;
2. The components are produced;
3. The components are stored;
4. An Employee withdraws the components from the ASRS.

The only external interactions are the employee who initialises the system and who, at some later stage, retrieves the completed components. The use-case diagram establishes that the system has two main functions, i.e. the production of components and their subsequent storage. Whilst withdrawing and assembling the components is vital to the organisation it is a manual task performed by an employee and needs no further thought in this design process.

5.3 Identify Sub-Systems

The sub-systems can most easily be identified by analysing the processing that is undertaken in order to achieve the goal of the system. It has already been established that the system needs to produce two components, carry out some tasks upon them and then store them for later retrieval. Further analysis of the system indicates that the system is comprised of:

- A raw materials station (RMS), which achieves the task of providing raw materials;
- A Mill and Lathe, which produce the components in combination with an associated robot arm (the combined unit is defined as the Machining Unit (MU));
- A conveyer belt for transporting raw materials and components around the system;
- The Automated Storage and Retrieval System (ASRS) for storage of the machined components.

Identification of tasks is useful for establishing the modular composition of the system. The tasks identified are:

- Provide raw materials (RMS);
- Produce components (MU);

- Transport items (Conveyer belt);
- Store components (ASRS).

The resultant sub-systems that have been identified are shown in Figure 5-4 below:

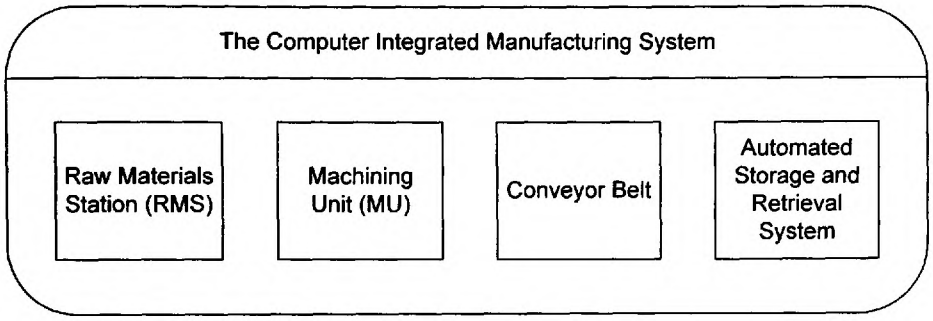


Figure 5-4: Identification of Sub-Systems

5.4 Task Controllers

To demonstrate the process of analysing a task controller within this methodology, the RMS will be examined in more detail. Figure 5-5 shows a schematic for the RMS which consists of two manipulators and two storage units. The storage units contain blocks and cylinders respectively. One of the manipulators is used to load cylinders onto a pallet waiting in the loading area, whilst the other serves the dual purpose of placing pallets onto the loading area, and populating pallets with blocks. The whole station is controlled by a programmable logic controller (PLC) which interacts with a series of valves and pneumatic actuators.

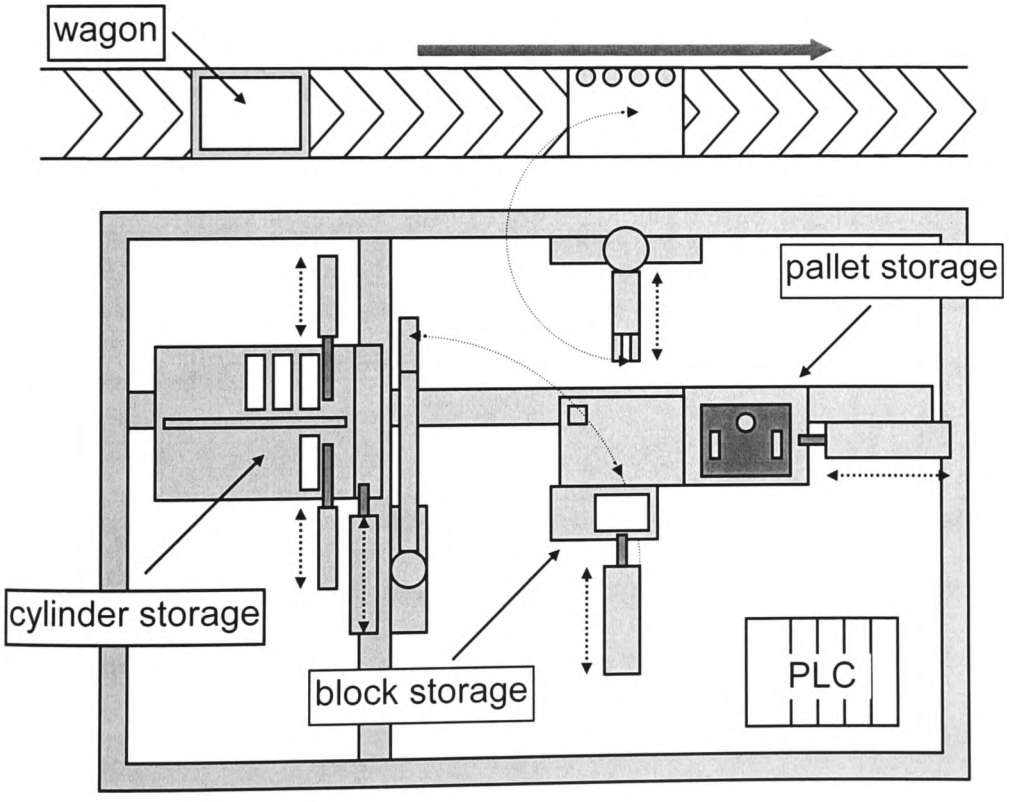


Figure 5-5: Schematic of the RMS (adapted from Stanton, 1999)

The RMS undertakes two main tasks and use case scenarios can describe these as:

Scenario 1: Provide a Block

1. A pallet is retrieved from the pallet storage area by the pallet manipulator and placed onto the loading area;
2. A block is retrieved from the block storage area by the pallet manipulator and placed onto the pallet;
3. The loaded pallet is placed onto the conveyor belt by the pallet manipulator.

Scenario 2: Provide a Cylinder

1. A pallet is retrieved from the pallet storage area by the pallet manipulator and placed onto the loading area;
2. A cylinder is retrieved from the cylinder storage area by the cylinder manipulator and placed onto the pallet;
3. The loaded pallet is placed onto the conveyor belt by the pallet manipulator.

The use case-scenario is interesting as it demonstrates that step 1 and 3 of both scenarios is identical providing an opportunity to reuse code. As the system requires both components to undertake a complete production cycle the two scenarios can be thought of as a single task which is to provide raw materials.

The resultant use case diagram is shown in Figure 5-6. The use case diagram combines both scenarios i.e. putting a Block onto the conveyer and putting a Cylinder onto conveyer. These scenarios are combined as the use-case scenario clearly indicates that they are highly similar. Use case modelling has already shown one of its benefits as it becomes apparent from the diagram that a raw material cannot be placed onto the conveyor without first requesting a pallet. This may not have initially been evident from any textual description. For control purposes two commands can be established – getBlock and getCylinder and these represents the tasks for which the RMS is responsible. A further command – getPallet is also utilised but as this does not fulfil a task (for goal purposes) it can be called from within the other functions. *Using* a function in this way helps reduce the complexity of the task level controller. Without identifying the overlap in this sub-system the task level controller would have needed four operations to be invoked in sequence – getPallet, getBlock, getPallet, getCylinder. This has a significant impact on the amount and efficiency of the control code required. It is important to note that communication from the rest of the system can only be conducted *via* the RMS controller through its public interface containing the two specified public functions – getBlock and getCylinder.

When attempting to establish how to decompose a system, the need for reuse should be borne in mind. For example, the system described

contains a RMS which is responsible for supplying the raw materials to processes later in the system. It can be considered in three ways depending upon the design approach:

1) **Provide: A block, a cylinder and a pallet** - These are the actual raw materials that are contained in the RMS and one description of the system could be: Put Pallet, Put Block and Put Cylinder. These cases would need to be called by the main system controller and would have to be distinctly sequenced by that control mechanism.

2) **Provide: A block and cylinder** - The RMS cannot provide either a block or cylinder unless a pallet is supplied first, and therefore it is possible to describe a Put Block and Put Cylinder method with each implicitly relying on, including in use case *parlance*, the Put Pallet operation. The former two methods are again called from, and sequenced by the main system controller, however a small degree of control has been taken out of this level and placed within the scope of the RMS itself.

3) **Provide: Raw materials for a complete unit** - In terms of the system's goal, the fact that a block and cylinder are needed is

incidental, whilst the pallet is irrelevant. These details, whilst crucial to the operation of the system, do not form part of the goal, which is to supply a complete unit. This goal level control is specific to the system and it is therefore usually difficult to encompass any form of reusability at this level. Therefore the goal controller i.e. that controlling the overall flow of the system needs to be as streamlined as possible. In terms of this work, streamlined would mean containing the least functionality possible, or at its simplest, the least number of functions. By adopting, the "provide raw materials for a complete set approach" the designer has reduced the number of functions in the goal controller from three to one and encapsulated more of the detail into the RMS itself, which would now be responsible for providing a pallet, then block, a pallet and then cylinder. The RMS would include all the functions for this action and would be responsible for sequencing the order of events at goal level.

The order in which the block or cylinder tasks are accomplished and whether the system should be able to provide individual raw materials is established by discussion with the end users of the system. In this case-study the order of raw materials processing is irrelevant as long as both components are provided. However, the goal controller needs to be

'aware' of what is being loaded onto the conveyor in order to decide whether to invoke the mill or the lathe in later stages of the process. If this were not the case a single, provide raw materials, command would have sufficed.

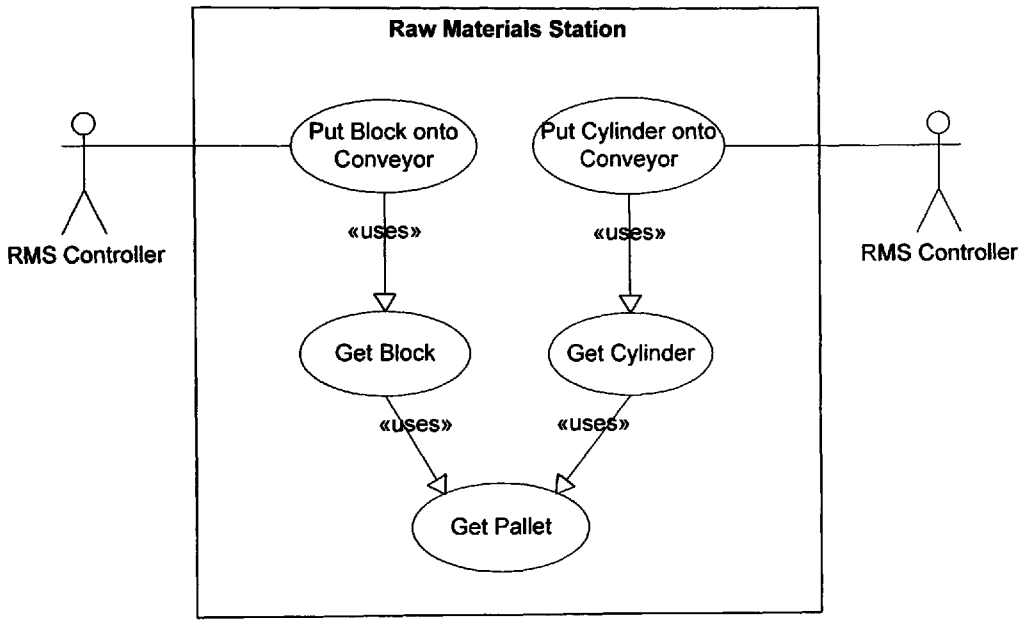


Figure 5-6: Use case scenarios for the raw materials station

The use-case modelling technique provides an intuitive method of capturing user requirements. It is important to initially model the system in its 'optimum' state, i.e. fully working as intended by the end-user as this is ultimately what the system designer is attempting to achieve. However, subsequent iterations can capture more detail about the system, including building in fault tolerance. Figure 5-7 shows the use-case diagram

extended to allow for possible exception conditions such as running out of raw materials.

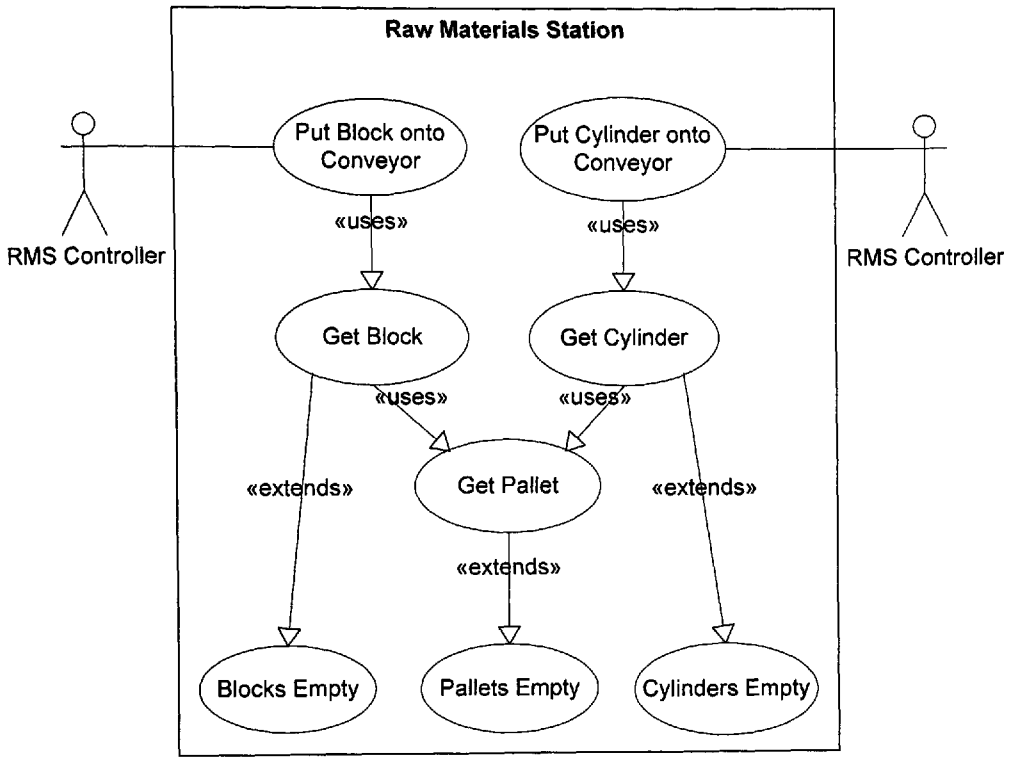


Figure 5-7: The RMS use case extended to show exception conditions

The use case diagram in Figure 5-7 identifies the communication between the RMS task controller and the components within the module. By *encapsulating* (also known as *information hiding*) this information so that external entities can only communicate with the RMS through its public interface, this technique achieves the concept of *loose coupling* (Sommerville, 2006), (Meyer. 1997), (Pressman, 2004). The internal operations of the module are hidden from the user. In order to operate this module, external entities only need to know about its interface, which describes the operations it performs. The internal details of how it

provides a cylinder or block, or gets a pallet, are unimportant when calling these operations. Therefore, modifications made to the internals of an object should have a minimal, if any, effect on other objects in the system, as long as its interface remains unchanged.

The models describe so far demonstrate the concept of a hardware/software object, where no distinction is drawn between the software and hardware in the module. Instead the module is thought of in terms of the operations it performs and the interface to those operations.

The storage units are fairly intuitive devices and simply provide raw materials on demand. Each storage unit contains an actuator which is used to release the component into an area for retrieval by the manipulators. The two manipulators within the system however are required to do a range of tasks. The Pallet Manipulator (PM) is responsible for retrieving pallets from the conveyor belt and placing them on the loading bay and retrieving blocks from the block storage unit (BSU) and placing them on pallets. It is also responsible for placing loaded pallets, containing either a block or a cylinder onto the conveyor belt. This whole series of operations would generally take place as a result of a single command from the RMS controller requesting that a block be placed onto the conveyer (Figure 5-8).

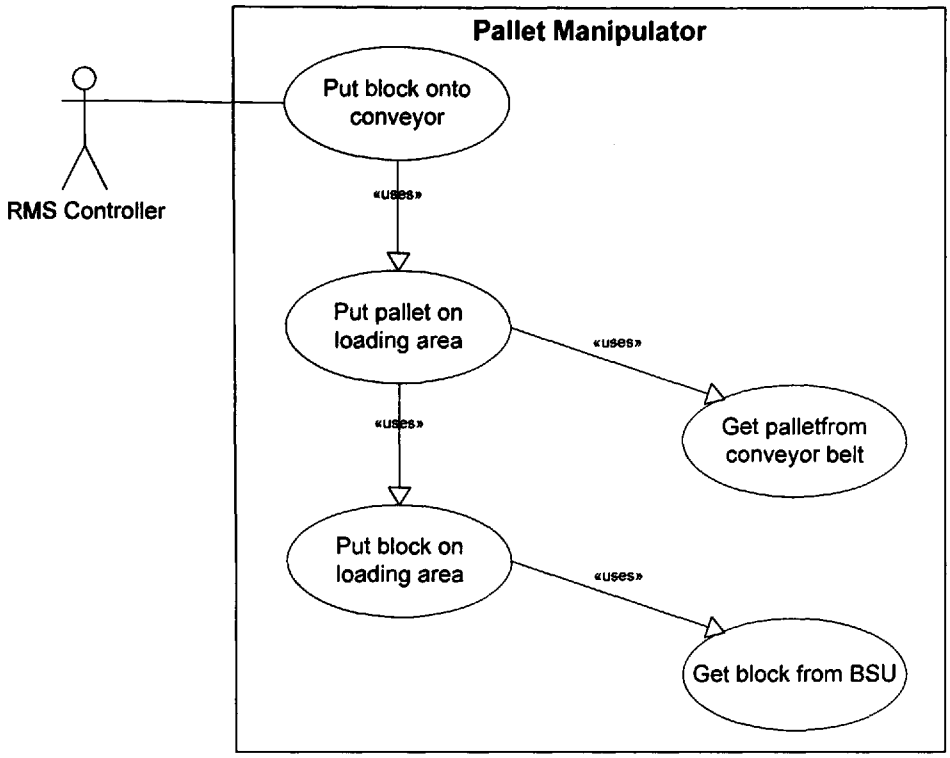


Figure 5-8: A use-case diagram for the pallet manipulator

The Cylinder Manipulator (CM) has a single function and that is to retrieve cylinders from the Cylinder Storage Unit (CSU) and place them onto a pallet waiting in the loading area. However, it is reliant on the PM first providing a pallet from the conveyor onto the loading area.

The use-case diagram for the CM (Figure 5-9) clearly shows that to complete a request by the RMS controller to provide a cylinder, it must rely upon the PM and so these two objects are tightly coupled. Such tight coupling is acceptable in this case as these two objects are being conceptually integrated to form a module.

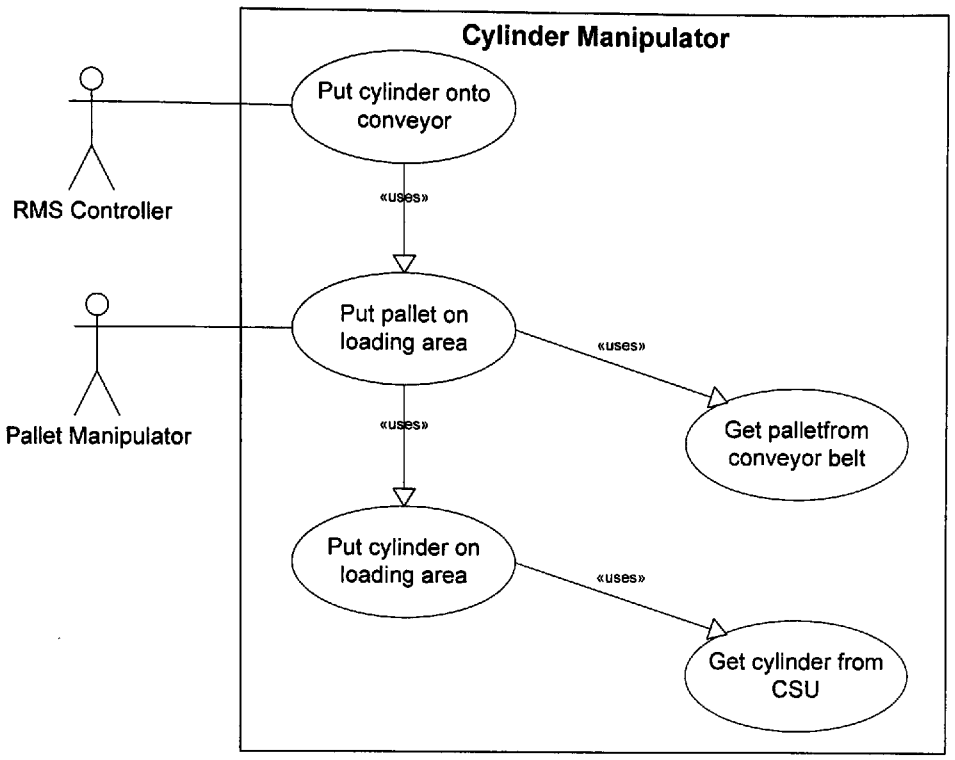


Figure 5-9: A use-case diagram for the cylinder manipulator

The system goal requires that both a block and a cylinder are provided in order to produce a complete unit and therefore the system must present both items for a successful execution cycle.

A typical use-case scenario for the execution cycle would proceed as show in Table 5-1.

Stage	Process One (Block)	Process Two (Cylinder)
1	A *pallet* is retrieved from storage and placed onto the loading area.	
2	A *block* is retrieved from storage and loaded onto a pallet before being placed on a conveyer belt. This process is carried out by a pneumatic manipulator	A *cylinder* is retrieved from storage and loaded onto a pallet before being placed on a conveyer belt. This process is carried out by a pneumatic manipulator
3	The loaded pallet is moved to a machining unit by the conveyer belt	
4	A robot arm removes the block and positions it in the cradle of the mill	A robot arm removes the cylinder and positions it in the cradle of the lathe
5	A threaded circle is cut into the block by the mill	The cylinder has a thread cut into it by the lathe
6	The robot arm places the completed component back into its pallet on the conveyer belt	
7	The loaded pallet is moved to an automated storage and retrieval unit (ASRU).	
8	A robot arm loads the pallet and its cargo into the ASRU	

Table 5-1: A use-case scenario for the RMS

5.5 Capturing the Static System for Reuse Purposes

From an abstracted viewpoint, it can be seen that the RMS is made up of a number of storage units and manipulators. Figure 5-10 shows such an aggregation (or generalisation) relationship using the UML's class diagram convention. An aggregation relationship is denoted by an unfilled triangle and each end of the relationship is qualified. As this is a generic and multi-purpose description, which is not system specific, a one to many (*) qualifier is shown. This denotes one RMS as being an aggregation of a number of, or many (*) StorageUnit classes and Manipulator classes. In the actual case-study the relationship would be one to two for both the RMS to StorageUnit and RMS to Manipulator.

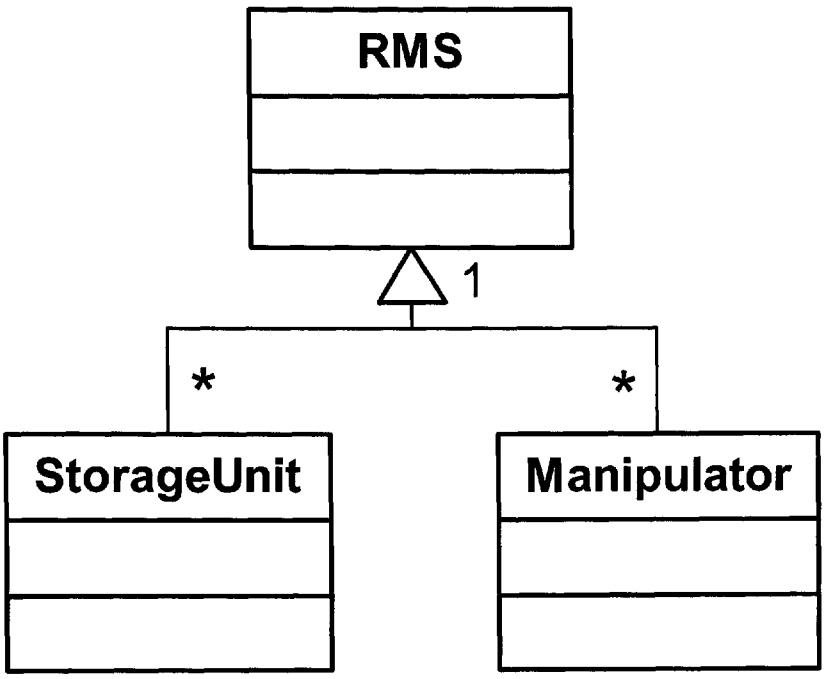


Figure 5-10: The RMS as an aggregation

Examining the RMS system in more detail it can be seen that whilst the RMS is the top level grouping for two other classes, i.e. the StorageUnit and Manipulator classes, each of these in turn is composed of its own sub-classes. The StorageUnit class is composed of StorageBay, Sensor and Actuator classes. The Manipulator Class is composed of Sensor and Actuator Classes. This relationship denoted in Figure 5-11 differs from that shown earlier in Figure 5-10, as this is a different type of inheritance relationship known as a composition. The relationship in Figure 5-11 denotes the actual case study rather than a generalised version. A composition relationship is a stronger form of relationship than an aggregation. A RMS is generally composed of a number of StorageUnit

classes and Manipulator classes, but could possibly be made up of just StorageUnits with no Manipulators. A Manipulator, however, can only exist with its child classes intact. An aggregation can be thought of as an optional or loosely coupled form of inheritance, whilst a composition is a compulsory or tightly coupled relationship.

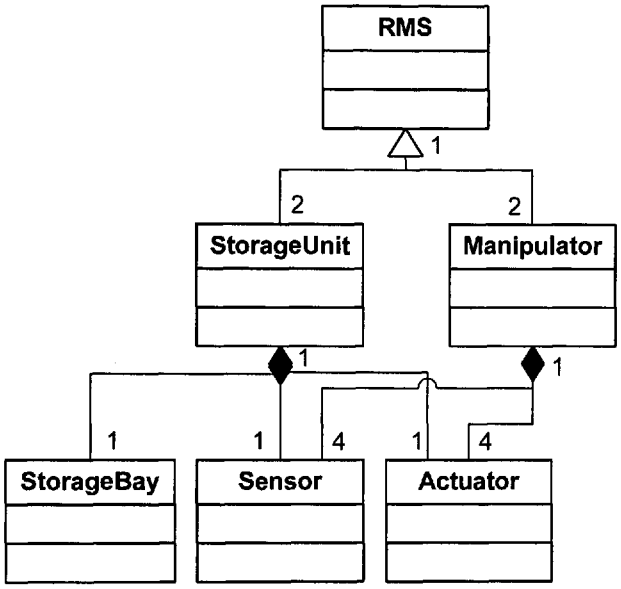


Figure 5-11: The full class make-up of the RMS

The class diagram in Figure 5-11 aids in the identification of sub-modules and this highly abstracted model is subsequently refined to flesh out the details of each individual class, as described below.

A Manipulator is actually composed of a number of actuators and sensors (4 in this case), which are classes in their own right. Figure 5-12 shows this composition with each end of the relationship being qualified, i.e. for one manipulator there is a composition of four actuators (for the Newport

system at least). It is also possible to qualify these roles with ranges, such as 0..4 meaning a possible range of 0 to 4 children to a parent class; with no fixed limitation (*) meaning an infinite number of children to a parent class including none; or 1..* meaning at least one, but no upper limit.

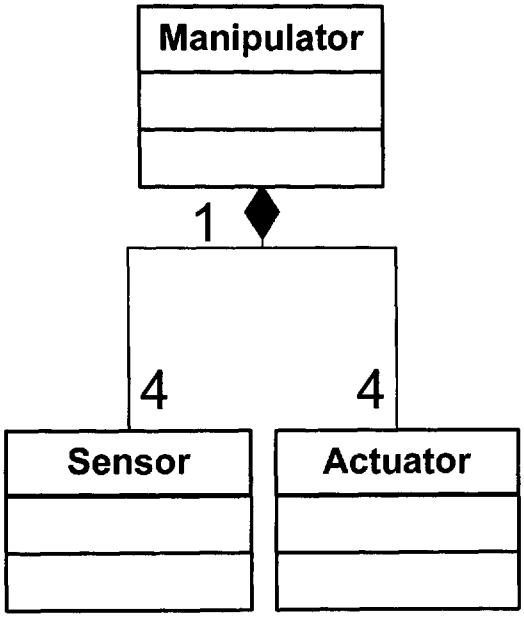


Figure 5-12: A pneumatic manipulator showing a composition relationship

From an object-oriented point of view, it can be seen that the system described consists of a series of classes. Booch *et al* (1999) define a class as "a description of a set of objects that share the same attributes, operations, relationships and semantics." An object then, is a unique instance of a class.

Two other important properties of a class are:

- Operations - which are used to read or manipulate the data of an object and;
- Attributes - the structure of the objects: their components and the information or data contained therein

The 'building block' of this section of the system is the pneumatic actuator, which can be either actuated or de-actuated. These operations take place when a Programmable Logic Controller (PLC) opens a valve, which pumps air into the actuator, thus actuating it. When the PLC closes the valve, the air is removed and the actuator de-actuates. This description provides a basic overview of the system, which is all that is required in order to model it. From this description it is possible to identify the following classes: - PLC, Valve and Actuator. These classes can now be examined in further detail.

The actuator can be in either one of two final states - actuated or de-actuated. It can also be midway between these states, i.e. it can be in the process of actuating or de-actuating. Therefore, the actuator requires two operations, one to carry out the action of actuating and the other to carry out the action of de-actuating. In addition, if the system is to provide feedback it must allow external entities, such as a controller, to interrogate

the actuator to determine its current state. This can be achieved *via* feedback from sensors. It is therefore possible to establish that an actuator has a state attribute and an operation that provides that state to external entities. The possible range of values that the actuator can take are: actuating, de-actuating or busy. An important concept for attributes is that of visibility. Visibility applies to attributes and operations and specifies the extent to which other classes can use a given class' attributes or operations. Three levels of visibility can be described. At the public level, usability extends to other classes (represented by a "+" symbol). At the protected level, usability is open only to the classes that inherit from the original class (represented by a "#" symbol). At the private level, only the original class can use the attribute or operation (represented by a "-" symbol). The actuate and de-actuate operations are called by the Valve class and therefore are public, as is the getState operation. Generally, classes are shown with the first letter of each word in uppercase. Attributes and operations usually start with a lower case letter. Figure 5-13 shows a class diagram for a class of type Actuator.

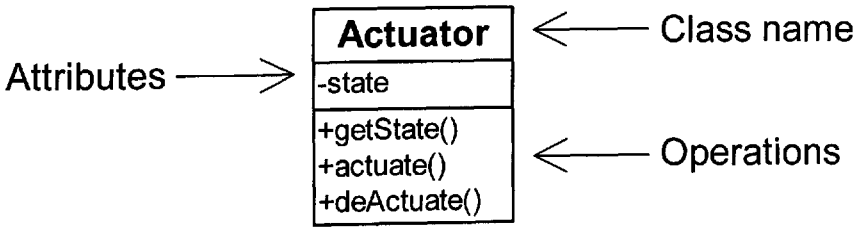


Figure 5-13: The Actuator Class

In addition, a state diagram shows all the possible states an actuator can have. This helps to identify all the possible values the state attribute can take. This is shown in Figure 5-14, where it can be ascertained that in order to arrive at the desired states of actuated or de-actuated, the actuator must pass through a 'working' phase where it is either actuating or de-actuating.

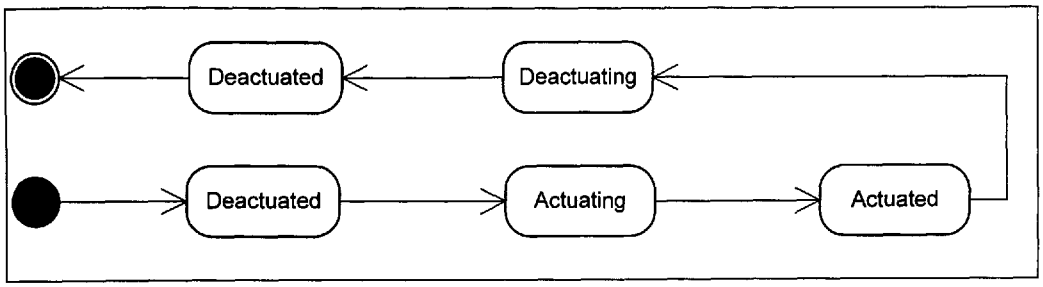


Figure 5-14: A state diagram for the actuator class

The more explicit Actuating and Deactuating states have been used rather than busy, as it is important for the system sequence controller to be aware of these states, so that it does not try to invoke the operations of a busy object.

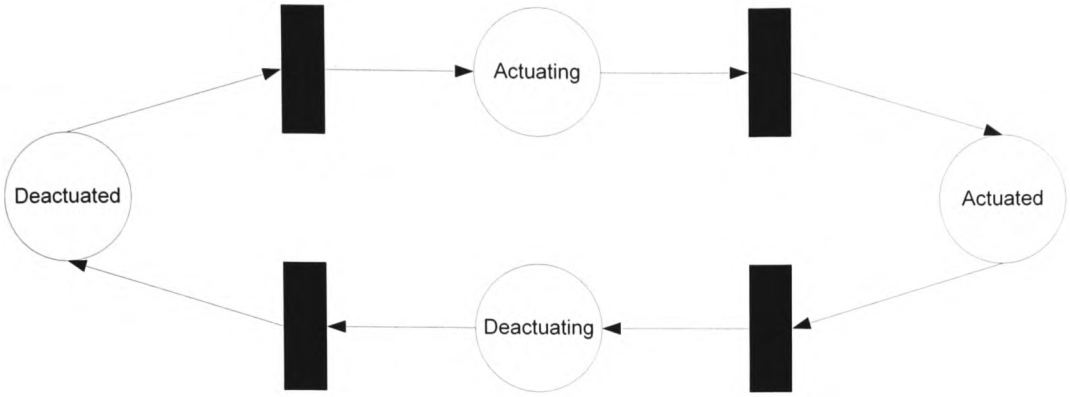


Figure 5-15: A Petri net diagram for the actuator class

Figure 5-15 gives the equivalent Petri net model for the state diagram shown in Figure 5-14. It can be seen that the resultant diagrams are similar. The only notable difference being the addition of transition places in the Petri net graph. Transition places are an important aspect of the code generation discussed in section 5.9.

The textual description shows that the valve can be either open or closed, and again there must be the intermediate steps of opening or closing. For feedback purposes it will be necessary to establish the current state of the valve. The class and state diagrams for the valve are similar to those for the actuator. From the description it becomes apparent that the PLC class controls the operation of opening and closing the valve, therefore it is logical to assume that the open and close methods are public, as is the valve's state attribute.

In this instance, the PLC class (shown in Figure 5-16) needs only to start or stop the predetermined sequence of events of the objects under its control. These operations (start and stop) are public.

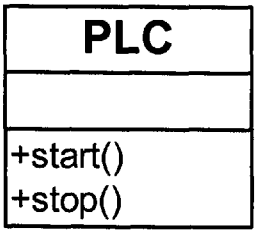


Figure 5-16: The PLC class

It would be reasonable to assume that in other circumstances the controller could be another object in the system. This is known as an external entity, which may well be an integral part of the larger system. Decomposition into subsystems allows the external entity or actor to be represented with a stick figure, indicating that while it is understood to be an important object, which needs representation, its complexity need not be modelled at this stage. It is sufficient to know that it performs the action of starting and stopping the PLC. The ability to generalise in this way enables the system designer to plan for various types of implementation. For example, the PLC could be controlled by a human, another PLC or a computer. The diagram would not need to change in any of these circumstances. The functional details of how the PLC works are encapsulated within the Class. In order to interface an instance of class PLC with a controller, all the required information can be accessed via the

public operations and attributes. These represent the interface between the Class and the outside world.

5.6 Modelling System Dynamics

The method calls between objects can be more clearly seen on a sequence diagram, which also shows the order in which the operations are invoked (Figure 5-17). The diagram gives a pictorial representation of the two possible final states of a pneumatic actuator and the procedure for arriving in those states i.e. the actuator being actuated, and the actuator being de-actuated. This model shows the functional detail of the dynamics of the actuator. Once this information has been captured it is possible to write all the functional code and ignore the complexity involved in the actions of the actuator. This 'code and forget' approach means it is no longer necessary to consider the PLC or the valve, instead the system designer can concentrate on the detail of what the actuator is intended to do, as part of the greater system. However, as the functionality of the system increases in complexity, or the system becomes larger, these diagrams become complex and unwieldy.

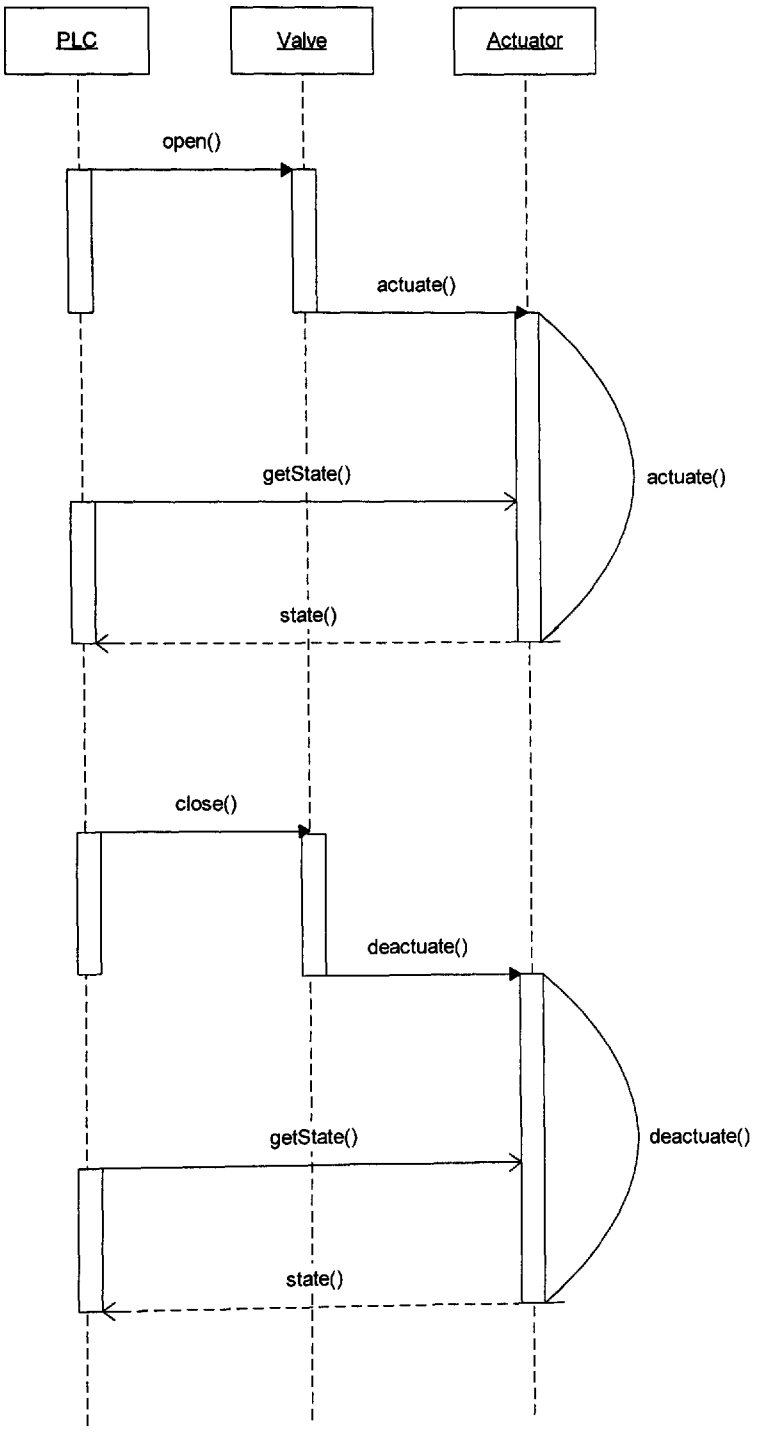


Figure 5-17: A sequence diagram for control of an actuator

In the four-actuator model described, each of the actuators can carry out the same basic function, i.e. they can actuate or de-actuate. However, the same operation call has differing effects on the action being performed by its recipient object. For example, actuating an actuator can raise it, move it left, or open it. This sharing of an operation is called polymorphism, which may be described as the situation where an operation has the same name in different classes and each class 'knows' how that operation is supposed to take place.

An instance of class Manipulator may comprise of four pneumatic actuators as shown in Figure 5-12. It can be clearly seen in this diagram that the four separate actuators whilst all having the same basic characteristics, are slightly different. This raises another important object-oriented concept, that of inheritance. As Yourdon (1994) defines it "[inheritance] allows an object to incorporate all or part of the definition of another object as part of its own definition." The Class Actuator used to make up the Manipulator above is actually decomposed into three subclasses or child classes - a LinearActuator, RotaryActuator and Gripper (Figure 5-18).

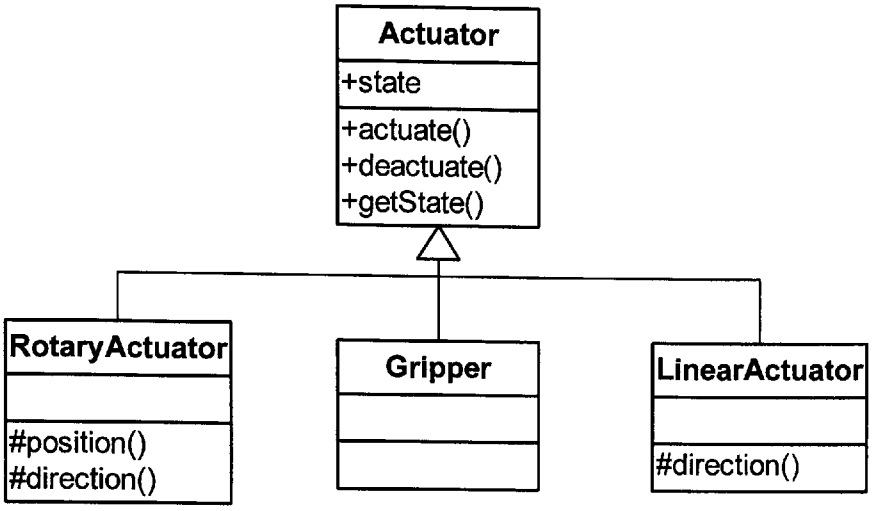


Figure 5-18: The actuator class showing inheritance

Each subclass inherits all the attributes and operations of the Actuator class and each add their own unique attributes. For example, the LinearActuator Class adds the position attribute which enables the actuator to have a horizontal or vertical position. The RotaryActuator has a direction which enables it to find out the direction of travel when the actuate operation is carried out. The Gripper will open when actuated and close when de-actuated. This demonstrates the principle of polymorphism, each of the Actuators has an actuate operation, but each reacts differently when called. These new attributes are visible only to the creating class and are therefore protected.

Focusing on the CM, it can be observed that the object is an *instance* of class manipulator, and that this class itself is a *composition* of four instances of class actuator. The actuator class has two simple methods

that allow it to actuate or deactuate. However, these actions carry out a different operation depending on the receiving object. For instance, an actuator in the system under consideration may take one of four types. It contains a rotary actuator, which is able to actuate right or deactuate left. It contains a horizontal actuator which is able to extend or retract, and a vertical actuator which is able to move up or down upon receiving its actuate or deactuate command. Finally it contains a gripper which when actuated opens and on deactuation closes. This demonstrates the object-oriented concept of *polymorphism* whereby each of the classes responds differently to the same command based upon its hidden internal mechanisms. The manipulator class itself responds to commands such as move left, move right, up, down, open and close. These commands or operations form the interface to the manipulator class, with the individual actuators, and indeed their pneumatic valves and the PLC controller being encapsulated from the user.

Figure 5-18 demonstrates how the fundamentals of a hardware/software object are designed. From a software perspective the different variants of actuator all have exactly the same functionality. The class diagram enables the system designer to represent the hardware differences of each such as direction and position. This information is valuable in terms of sourcing and utilising the correct hardware but, as can be seen, has no impact on the software as each type of actuator will respond in the correct

way due to its physical makeup. For example an actuate signal to a rotary actuator will make it rotate whereas an actuate signal to an vertical actuator will make it extend. As long as the correct hardware is used the software will enable it to function correctly within the system.

5.7 Functionally Encapsulated Modules

As demonstrated in this chapter the UML uses a series of models to describe the varying levels inherent in a system, until enough detail is established to translate the design into code. The models enable the system designer to establish modularity within the system and identifies, using inheritance, elements that are capable of being shared amongst components. This aspect of the design is crucial for enabling the system to be decomposed into modules for team development and for the reduction in code duplication. It is also fundamental to the principles of reuse which are a cornerstone of the techniques presented in this work.

In a system thus modelled with objects, operations (functions and procedures) are used to access and alter the internal state of an object and to invoke its behaviour. In the UML sequence diagrams are used to capture the interaction between objects whilst state diagrams are used to capture state changes. Neither diagram provides the functional level detail necessary to module the detailed operations of an object, or indeed, a system. In addition, the UML diagrams do not offer any form of mathematical provability which can be vital in safety critical systems.

This work uses Petri nets to model the operations of objects and their resultant behaviour changes (states). By modelling only the limited range

of states and operations within a single object the complexity of the graphs is reduced considerably. The resultant diagrams, entitled Functionally Encapsulated Modules (FEMs), provide true object-oriented capabilities and combined with the modelling technique presented in this work are:

- Mathematically provable utilising Petri net analysis techniques as outlined in (Delatour and Paludetto, 1998);
- Provide full object-oriented capabilities;
- Allow the capture of user requirements in a form that is easy to communicate between users and system designers;
- Enable a system to be rapidly deployed using pre-tested components;
- Provide a simulation tool and;
- Facilitate the automatic generation of control code.

Having captured the class diagrams and any inheritance present in the system using standard UML notation, it is possible to model the dynamic capabilities of the class with Petri net graphs. These are the operations that need to be invoked in order to make the class carry out its functions. In addition, the operations provide a method of altering the state or behaviour of the object.

In a discrete event system (DES) such as the CIM system being considered, the state of the system at any moment in time can be captured by obtaining the states of all objects in that system. A Petri net graph allows these states to be represented visually or, if required, mathematically.

The actuator class can be modelled using the Functionally Encapsulated Module (FEM) shown in Figure 5-19. In the diagram smaller circles represent control places and feedback. The former are signals from the controller that invoke the method of the object. In this instance these can be either actuate or deactuate. The feedback is being sent to the controller object, with double circles representing input from external feedback sources. The dashed line represents the external (public) interface to the object.

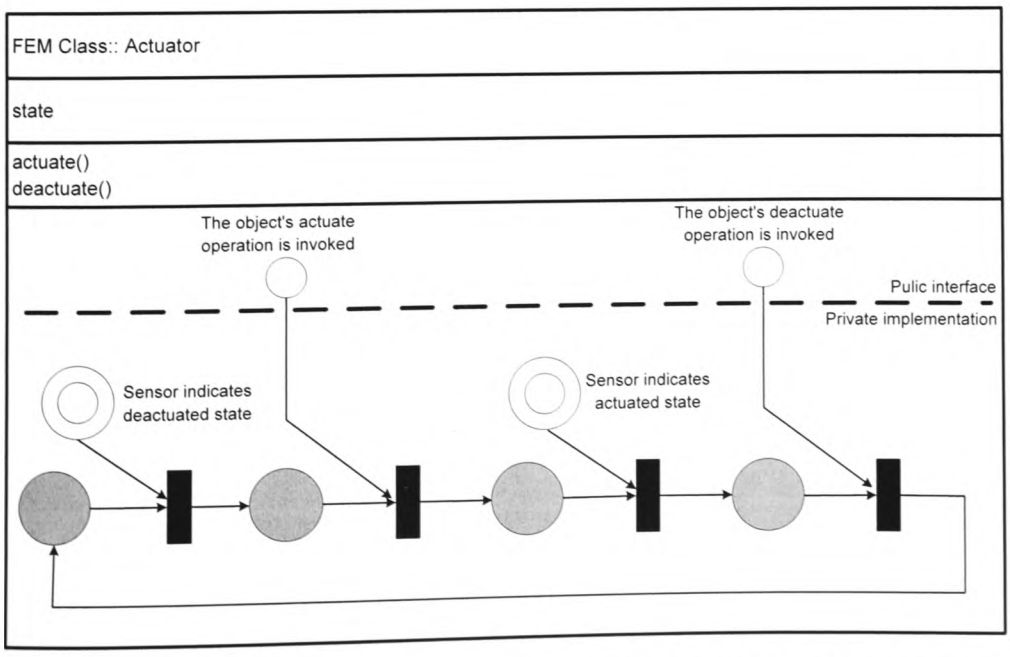


Figure 5-19: A functionally encapsulated module for the actuator class

The FEM uses standard UML class notation but adds a Petri net representation of the operations within the object. The Petri net graph in Figure 5-19 contains all the detail required to code the Actuator object.

Figure 5-19 contains labels to clarify the objective of the FEM Class:: Actuator. To generate the code from the model a formally labelled Petri net graph is shown in Figure 5-20.

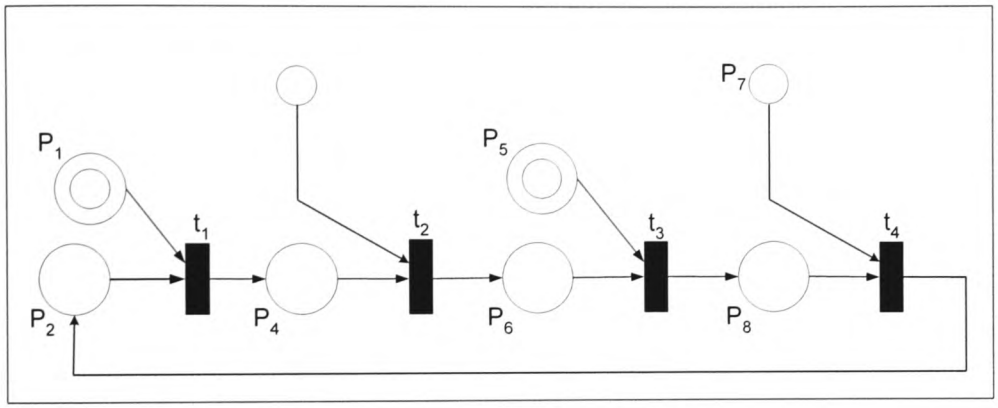


Figure 5-20: A Petri net for code generation

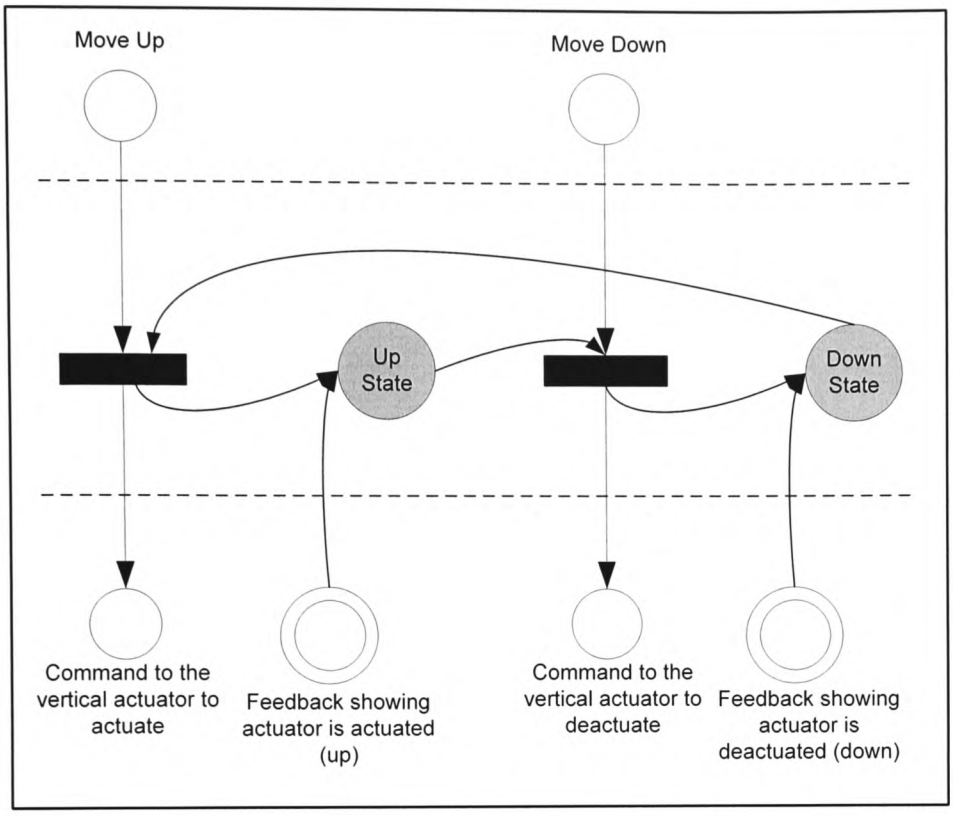
The private implementation contains the low level code for each operation as follows:

```
Public Sub objectName.Actuate (P3 = true)
        object.name.State = "actuating"
        If P3 = true and P4 = true THEN
                P6 = true
                P3 = false
```

```
            P4 = false
        End IF
        If P5 = true and P6 = true THEN
            P8 = true
            P5 = false
            P6 = false
            object.name.State = "actuated"
        End IF
    End sub
    Public Sub objectName.Deactuate (P7 = true)
        object.name.State = "deactuating"
        If P7 = true and P8 = true THEN
            P2 = true
            P7 = false
            P8 = false
        End IF
        If P2 = true and P1 = true THEN
            P4 = true
            P1 = false
            P2 = false
            object.name.State = "deactuated"
        End IF
End sub
```

The actuator (hardware and software) is now reduced to its functionality (behaviour) represented by its public operations. The functionality is encapsulated away in the private implementation and can only be accessed *via* its public interface. Any system wishing to reuse the actuator object only needs to be aware of its public interface.

The next stage in developing the functionality is to define an FEM for the Manipulator's controller. The Manipulator is a physical object which relies on a composition of four actuators. As the individual code for each actuator has already been defined all that is needed at this stage is to define the control sequence for it to be able to operate flexibly. The manipulators can move up/down, left/right, extend/retract and can open/close a gripper. Each of these four actions is the individual responsibility of an actuator so it is possible to take the code from the FEM created above and utilise its functionality within this component.



5-21: Part of the control structure for a Manipulator

Figure 5-21 shows the private implementation section of an FEM Class for the Manipulator. As the manipulator controls four separate actuators it would have the structure in Figure 5-21 replicated a further four times one each for up/down (shown), left/right, extend/retract and open/close. Each net would be identical to the one shown above other than the labels and which object it interacts with. The manipulator describe above is "aware" of the state of its child components through the information gained from feedback places.

From the case study it can be established that there are two manipulators within the RMS. One is responsible for providing a blocks and pallets, the other takes care of placing cylinders onto the loading area (upon a pre-placed pallet).

The manipulator control has been generically designed to provide maximum reuse capabilities, however, in the system under consideration the cylinder manipulator has a very specific task within the RMS. Its task is to retrieve a cylinder and place it upon a waiting pallet in the loading area which requires a precise set of movements. The steps the controller must take to achieve its goal, outlined below and shown in Table 5-2:

- Move up;
- Extend;
- Move down;
- Close;
- Move up;
- Move right;
- Move down;
- Open;
- Move up;
- Retract;
- Move left;
- Move down.

Up/Down 1/0	Right/Left 1/0	Extend/Retract 1/0	Close/Open 1/0
0	0	0	0
1	0	0	0
1	0	1	0
0	0	1	0
0	0	1	1
1	0	1	1
1	1	1	1
0	1	1	1
0	1	1	1
0	1	1	0
1	1	1	0
1	1	0	0
1	0	0	0
0	0	0	0

Table 5-2: Markings for the RMS Manipulator carrying out its task

5.8 Applying constraints to the object

The actuator class has been designed to be as generic as possible, as indeed is the resultant manipulator. It can be seen that this actuator object can be reused in any application. To ensure the object remains as general purpose as possible the environment specific constraints are built into a separate object which acts as an intermediary between the controller, which is task specific and the manipulator object itself. In the system under consideration, the only constraint for the cylinder manipulator is that the gripper cannot be opened when the arm is raised. Imagining the cylinders to be quite heavy, doing so could amount in considerable damage to the other objects in the system and possibly the cylinder itself.

Constraint objects act as intermediaries between the controller and the object. Messages passing from one to the other are routed *via* the constraint object. To develop a constraint it is first necessary to model all the states that can be achieved in the object under consideration. This will be applied to the cylinder manipulator which is required to retrieve cylinders from the storage unit and place them onto the loading bay.

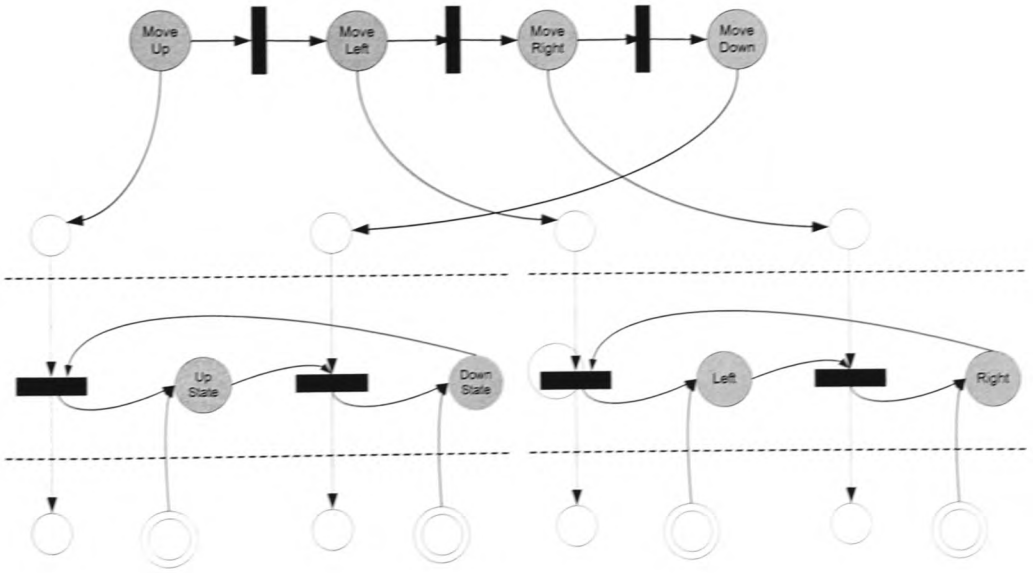
The Manipulator is composed of four actuators – direction (A1), horizontal (A2), vertical (A3) and the gripper (A4). As these are discrete event objects their states can be represented with binary as shown in Table 5-3.

A1	A2	A3	A4
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Table 5-3: All possible states of the cylinder manipulator

Actuator A3 controls the arms upwards movement and when it is enabled (1) it is in the raised position. Actuator A4 controls the gripper and when it is enabled (1) it is open. Therefore it can be seen that any series of states which gives a * * 1 1 result is a forbidden state. By analysing Table 5-3 it can be seen that this state arises four times (shaded areas).

It is intuitive to identify forbidden states in this way and control nets are cross checked against the forbidden state list to ensure no such states are embedded into the system.



5-22: A constraint applied to the controller

Figure 5-22 shows how a system specific behavioural constraint is applied to the controller which ensures that it only performs the actions required for this system. The constraint can be easily detached from the object as

it is bound at run-time ensuring the controller contains the maximum, generic reuse capabilities.

In the system utilised for this case study the overall control of the system is reduced to its simplest with the controller acting as a sequencer to co-ordinate the actions of sub-units. Functional level control is devolved to the lowest system level possible to ensure the maximum reuse capabilities are obtained.

In Figure 5-22 it can be seen that if the requirement for the manipulator to move down first and up last became necessary, the controller software would need to reconfigure to accommodate this change but the object itself remains unchanged. This is crucial to ensure the loose coupling of the objects in the systems and to retain full reuse capabilities. In the case study presented, if the requirement for the mill to operate before the lathe became important the system controller would need slight readjustment but the underlying sub-systems and the objects themselves would function without change.

In Figure 5-22 the controller for the pneumatic manipulator is shown but due to the use of public interfaces which simply control the movements of the component replacing the pneumatic manipulator with a robot arm would have no impact on the system provided the public interface of both

objects was designed to accept the same commands and give the same feedback.

5.9 Automated Code Generation

The process used to automatically generate code is based on a Boolean/decision structure that relies, in this implementation, on the net being safe, i.e. a place can never contain more than a single token.

From the coding perspective a place is represented in the application as a Boolean value, with 'true' representing a marked place and 'false' representing an unmarked place. At the initialisation stage all places are declared as Boolean type variables with their corresponding values set to the initial marking of the net.

A transition in the diagram is represented by an IF statement in the code. The IF statement's conditions are based upon the corresponding input places for the transition it represents. The IF statement, and therefore the transition, is enabled when all its input places (Boolean values) become true. The IF statement then performs the result of setting its input places to false and its output places to true. The code is controlled *via* a software application which is responsible for the generation and co-ordination of hardware linked directly to the software.

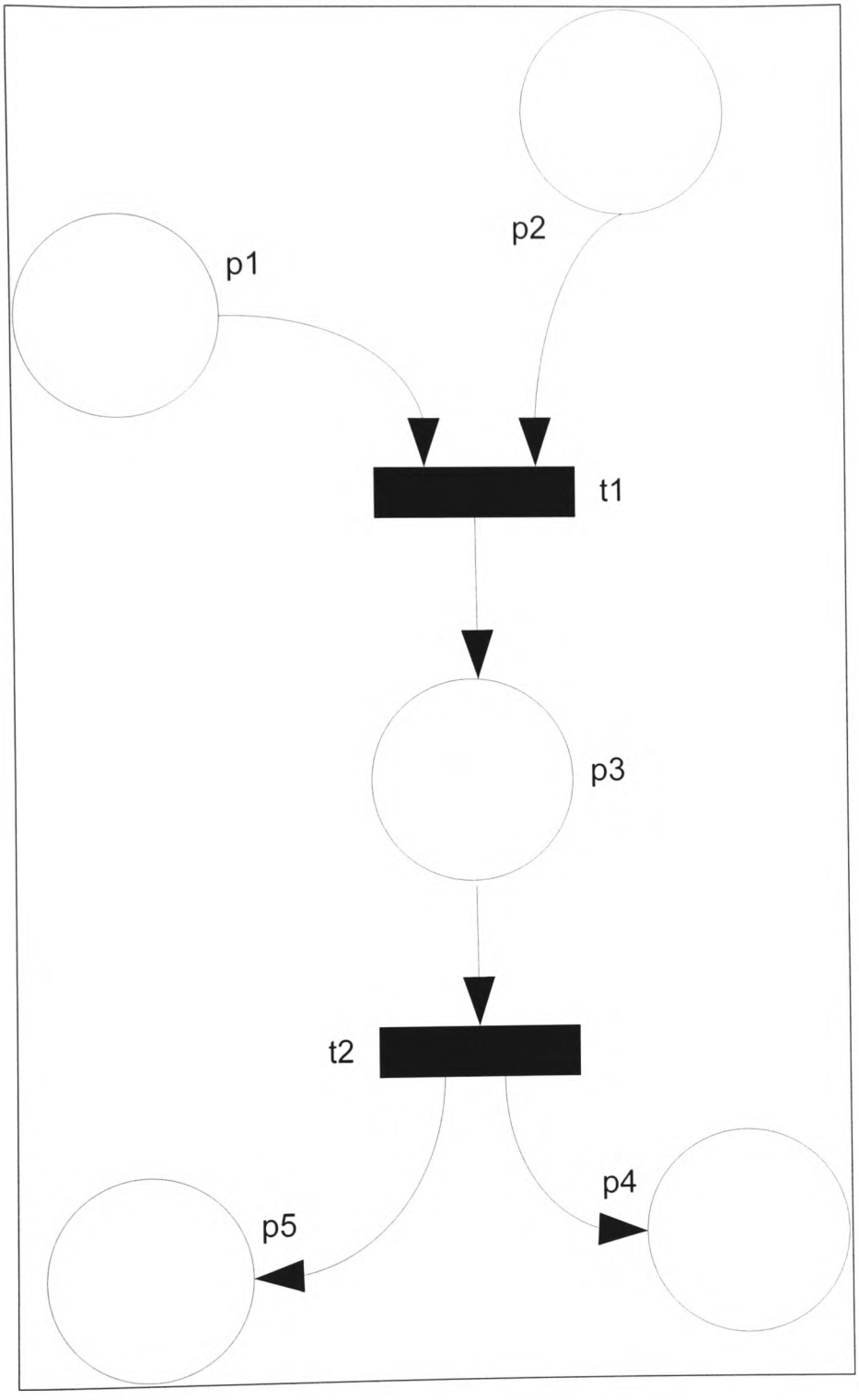


Figure 5-23: Automated code generation from Petri net models

Using the rules outlined above the Petri net listed in Figure 5-23 the following pseudo-code would be generated:

'Transaction t_1

IF p_1 = TRUE and p_2 = TRUE then

p_3 = TRUE

p_1 = FALSE

p2 = FALSE

'Transaction t_2

ELSE IF p_3 = TRUE then

p_3 = FALSE

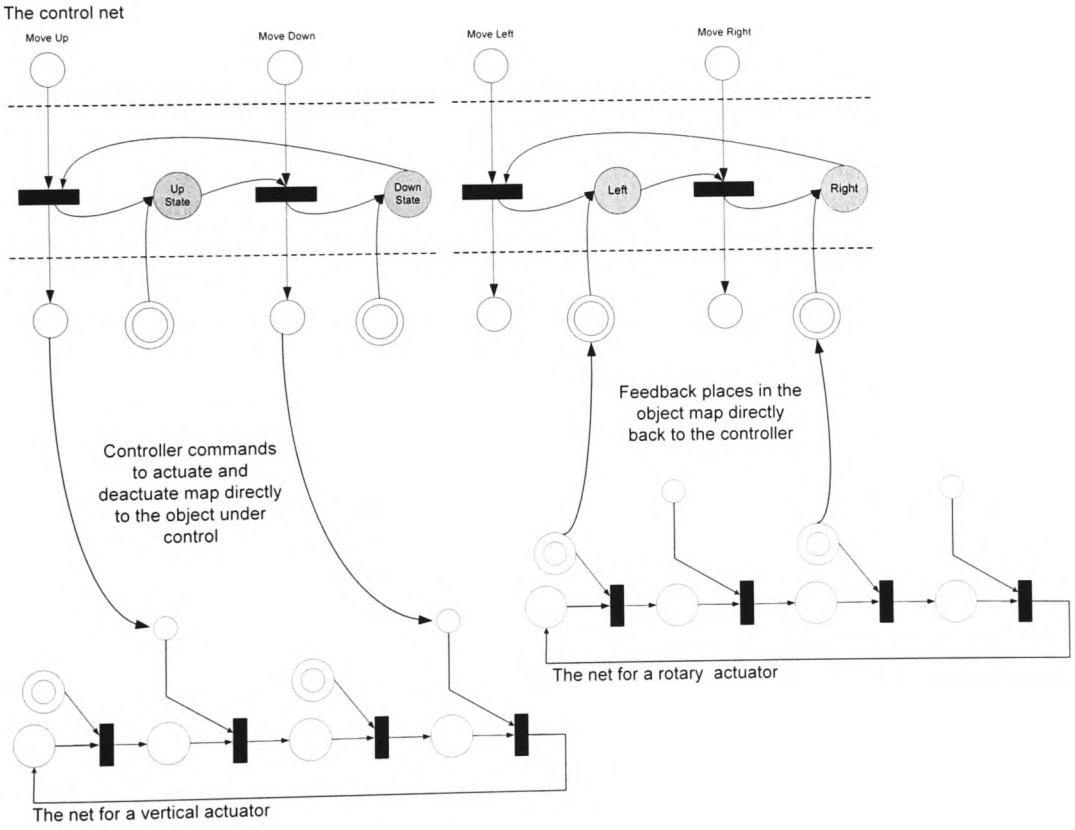
p_4 = TRUE

p_5 = TRUE

END IF

5.10 Simulation

The operation of individual actuators in Figure 5-20 can be simulated using a Petri net token player. This can be useful to establish that they are functioning correctly, and to ensure they work in the most optimised fashion. Once the system designer is certain that a unit functions to its full capability it can be placed into a class library for reuse in other projects. The controller shown in Figure 5-21 can also be simulated to ensure all of its operations function correctly, including its interactions with the pre-tested actuators.



5-24: A merged net showing how the controller and objects interact

To facilitate full simulation using token players the associated control and feedback places in the controller can be mapped directly onto their equivalents in the object as shown in Figure 5-22. The diagram also illustrates how controllers interact with their child objects.

Figure 5-22 also demonstrates how the controller is able to perform any of its available actions and is in no way constrained. This ensures that the manipulator object, which is a collection of one controller and four actuators has the capability of being used in any system. Further the actuators can also, individually be used in any system with no modifications required.

Simulation in this work is undertaken using Petri net token players. The token player allows the Petri net (and therefore the model of the system) to be stepped through by allowing a visual representation of a marking tree. Simulation in this way enables system modellers to validate and verify the operation of individual objects, sub-systems and complete systems. By merging the contol and feedback places and integrating the behavioural constraints a complete Petri net can be de described which represents the whole system. Simulation in this work facilitates what if analysis on all aspects of the system to ensure safety and enables designers to optimise system operation.

5.11 Chapter Summary

It is widely accepted that manufacturing systems need to be flexible, customisable and maintainable. This is effectively addressed in the proposed modelling technique where individual objects can be customised and updated using the key features of UML, such as inheritance, polymorphism and encapsulation while the functional requirements of the object is expressed by Petri net models.

By integrating the two types of models, the design of manufacturing systems is greatly enhanced. Manufacturing systems will be able to take advantage of the concepts of object-oriented programming that have been widely available in software engineering for some time. Future upgrades to the resultant system will be more intuitive as manufacturing design adopts the 'plug and play' philosophy of other computer systems. The technique provides a model that can be used initially as a simulation tool and later as the basis for the automated generation of the control software. Once the initial design has been carried out many objects can be reused in future systems with no requirement for additional modelling.

In a climate governed by costs and rapidity it is important to reduce the time from conception to market as stated previously, however, this rapidity cannot have any impact on product quality. Modern programming is

typified by a code and fix approach, however this technique falls down when the software will need to function in complex and rapidly changing circumstances such as those found in manufacturing organisations. The UML has gone some way to addressing many of the issues to overcome in manufacturing system design by providing a user-centric view of the system using use-case analysis and design. This facilitates effective communication between system modellers, software designers, management and all levels of staff involved in the operation of the system.

The object-oriented paradigm focuses very firmly upon design for reuse which is an important property for manufacturing systems where the market demands high quality products, at a low cost, with shortening product lives and ever increasing demands for customisation. Whilst manufacturing systems lend themselves to such an approach, the long and iterative process demanded by successful design for reuse and the time overhead spent translating models into code is at odds with the rapid approach required in global manufacturing organisations. It can be seen, therefore, that such approaches are long overdue for an automated code generation phase.

The techniques presented in this paper achieve these aims by combining the best features of object-orientation and structured Petri nets with models that are iteratively refined until the detail required for automatic

code generation are established. The modelling and development stage are integrated meaning all time spent in the initial stages is utilised all the way through to the final system.

FEMs reduce the state space explosion by removing the requirement for a complete Petri net model in order to drive the system. The modules are invoked by feedback and control places so there is an increase in message passing within the system but a complete removal of state space explosion. Each object can be tested and simulated in isolation. Modules that take advantage of a collection of objects can be tested to ensure they work together as expected. Once each object works as it should the module level controller can be tested to ensure it works correctly. Behavioural constraints can be added to control system specific factors. The overall system control can them be tested to ensure it functions correctly by invoking the systems under its command.

The case study outlined in this chapter has demonstrated how the technique developed in this work utilised the UML to *decompose* the system into a number of sub units which are then further sub-divided until the operational units are identified. In the case study the system is broken down into the Raw Materials Station, the Mille/Lathe and the Automatic Storage and Retrieval Unit. The case study focuses on the RMS which has been broken down into a number of pneumatic manipulators and

storage units. Each manipulator has then been broken down into its lowest level functional unti which is the actuator. Once the functional detail of the actuators was captured, modelled, coded and tested, they have been *composed* into a logical unti to form a manipulator which in turn forms part of the larger Raw Materials Station. As the objects within the system are considered individually from a functional perspective, the resultant models are *understandable* to stakeholders at all levels of the system, due to their small size and limited scope.

References

Bittner, K. 2003. *Use Case Modelling.* Boston, USA: Addison Wesley. 0201709139.

Booch, G., Rumbaugh, J. &Jacobson, I. 2005. *The Unified Modeling Language User Guide. 2nd edn.* USA: Addison Wesley Longman. 0321267974

Delatour, J. &Paludetto, M. 1998. UML/PNO: A Way to Merge UML and Petri Net Objects for the Analysis of Real-Time Systems. *Lecture Notes in Computer Science*, **15**(43), pp. 511-514.

Meyer, Bertrand. 1997. Object-oriented Software Construction. 2nd edn. London: Prentice-Hall. 0136291554.

Pressman, Roger S. 2004. Software Engineering a practitioner's approach. 6th Ed. London: McGraw-Hill. 0071238409.

Sommerville, I. 2006. Software Engineering. 8th end. London, UK: Addison-Wesley. 03211313798.

Stanton, M. J. 1999 .Doctoral Thesis: Structured Petri Nets for the Design and Implementation of Manufacturing Control Software with Fault Monitoring Capabilities. University of Wales College, Newport.

Yourdon, E. 1994. *Object-Oriented Systems Design.* New Jersey, USA: Prentice-Hall. 0136363253

Chapter 6
Conclusions, Analysis of Findings and Future Work

In this concluding chapter this work will be evaluated and it will be demonstrated that this thesis provides several contributions to the field of manufacturing system design. Functionally Encapsulated Models implemented within a three level control architecture are evaluated against the findings of this research. The Behavioural Constraint method outlined in this work is shown to reduce design times by enabling all system designs to make use of a library of off-the shelf components. It is proposed that the techniques outlined in this work go some way towards overcoming the problems involved with an integrated approach to CIM implementations. This chapter proves how the novel modelling technique developed in this work addresses all of Meyer's criteria for modularity. The benefits of object-orientation for manufacturing systems modelling are reinforced and the findings of this work are conclusively evaluated Finally, some discussion is given for areas of future research within this field.

6.1 Introduction

Manufacturing has changed dramatically since the mass production era of the Second World War. As demand for high, quality, low cost and customisable products increased in the post war years Computer Integrated Manufacturing (CIM) was introduced. The integration of computers into manufacturing helps organisations to centralise their data and control. Computers are able to interact with all levels of the manufacturing design process. They key factors for organisations wishing to compete in a twenty first century global economy are established in this work as:

- **Speed**. Organisations need to get their products onto the market as quickly as possible. This requires design or redesign of manufacturing systems in the shortest possible timescale;
- **Cost**. Production costs need to be reduced to the minimum possible, whilst maintaining quality;
- **Quality through Consistency**. Consistency throughout the organisation can improve the quality of its processes and greatly aid in communication between end users and system designers.

It has been established that the ability for manufacturing organisations to be able to quickly refocus their systems is vital. Indeed this is almost as important as being able to design totally new systems.

6.2 *Modularity for Manufacturing*

Modularity is identified in this work as an important concept which can help manufacturing system designers achieve flexible systems that can be easily reconfigured. Modularity is also an important factor in the speed of design and development of systems as it enables distributed team development. The main benefits of modularity identified in this work are outlined below:

6.2.1 Hardware/Software Objects

The concept of a hardware/software object enables system designers from manufacturing to utilise technologies available in software engineering. This concept is a vital component in this work as it enables development of complex systems containing hardware and software to be thought of as exclusively software problems.

6.2.2 Removal of Islands of Automation

Islands of automation, which have plagued CIM implementations, generally arise where sub-systems are redeveloped with no thought to how they will interact with the rest of the system. Modularising the system, and defining public interfaces to the objects within them, ensures that the 'islands of automation' problem is completely addressed. As long as interfaces are defined by what the module does rather than how it does it all sub-systems are able to inter communicate.

6.2.3 Minimised Disruption from Upgrade or Redesigns

The use of public interfaces will also reduce downtime considerably as for much of the process modules are only conceptually upgraded. This solution will dramatically reduce development times and be a vital step in addressing the issues of system development speed.

6.2.4 Reusable Class Libraries

Generic and highly reusable objects will enable system builders to utilise previously designed high quality components that will rapidly decrease development times whilst maintaining quality. Utilising such pre-tested, high quality components would clearly address the need for a "first time right" design.

6.2.5 Enterprise Wide Consistent Modelling

By utilising the UML for the design of such systems manufacturing organisations can benefit from the ability of the technique to model all elements of the company enhancing communication amongst stakeholders and ensuring organisational consistency.

6.2.6 Reduced Modelling Complexity

The integration of Petri nets into the UML reduces the number of models required and solves the state space explosion problem. Such a technique provides a ready-made simulation and testing tool and lends itself well to the automatic generation of control code considerably reducing the time to implementation.

6.3 Manufacturing System Design

Many system modellers face the inimitable problem of having to cope with the recurrent need to become experts in a range of disciplines other than their own. For example, a computer system's analyst may need to analyse and design a software system for a petrochemical company, or an information system specialist may need to develop a new system for a supermarket chain.

This implies the need for rapid personal knowledge expansion, however in reality the system modeller relies on an intuitive and highly detailed progression of models which enable them to overcome the barriers and bridge the gap between those with a dedicated knowledge of the system under consideration and those with the specialist skills needed to develop the new system.

In short, system modellers need models which facilitate communication between the stakeholders at all levels within the system and those undertaking the development. These models need to be intuitive enough for all parties to understand and yet contain enough expressive power to enable the analysis and design of the system under consideration, in iteratively more complex levels of detail.

Despite considerable research into software engineering two out of eight software projects fail and fifty percent are over time and budget. This work has gone some way to establishing the reasons for system

failure and has noted that many are due to poor design methodologies. In some cases the methodology fails because the users do not understand it.

It is clear that the importance of selecting the most appropriate design methodology is paramount in any successful system implementation. From the literature it has been established that a successful design methodology should:

- Accurately capture user requirements in a manner which can be understood by all stakeholders. Each stage in the design process must constantly and consistently cross reference user requirements to ensure they are fully met;
- Support iterative refinement of user requirements into low level technical detail for implementation. This can be achieved by a hybrid top-down/bottom-up approach;
- Endeavour to minimise resource wastage. Utilising off the shelf, pre-tested components from a library of objects can achieve this goal;
- Support loose coupling of objects and should allow for modular decomposition;
- Ensure that code is well documented and adopt the relevant coding conventions to ensure subsequent maintenance, modification or upgrades are possible;

- Permit modules to be tested individually and subsequently as part of the system.

Other important points drawn from the literature are that:

- Incrementally implementing a system over time spreads costs. Implementation should be based on priority;
- Building in modular stages allows the project to be broken into smaller, more manageable pieces giving staff the time to adapt to the new system and facilitating team development;
- Implementation in small steps will have the most dramatic impact rather than redesign the whole system in one go;
- Including staff at all levels in the development will ensure they take ownership;
- It is important to make decisions for future enhancements based on actual results of previous phases;
- Goals and measurement criteria should be defined prior to each new phase of implementation.

6.4 Object-Oriented Modelling for Manufacturing

This work highlights some important benefits to manufacturing organisations for adopting an object-oriented design methodology:

- Manufacturing personnel already think of their systems in terms of objects and therefore an OO approach should prove to be intuitive;
- Simulation techniques are useful for validating control strategies and for generating software;
- Incremental development approaches reduce costs;
- OO systems utilising class libraries offer customisation opportunities and aid in system maintenance;
- Object class libraries that can be reused in other systems aid in breaking down the complexity of manufacturing system design

6.5 *Petri Nets for Manufacturing Modelling*

This work has demonstrated how a Petri net can describe a manufacturing system graphically allowing system users and designers to gain a better understanding of the complex interactions within the system.

6.5.1 Visualisation of System Events

The basic structure of a Petri net graph allows system modellers to identify and visually describe the events present in a system.

6.5.2 Modelling System States and Behaviour

The use of tokens in a marked n*et a*llow the representation of the sequence of transition firing and subsequent changes in behaviour as the system moves through the sequence of events required to achieve its goal.

6.5.3 Simulation and Optimisation

Using a token player it is possible to simulate a system hypothesis and the Petri net graph's simplicity means that it is intuitive to modify the net to carry out 'what if' analysis on the proposed system.

6.5.4 Mathematical Proof

The analysis of Petri net graphs provides manufacturing system's analysts with a method of mathematically proving designs.

6.5.5 Synchronicity and Concurrency

The models allow for the specification of systems which display properties of synchronicity and concurrency and these properties are highly relevant for manufacturing systems.

6.5.6 State Space Explosion

State space explosion can be a major drawback to the use of Petri nets exclusively for modelling manufacturing systems. Complex systems produce complex Petri nets which remove the ability of users to visualise the system.

6.5.7 Lack of Object-Oriented Modelling Power

Whilst some attempts have been made to modularise Petri nets, full object-orientation has yet to be achieved.

6.6 *Merging the UML and Structured Petri Nets*

This work presented a novel methodology for a combined object-oriented and Petri net approach to the development of manufacturing systems. This is one of the main contributions of this work.

A novel technique, entitled Functionally Encapsulated Modules, utilises Petri net graphs to model the functions of each object. This allows system designers to capture both the state and dynamics of an object in a single visual representation. It also allows for each module to be simulated for testing purposes.

Using structured Petri nets which allow for modelling of control and feedback signals considerably reduces the complexity of the resultant Petri net graphs. This goes some way to reducing the state space explosion problem inherent in large complex systems.

The technique developed addresses many of the methodology issues highlighted with manufacturing system design:

- User requirements are iteratively captured using a series of use case diagrams and scenarios. A top down, abstracted view of the system from the perspective of its goals is initially taken. This view is then refined to establish more and more detail about the system. The use case models are intuitive for all stakeholders and ensure clear communication between technical and non-technical

personnel. The use cases can be cross referenced at each stage of the design process to ensure that the system adheres to the user requirements;

- Once the system has been modularised a bottom up approach is taken to capture the capabilities of each system object. Viewing the objects as independent entities ensures their full functionality is captured. Object controllers are developed for individual objects or groups of objects which are inter-dependant. This facilitates the building of a library of generic and reusable classes which can be utilised in other systems or later in redesign processes;
- Communication between objects is only undertaken *via* public interfaces in the objects. This is facilitated by control and feedback places in the Petri net structure. At implementation stages the control and feedback places are coded as public operations. This feature ensures that systems are loosely coupled. Loose coupling in this case will ensure that changes to objects in the system have a minimal impact on other objects. Objects can be used based on what they do rather than how they do it;
- Objects and modules can be individually tested using the token player facilities of Petri net graphs. Upon system integration the entire system can be simulated using the same method;
- The well defined interfaces presented by FEMs enable system designers to incrementally upgrade parts or all of a system.

6.7 Contributions of this Research Work

The original contributions of this work are evaluated below. It is demonstrated how the original methodology and modelling tool developed in this thesis satisfy Meyers five criteria for modularity (1997).

6.7.1 The Application of the UML to Manufacturing Systems

The Unified Modelling Language has successfully been applied to a complete manufacturing system. (Llewellyn et al, 2000). This has proven benefits for manufacturing organisations including the provision of a reusable system, and the opportunity to build a library of classes, which makes subsequent designs or modifications to existing systems more intuitive.

Using the technique presented in this theses it can be seen that the UML provides manufacturing organisations with the full benefits of object-orientation including encapsulation, inheritance and the ability to use class hierarchies.

By focusing on the objects and their interactions via a public interface, the dynamics of the system can be presented to technical and non-technical users, allowing the designer to focus on what the object/system is to do, without an in-depth knowledge of how it does it. This satisfies Meyer's criteria for modular understandability.

The UML also facilitates the unique ability to model all aspects of a manufacturing organisation from business processes through to shop floor machinery.

6.7.2 A Methodology for Incremental Implementation

An incremental approach to the analysis of CIM systems enables manufacturing organisations to computerise anything from individual manufacturing workstations through to entire departments on a staged basis.

Manufacturing organisations adopting this approach will see a reduction in the development times for new systems and for redesigns.

Use-case analysis ensures user requirements are accurately captured and in a format which enhances communication between system modellers and stakeholders. This satisfies Meyer's criteria for modular understandability.

The design stages of a use-case driven approach take into account the needs of all levels of the workforce, ensuring all personnel are involved in the process.

The initial use-case scenarios used to capture the system requirements can be reused at the testing stage to verify all requirements are adequately met.

6.7.3 Development of a Three Level Control Architecture

The hybrid bottom-up and top-down approach of the incremental methodology proposed enables the controllers required at all levels of the system to be adequately modelled and ensures the functionality of the system is maintained.

The methodology developed in this work distributes the complexity of system control across the sub-systems. This ensures that most of the objects and modules within the system can be reused with little or no changes. This achieves Meyer's criteria for modular composability.

The object-oriented approach to the system design allows designers to capture the system at its most generic, but also provides a method of capturing the dynamics of the system.

Utilising token players each object and module can be tested independently before they are integrated into the complete system. The final system can also be simulated for optimisation testing.

Behavioural constraints ensure that all objects in the system can be instantly stored in a reusable class library that will enhance the speed and quality of subsequent system designs.

The constraint objects also server the purpose of capturing error conditions. This goes some way towards achieving Meyer's criteria for modular protection.

6.7.4 Merging the UML and Petri Nets

A technique for successfully combining the UML and Petri nets has been developed called Functionally Encapsulated Modules (FEM) (Llewellyn et al, 2001).

The technique is superior to other attempts to merge these two powerful modelling tools in that it supports the full range of object-oriented capabilities.

The FEMs also enable UML designers to utilise the functional modelling power of Petri nets.

State space explosion is reduced by modelling only parts of the system, i.e. the operations of objects. However, utilising control and feedback places ensures the system meets the criteria of modular composability.

FEMs reduce the number of diagrams required to model both state and behaviour of systems.

The FEMs develop a unique method of capturing the attributes of both software and hardware which can be intuitively implemented into any manufacturing system.

The encapsulation of hardware and software with a distinct user interface allows the designer, and the users of the system, to visualise the objects that make up the system's model without worrying about the

inherent complexity. Public interfaces also satisfy Meyer's criteria for modular continuity.

6.7.5 Simulation and Automated Code Generation

Each object and module can be tested independently by comparing their use case scenarios against the simulation tool provided by the Petri net models.

The whole system can be tested against the goal of the system against the token player aspect of Petri nets.

The system can be reconfigured intuitively by adjusting the Petri net graphs to test out control optimisation scenarios. As the code is based directly on the graphical notation the code will reconfigure.

A method for mapping Petri net diagrams to pseudo-code is presented in this thesis. It is an intuitive method which corresponds well with discrete event systems.

6.8 Thesis Conclusions

It is widely accepted that manufacturing systems need to be flexible, customisable and maintainable. This is effectively addressed in the object-oriented system outlined in this these where individual objects can be customised and updated using the key features of UML, such as inheritance, polymorphism and encapsulation.

By integrating the two types of models, the design of manufacturing systems is greatly enhanced. Manufacturing systems will be able to take advantage of the concepts of object-oriented programming that have been widely available in software engineering for some time.

Future upgrades to the resultant system will be more intuitive as manufacturing design adopts the 'plug and play' philosophy of other computer systems. The technique provides a model that can be used initially as a simulation tool and later as the basis for the automated generation of the control software.

Once the initial design has been carried out many objects can be reused in future systems with no requirement for additional modelling.

In a climate governed by costs and rapidity it is important to reduce the time from conception to market as stated previously, however, this rapidity cannot have any impact on product quality. The UML has gone some way to addressing many of the issues to overcome in

manufacturing system design by providing a user-centric view of the system using use-case analysis and design. This facilitates effective communication between system modellers, software designers, management and all levels of staff involved in the operation of the system.

The object-oriented paradigm focuses very firmly upon design for reuse which is an important property for manufacturing systems where the market demands high quality products, at a low cost, with shortening product lives and ever increasing demands for customisation. Whilst manufacturing systems lend themselves to such an approach, the long and iterative process demanded by successful design for reuse and the time overhead spent translating models into code is at odds with the rapid approach required in global manufacturing organisations. It can be seen, therefore, that such approaches are long overdue for an automated code generation phase.

The techniques presented in this thesis achieve these aims by combining the best features of object-orientation and structured Petri nets with models that are iteratively refined until the detail required for automatic code generation are established. The modelling and development stage are integrated meaning all time spent in the initial stages is utilised all the way through to the final system.

6.9 *Future Work*

As with all research work there is still much to be done before the tools proposed in this work can be applied to manufacturing systems.

6.9.1 Development of a Graphical Modelling Tool

Though many prototypes and simulations have been conducted by the author, there is a need for the development of the graphical modelling tool.

Many token player tools have been analysed during the progress of this work but due to the complexity of modelling even small systems with Petri net graphs they either are limited to a restricted number of places or become difficult to interpret.

Utilising the FEM approach outlined in this work the software would focus on individual objects and their operations. This would considerably reduce the complexity of the diagrams and make the screen layout more intuitive.

The tool should encompass the methodology proposed in this work and should enable a system designer to document all stages of the design process.

6.9.2 Expansion of the Modelling Power of the Petri net Graphs

This work has focussed on the use of Structured Petri nets due to their ability to model control and feedback places. Petri net theory is being continuously developed and a number of interesting possibilities for modelling time factors are being researched.

The scope of this work could be expanded to explore the potential of this technique for modelling systems with a time dimension such as real time systems.

6.9.3 Development of an Automated Coding Tool

This work has presented a method of mapping Petri net diagrams to pseudo-code. Whilst the resultant code can be interpreted into a range of programming languages it would be useful to have a tool which could generate the code directly.

Such a tool could generate pseudo-code, as in this work, which could then be output into a range of programming languages based on add-on modules.

The code generation work in this thesis takes no account of code optimisation. Much work is being undertaken into the area of Aspect Oriented and Generative Programming. This is certainly and area in which this work could expand.

6.9.4 Regenerative Coding for Autonomous Robots

The Petri net diagrams in this thesis can be directly mapped to the code required to control the system. Changes to the diagram result in corresponding changes to the code.

Research needs to be conducted into the feasibility of expanding this idea to autonomous robots where the code can regenerate in response to factors such as environmental considerations.

References

Llewellyn, E.W., Stanton, M.J., Roberts, G.N. 2000. *Towards the implementation of the Unified Modelling Language (UML) into a Computer Integrated Manufacturing (CIM) environment.* Fourteenth International Conference on Systems Engineering. 12th – 14th September 2000. Coventry, UK, pp 398 – 403.

Llewellyn, E.W., Stanton, M.J., Roberts, G.N. 2001. Discrete event systems design based upon the UML and Petri net objects. *3rd Workshop on European Scientific and Industrial Collaboration.* 27th – 29th June 2001. Twente, The Netherlands, pp. 211-219

Llewellyn, E.W., Stanton, M.J., Roberts, G.N. 2003. A combined object-oriented and structured Petri net approach for discrete event systems' design. 4th Workshop on European Scientific and Industrial Collaboration. 28th – 30th May 2003. Miskolc, Hungary, pp. 398-403.

Meyer, Bertrand. 1997. Object-Oriented software construction. 2nd edn. London: Prentice-Hall. 0136291554.

Appendix 1
Papers

Appendix 1 contains copies of all published papers taken from this research. They are summarised as follows:

LLEWELLYN, E., STANTON, M.J. and ROBERTS, G.N. 2000. Towards the implementation of the unified modelling language (UML) into a computer integrated manufacturing (CIM) Environment. International Conference on Systems Engineering 2000 (ICSE2000). Coventry. Vol. 2, pp 398-403.

LLEWELLYN, E., STANTON, M.J. and ROBERTS, G.N. 2001. Discrete Event Systems Design Based upon the UML and Petri Net objects. 3rd Workshop on European Scientific and Industrial Collaboration (WESIC 2001) Enschede, The Netherlands, pp 211-219. ISBN 90 365 16102.

LLEWELLYN, E., STANTON, M.J. and ROBERTS, G.N. 2002. Nine-step approach to designing successful visual programming applications. Computing and Control Engineering. (13)2 pp 82-86. ISSN Number 0956-3385.

LLEWELLYN, E.W., STANTON, M.J., ROBERTS, G.N. 2003. A combined object-oriented and structured Petri net approach for discrete event systems' design. 4th Workshop on European Scientific and Industrial Collaboration. 28th – 30th May 2003. Miskolc, Hungary. ISBN 963 661 570 5.

TOWARDS THE IMPLEMENTATION OF THE UNIFIED MODELLING LANGUAGE (UML) INTO A COMPUTER INTEGRATED MANUFACTURING (CIM) ENVIRONMENT

Llewellyn, E.W.[1], Stanton, M.J.[2] and Roberts, G.N.[1]

Mechatronics Research Centre[1]
Department of Engineering[2]
University of Wales College, Newport
Allt-yr-yn Campus. PO Box 180, Newport, NP20 5XR. U.K.
Tel: ++44 (0) 1633 432487 Fax: ++44 (0) 1633 432442 E-mail: eric.llewellyn@newport.ac.uk

Keywords: Unified Modelling Language (UML), Computer Integrated Manufacturing (CIM), Object-Oriented software, Computer Aided Software Engineering (CASE).

Abstract

A methodology based on the use of the Unified Modelling Language (UML) for the modelling and design of control software for a Computer Integrated Manufacturing (CIM) environment is presented. This is demonstrated by modelling a pneumatically controlled manipulator comprising of four separate actuators, which forms part of a pneumatic station - an integral component in the University of Wales College, Newport's (UWCN's) CIM system. The major causes of CIM implementation problems along with their possible UML solutions are identified and it is shown that the resultant models allow the designer to capture the static, dynamic and behavioural attributes of the system. This provides organisations with a unique opportunity to develop a system that, if required, can be used initially as a simulation tool and later as the basis for the development of control software.

1 The problems facing manufacturing organisations in the 21st Century

The latter part of the last century saw a paradigm shift from mass production to customisation. Waldner [1] describes how the markets in which manufacturing organisations compete have become increasingly complex and diverse with highly customisable, small-scale products replacing mass production. To compete in the global markets of the 21st Century, industry needs to produce new products to customer's requirements with the shortest possible lead and delivery times, to the highest possible quality, and at the lowest possible price [2]. During the 1970s manufacturing focus shifted from productivity to flexibility and quality. To compete with these changes the Japanese attempted to increase flexibility by reducing the administrative procedures involved in management and control. These ideas also revealed that over-automation of processes could have undesirable consequences. Often it was found that functions performed by very complex automated systems could equally well be achieved by elementary manual procedures or by simple mechanical devices. The notion of system and process simplicity had been lost. Computerisation became an automatic response in companies, with no attempt being made to find a less complex alternative to a proposed procedure [1,1]. This in turn has led to so called 'islands of automation' where isolated cells of computer controlled machines are unable to link together [3]. Many factors have exacerbated this problem, such as the undocumented internal departmental solutions to problems, the ad-hoc acquisition of hardware and software and the lack of standardisation among vendors. Communication among these 'islands' is often made via media such as disks, CD-ROMs or hard copy [4] taking valuable time and effort and adding to the overall inefficiency of the system.

2 Computer Integrated Manufacturing (CIM) as a solution

Computer Integrated Manufacturing (CIM) aims to address the problems created by linking these 'islands' into a single system. Hannam [3] neatly defines CIM as "the integration of business, engineering, manufacturing and management information that spans company functions from marketing to product distribution." Clearly CIM is a way for companies to compete in the present global context. It is important to note however, that CIM is a goal, which cannot be purchased off the shelf, since its application will be unique to each company [5]. CIM itself has brought a whole host of new problems. Design of control software for CIM systems by traditional methods is typically characterised by high installation costs and long lead times. The resultant software is often difficult and costly to maintain, making the system limited in functionality and almost impossible to expand [6]. The software for CIM systems needs to reflect the flexible nature of the systems themselves. It should be easy to design, maintain and upgrade. The code should be *modular* so as to be reusable, and *general* to ensure the interface remains unchanged [7]. These techniques are already in use in software engineering and have been enhanced via the use of Computer Aided Software Engineering (CASE) tools.

3 Why use the UML in a manufacturing context?

Manufacturing systems can be complex and varied in nature due to the wide range of interconnected objects and the myriad of messages passing between them. It follows therefore that manufacturing software needs cannot be met by general purpose 'off the shelf' packages. One approach is to design generic solutions, and then to customise them to the requirements of each company. The resultant generic object class libraries are customisable through object-oriented techniques, and provide a good starting point for the design of practical control software. This abstraction of complex manufacturing systems into a series of objects is more intuitive because manufacturing end users already consider their systems in terms of objects, i.e. parts, conveyers, lathes, or drilling machines etc. It is widely accepted that CIM systems need to be flexible, customisable and maintainable [8]. This is effectively addressed in an object-oriented system where individual objects can be customised and updated using the key features of the UML, such as *inheritance*, *polymorphism* and *encapsulation*. Further, a common object model for the design of CIM systems provides a method of incremental implementation where the building of custom applications from a common repository of software objects helps to achieve conceptual integration. The conceptual design allows for 'what-if' analysis to be carried out on any proposed system before implementation saving the company time, money and effort. Gunasekaran and Thevarahjah [9] identify three key stages in the successful implementation of CIM i.e. simplification, integration and computerisation. The UML aids in all three areas as will be demonstrated.

In order to capture all aspects of the system under investigation, it is important for the designer to capture three aspects of it i.e. its static state, its dynamic state and its behaviour. The static state of a system describes the objects that it is comprised of, and how they relate to each other. The dynamic state describes how these components interact to make the system serve its purpose. Finally the behaviour of a system describes the states a component may be in at each stage of its operation. The term behaviour has been deliberately used to demonstrate the fact that it may be necessary to model undesirable states. For example, a crane should never be allowed to open its gripper when in the air, if it is carrying a two ton weight!

4 A simple pneumatic actuator system based system

As a practical example the Pneumatic Station, part of the University of Wales College Newport's CIM system will be used. The 'building block' of this section of the system is the pneumatic actuator. A pneumatic actuator can be either actuated or de-actuated. These *operations* take place when a Programmable Logic Controller (PLC) opens a valve, which pumps air into the actuator, thus actuating it. When the PLC closes the valve, the air is removed and the actuator de-actuates. This description provides a basic overview of the system, which is all that is required in order to begin the design of the system.

5 Class identification

From an object-oriented point of view, the system described consists of a series of classes. Booch et al [10,10] define a class as "a description of a set of objects that share the same attributes, operations, relationships and semantics." An object then, is a "unique instance of a class" [11]. Two other important properties of a class are:

- Operations:- which are "used to read or manipulate the data of an object" [12] and;
- Attributes:- "the structure of the objects: their components and the information or data contained therein" [13]

From the brief description given it is possible to identify the following classes: - PLC, Valve and Actuator. These classes can now be examined in further detail.

5.1 The Actuator Class

The actuator can be in either one of two final states - actuated or de-actuated. It can also be midway between these states, i.e. it can be in the process of actuating or de-actuating. Therefore the actuator requires two operations, one to carry out the action of actuating and one to carry out the action of de-actuating. In addition, if the system is to provide feedback it must allow *external entities*, such as a controller, to interrogate the actuator to determine its current state. It is therefore possible to establish that an actuator has a state *attribute* and an operation that provides that state to external entities. An important concept for attributes is that of *visibility*. Visibility applies to attributes and operations, and specifies the extent to which other classes can use a given class's attributes or operations. Schmuller [14] identifies three levels of visibility. At the *public* level, usability extends to other classes (represented by a "+" symbol). At the *protected* level, usability is open only to the classes that inherit from the original class (represented by a "#" symbol). At the *private* level, only the original class can use the attribute or operation (represented by a "-" symbol). The actuate and de-actuate operations are called by the Valve class and therefore are public, as is the getState operation. Generally, classes are shown with the first letter of each word in uppercase. Attributes and operations usually start with a lower case letter. Figure 1 shows a class diagram for a class of type Actuator.

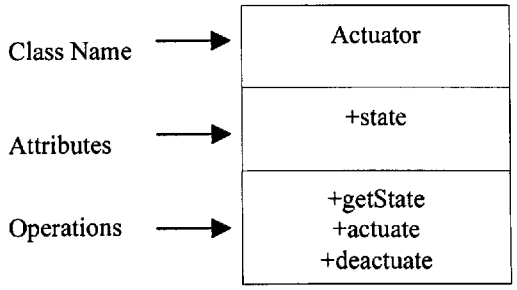


Figure 1: The Actuator class

In addition a state diagram shows all the possible states an actuator can have. This helps to identify all the possible values the state attribute can take.

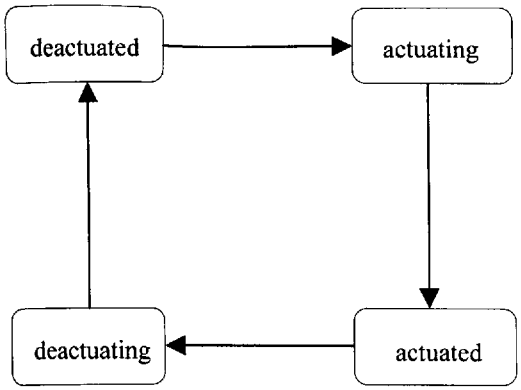


Figure 2: The Actuator state diagram

From figure 2 it can be ascertained that in order to arrive at the desired states of actuated or de-actuated, the actuator must pass through a 'working' phase where it is either actuating or de-actuating. It is important for the system sequence controller to be aware of these states, so that it does not try to invoke the operations of a busy object.

5.2 The Valve Class

From the description it can be seen that the valve can be either open or closed, and again there must be the intermediate steps of opening or closing. For feedback purposes it will be necessary to establish the current state of the valve. The class and state diagrams for the valve are similar to those for the actuator. From the description it becomes apparent that the operation of opening and closing the valve is controlled by the PLC class, therefore it is logical to assume that the open and close methods are public, as is the valve's state attribute.

5.3 The PLC Class

PLC's act as controllers in our system by instructing other objects to perform their operations. The PLC class works with a predetermined sequence of object method calls such as those shown in the following *pseudo-code* fragment:

```
BEGIN
   'Wait until the Valve is not busy
   DO
      InstanceOfValve.GetState
   LOOP UNTIL(InstanceOfValve.GetState)=
      "Closed"
   'Open the Valve
   InstanceOfValve.Open
   ..
   ..
END
```

In this instance the PLC class needs only to start or stop its predetermined sequence of events. In the example system described it is imagined that a human operator controls these operations, probably via some form of stop and start buttons, and therefore these operations (start and stop) are public. It would be reasonable to assume that in other circumstances the controller could be another object in the system. This is known as an external entity, which may well be an integral part of the larger system. Decomposition into subsystems allows the external entity or *actor* to be represented with the stick figure shown in figure 4. This indicates that while it is understood as an important object, which needs representation, its complexity need not be modelled at this stage. It is suffice to know that it performs the action of starting and stopping the PLC. The ability to *generalise* in this way enables the system designer to plan for various types of implementation. For example, the PLC could be controlled by a human, another PLC or a computer. The diagram would not need to change in any of these circumstances. The functional details of how the PLC works are encapsulated within the Class. In order to interface an *instance* of class PLC with a controller all the required information can be accessed via the public operations and attributes. These represent the interface between the Class and the outside world.

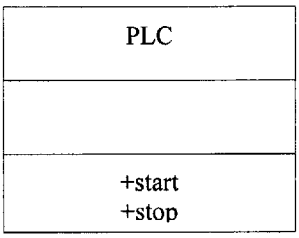


Figure 3: The PLC class

6 The associations between classes

The brief description of the system provides enough detail for a diagram outlining the various associations between the classes. Each of these associations represents a method call to another object. By analysing these associations it is possible to establish how objects in the subsystem interact with each other. It is also possible to establish how the subsystem itself interacts with objects outside of its *domain*. In order to make the diagram clearer, items such as the attributes and operations of each class are not displayed. As previously stated the external entity in this system could in fact represent a human or another object. The external entity in this instance represents an interface to another object in another part of the larger system being considered.

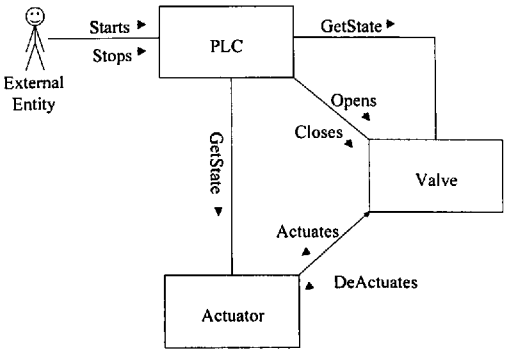


Figure 4: The Interaction between classes

7 The dynamic capabilities of a pneumatic actuator

The method calls between objects can be more clearly seen on a sequence diagram, which also shows the order in which the operations are invoked (figure 5). The diagram gives a pictorial representation of the two possible final states of a pneumatic actuator, and the procedure for arriving in those states, i.e. the actuator being actuated, and the actuator being de-actuated. This model shows the functional detail of the dynamics of the actuator. Once this information has been captured it is possible to write all the functional code and forget about the complexity involved in the actions of our actuator. This 'code and forget' approach means it is no longer necessary to consider the PLC or the valve, instead the system designer can concentrate on the detail of what the actuator is intended to do, as part of the greater system. However, as the functionality of the system increases in complexity, or the system becomes larger, these diagrams become long, complicated and unwieldy.

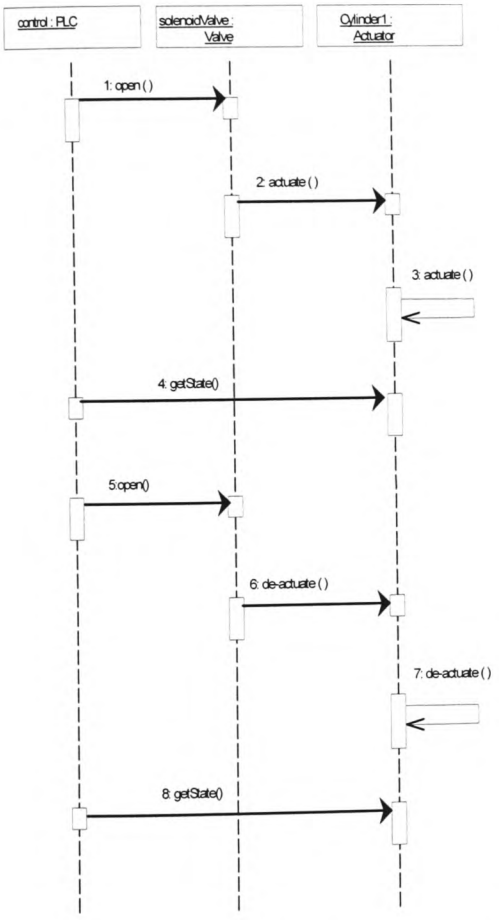


Figure 5: The sequence of available operations for a pneumatic actuator

The following text presents the idea of *Functional Encapsulation Modules (FEMs).* The whole dynamic process, represented by figure 5, can also be represented by using a single pair of FEMs as depicted in figure 6. It can clearly be seen that this provides a convenient way of describing these operations.

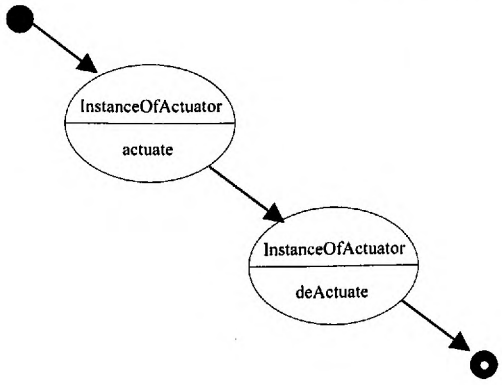


Figure 6: A FEM for the Actuator's two operations

These FEMs can now be encapsulated into whichever software language is used to control the system. It is envisaged that the FEMs could be implemented in an application that with its 'drag and drop' interface would allow system designers to manipulate the operations of the actuator with no thought to the technical detail. The important concept is that by manipulating easy to understand graphical symbols, the user is also manipulating the associated code. At the same time as they are remodelling the system, they are regenerating the code to control it. The idea of flexibility and quality can be taken to their extremes. A system can be redesigned, optimised, upgraded or maintained and be back on line with new code in a fraction of the time taken with traditional software design methods.

8 Expanding the idea to a manipulator

The idea of using the UML and FEMs will now be expanded to show its use on a manipulator subsystem of the UWCN CIM system.

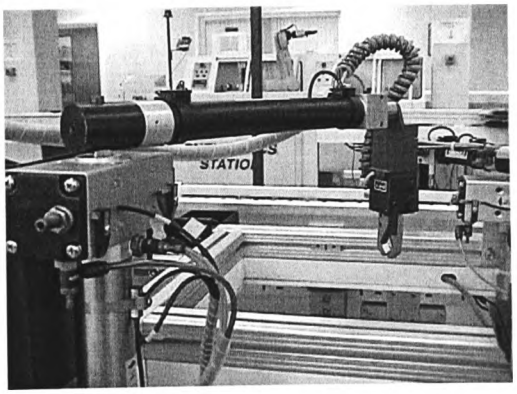


Figure 7: A Manipulator comprising of a series of pneumatic actuators

Schmuller [14] indicates that "sometimes a class consists of a number of component classes. This is a special type of relationship called an *aggregation.* The components and the class they constitute are in a whole-part relationship." A manipulator shares such an aggregation with a set of actuators, where the manipulator is the whole and the actuators are the parts. In the four-actuator model shown in

figure 7, each of the actuators can carry out the same basic function, i.e. they can actuate or de-actuate. However, the same operation call has differing effects on the action being performed by its recipient object. For example actuating an actuator can raise it, move it left, or open it. This sharing of an operation is called polymorphism. Schmuller [14] describes polymorphism as the situation where "an operation has the same name in different classes" and "each class 'knows' how that operation is supposed to take place." Figure 8 describes the aggregation relationship between a manipulator and its actuators. It can be seen that any particular one instance of the class Manipulator contains at least one and up to *n* actuators. A Manipulator is an aggregation of 1 to *n* Actuators. The UML denotes the many end of a relationship with the "*" symbol. The use of 1 in the relationship, i.e. 1..n, denotes the fact that the relationship must have at least 1 and, in this case, up to many actuators to each 1 manipulator.

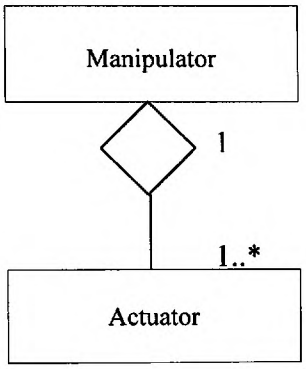


Figure 8: A Manipulator as an aggregation of Actuators

8.1 A practical example

An instance of class Manipulator may comprise of four pneumatic actuators as shown in figure 9. It can be clearly seen in this diagram that the four separate actuators whilst all having the same basic characteristics, are slightly different. This raises another important object-oriented concept, that of *inheritance*. As Yourdon [15] defines it "[inheritance] allows an object to incorporate all or part of the definition of another object as part of its own definition."

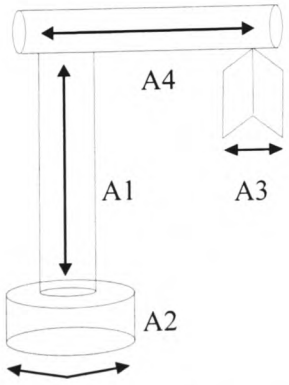


Figure 9: A Manipulator

The Class Actuator used to make up the Manipulator above is actually decomposed into three *subclasses* or child classes - a LinearActuator, RotaryActuator and Gripper (figure 10).

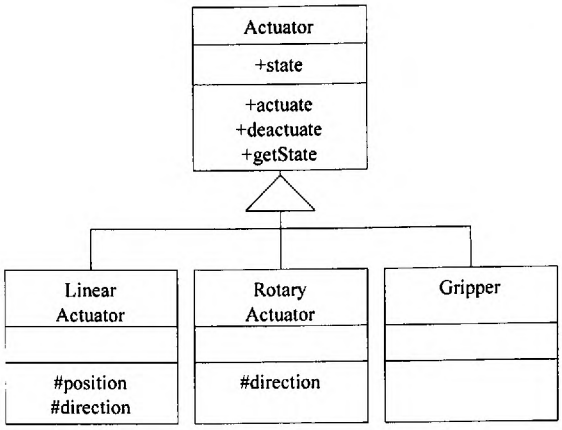


Figure 10: Three subclasses inheriting from their parent class

Each subclass inherits all the attributes and operations of the Actuator class and each add their own unique ones. For example the LinearActuator Class adds the position attribute which enables the actuator to have a horizontal or vertical position. The RotaryActuator has a direction which enables it to find out the direction of travel when the actuate operation is carried out. The Gripper will open when actuated and close when de-actuated. This demonstrates the principle of polymorphism, each of the Actuators has an actuate operation, but each reacts differently when called. These new attributes are visible only to the creating class and are therefore *protected*.

In the UWCN CIM system, the manipulator needs to pick up a metal cylinder from a tray and place it on a waiting palette. From its start position over the tray, the manipulator needs to follow this sequence of events:

```
Open the gripper; Move down; Close the gripper
(with a tube held); Move up; Move right; Move
down; Open the gripper (dropping the tube);
Move up; Close the gripper; Move left
```

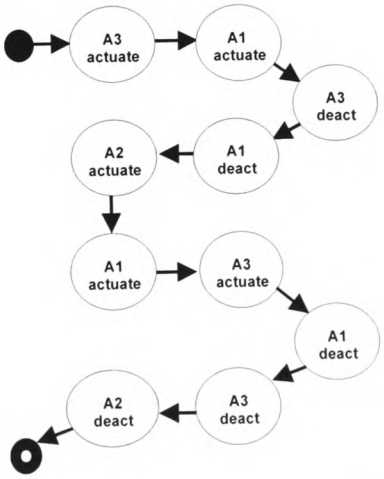


Figure 11: A FEM showing the operations required to move a cylinder between two points

The diagram in figure 11 represents this sequence of events, when mapped into actuator movements, as a FEM. It is worth noting again that if the FEM were to be implemented in a software application, the required control code would also be generated in the target language.

9 Conclusions

Presented in this paper are the building blocks for designing and maintaining manufacturing systems based on the UML. The method presented allows for a complex system to be broken down into subsystems and modelled in detail. This not only simplifies the task of modelling large, complex systems, but also allows for the building of a library of classes. These classes can then be used for developing new systems or for upgrading and maintaining existing systems.

The idea of FEMs proposes a method of modelling the overall control of related subsystems by controlling the sequence of events that needs to take place in order for that system to perform some task. It is envisioned that a software application developed around the concept of FEMs will enable relatively inexperienced users to design and alter manufacturing systems. The structure of the UML diagrams lend themselves well to the simplification of developing optimised control code in a high level language and provide the possibility of the automation of this task.

10 Future work

An investigation into the application of the UML into real time systems will be carried out, with particular attention to the reduction of redundant code normally produced by object-oriented techniques [16]. The idea of integrating structured Petri nets [17] into the UML diagrams offers many benefits including a formal validation method [18] and a method of modelling concurrent and non-deterministic systems [19]. The generation of control code from the model will be implemented and tested on the Pneumatic Station, at the University of Wales College, Newport.

11 Reference list

[1] Waldner, J. (1992) Cim. Chichester, Uk: John Wiley & Sons Ltd.

[2] Davies, B.J. (1997) Cim Software And Interfaces. *Computers In Industry* **33**, 91-99.

[3] Hannam, R. (1997) Computer Integrated Manufacturing From Concepts To Realisation. Essex, Uk: Addison Wesley Longman.

[4] Jardim-Goncalves, R., Silva, H., Vital, P., Sousa, A., Seiger-Garcao, A. And Pamies-Teixeira, J. Implementation Of Computer Integrated Manufacturing Systems Using Sip: Cim Case Studies Using A Step Approach. *International Journal Of Computer Integrated Manufacturing* **10**, 172-180. (1997)

[5] Hassard, J. And Forrester, P. Strategic And Organizational Adaptation In Cim Systems Development. *International Journal Of Computer Integrated Manufacturing* **10**, 181-189. (1997)

[6] Aguirre, O., Weston, R., Martin, F. And Ajuria, J.L. Mcsarch: An Architecture For The Development Of Manufacturing Control Systems. *International Journal Of Production Economics* **62**, 45-59. (1999)

[7] Maione, G. And Piscitelli, G. Object-Oriented Design Of The Control Software For A Flexible Manufacturing System. *International Journal Of Computer Integrated Manufacturing* **12**, 1-14. (1999)

[8] Adiga, S. (1993) Object-Oriented Software For Manufacturing Systems. London, Uk: Chapman & Hall.

[9] Gunasekaran, A. And Thevarajah, K. Implications Of Computer Integrated Manufacturing In Small And Medium Enterprises. *International Journal Of Advanced Manufacturing Technology* **15**, 251-260. (1999)

[10] Booch, G., Rumbaugh, J. And Jacobson, I. (1999) The Unified Modeling Language User Guide. Usa: Addison Wesley Longman.

[11] Eriksson, H. And Penker, M. (1998) Uml Toolkit. New York, Usa: John Wiley & Sons.

[12] Martin, J. (1993) Principles Of Object Oriented Analysis And Design. New Jersey, Usa: Prentice-Hall.

[13] Oestereich, B. (1999) Developing Software With Uml. Harlow, Uk: Addison Wesley Longman.

[14] Schmuller, J. (1999) Teach Yourself Uml In 24 Hours. Usa: Sams Publishing. 0-672-31636-6.

[15] Yourdon, E. (1994) Object-Oriented Systems Design. New Jersey, Usa: Prentice-Hall.

[16] Narisawa, F., Naya, H. And Yokoyama, T. A Code Generator With Application-Oriented Size Optimization For Object-Oriented Embedded Control Software. *Lecture Notes In Computer Science* **15**, 511-514. (1998)

[17] Stanton, M.J. Structured Petri Nets For The Design And Implementation Of Manufacturing Control Software With Fault Monitoring Capabilities. University Of Wales College, Newport. (1999)

[18] Delatour, J. And Paludetto, M. Uml/Pno: A Way To Merge Uml And Petri Net Objects For The Analysis Of Real-Time Systems. *Lecture Notes In Computer Science* **15**, 511-514. (1998)

[19] Zapf, M. And Heinzl, A. (1998) Techniques For Integrating Petri Nets And Object-Oriented Concepts. Anonymous

DISCRETE EVENT SYSTEMS DESIGN BASED UPON THE UML AND PETRI NET OBJECTS

Eric Llewellyn[†], Martin Stanton[‡] and Geoff Roberts[†]

[†] University of Wales College, Newport. Mechatronics Research Centre
Allt-yr-yn Campus. PO Box 180, Newport NP20 5XR, South Wales, United Kingdom.
email: eric.Llewellyn@newport.ac.uk and geoff.roberts@newport.ac.uk

[‡] University of Wales College, Newport. Department of Engineering
Allt-yr-yn Campus. PO Box 180, Newport NP20 5XR, South Wales, United Kingdom.
email: martin.Stanton@newport.ac.uk

Abstract

A method for the design of discrete event systems is presented which combines the Unified Modelling Language (UML) and Petri Net Objects (PNO). The genericity of the model is enhanced by the addition of constraint objects that allow reusable, general-purpose classes to be developed, which can be easily tailored to specific systems. By merging the UML and PNO the resultant models allow for accurate requirements analysis and provide object-oriented designs which are reusable and mathematically provable.

1 Introduction

Manufacturing systems are complex and varied in nature and therefore their software needs cannot readily be met by general purpose 'off the shelf' packages. The approach generally adopted by software engineering practitioners is to design generic solutions, which can be customised to the specific requirements of the system. The resultant generic object class libraries are customisable through object-oriented (OO) techniques, and provide a good starting point for the design of practical control software. The abstraction of complex manufacturing systems into a series of objects is more intuitive because manufacturing end users already consider their systems in terms of objects, i.e. parts, conveyors, lathes, drilling machines etc. (Adiga, 1993). The Unified Modelling Language (UML) has become the *de facto* standard for OO analysis and design and its application to manufacturing systems has already been demonstrated (Llewellyn *et al*, 2000). Petri nets are widely used to model discrete event systems (DES) and provide a model which is mathematically provable and using a token player, also functions as a simulation tool.

2 Merging the UML and PNO

The UML uses sequence and state charts for modelling the message flow and states of the system respectively. This results in two separate models, neither of which are

mathematically provable. It is proposed that Petri net graphs can be used to capture both the message passing and states of a system in one graph. Merging the UML and structured Petri net modules, as proposed in Stanton (Stanton, 1999), produces graphical models which take full advantage of current OO software engineering techniques. The models are also mathematically provable (Delatour and Paludetto, 1998) and allow the modelling of concurrent and non-deterministic systems (Zapf and Heinzl, 2000). One of the main drawbacks of Petri net graphs is their inherent complexity, even on relatively simple systems. In the UML, operations are used to access and alter the internal state of the object. The proposed technique uses Petri nets to model these operations and their resultant behaviour changes. By modelling only the limited range of states and operations within a single object the complexity of the graphs is reduced considerably. As well as capturing the static, dynamic and behavioural attributes of the system, the resultant models help in the identification of user requirements, are understandable to a wider range of users, are extendable and reusable, and provide enough low level detail for the automatic generation of control code{STANTON 1999 #97}.

3 Applying Constraints

Once the classes have been designed and their operations and attributes established and modelled, the resultant object is highly generic and can be applied to a range of applications. However, in order to utilise the object, strict control must be placed over the actions it is allowed to perform. For example, a manipulator may be able to move left and right, up and down, back and forth and the gripper may open and close. When applied to a specific system the manipulator may not be able to move right due to an impeding obstacle and therefore the controller must always raise the object, move it right and lower it, in order for it to achieve the required action of moving right. If this constraint is built into the object then it becomes system specific and loses some of its genericity. This paper proposes a method of applying a constraint object to the class in order to meet the system requirements whilst not affecting the genericity of the class itself.

4 Related Work

4.1 Object Oriented Design

(Adiga and Gadre, 1990)), (Adiga, 1993) proposed OO modelling as a method of designing manufacturing systems and expanded the idea to take account of the increasing use of robots (Lin *et al*, 1994). Much of the early work was based around the methods proposed by (Coad and Yourdon, 1991), (Yourdon, 1994). Booch, Jacobson and Rumbaugh amalgamated the early ideas (Booch *et al*, 1999), (Jacobson *et al*, 1999) into the UML, and current work by the authors (Llewellyn *et al*, 2000) has applied the UML to a manufacturing system.

4.2 Object Oriented Petri Nets

Petri Net theory has been a major research topic for some time and several attempts have been made to integrate PNO and OO techniques (Delatour and Paludetto, 1998), (Venkatesh and Zhou, 1998). Other researchers have extended PNO to incorporate OO concepts such as the Hierarchical Object Oriented Design (HOOD). The HOOD approach (Wu, 1995), (Di Giovanni, 1991) first proposed by the European Space

Agency was expanded by Chen and Lu (Chen and Lu, 1997) to incorporate Petri nets in their Petri-net and entity-relationship diagram based OO design method (PEBOOD). However, these approaches have led to extremely complex models where the link between Petri nets and OO systems design is at best tentative. In addition the techniques do not fully capture all the benefits of a true OO approach.

4.3 Constraints

The UML uses the Object Constraint Language (Warmer and Kleppe, 1999) in order to apply constraints to the model. However, these are little more than comments with no direct code conversion possible. The forbidden state problem is an is an area widely researched in Petri net theory and the work of Holloway and Krogh (HOLLOWAY and KROGH, 1990) in applying constraints to controlled marked graphs has been adapted to fit the Petri net/UML approached presented in this paper.

5 Application

The technique described in this paper is demonstrated by applying it to the raw materials station (RMS), part of the University of Wales College, Newport's computer integrated manufacturing (CIM) system. Initially the system views are captured from a user's perspective using use case scenarios. Next the classes in the system are identified and their attributes and operations captured. The attributes (or states) and operations are modelled using Petri net graphs, where one graph is used to model all operations for a particular class. Output places (Stanton, 1999) are used to represent message passing between objects. Finally the system constraints are identified and placed in a constraint class for each object.

5.1 An overview of the CIM system

The system shown in figure 1 is designed as an example of a CIM system. It is composed of a number of modules that interact in order to produce end products. The raw materials used by the system are a perspex block and a metal cylinder. The block and cylinder originate from the raw materials station and are placed into trays on a conveyor bel[t]

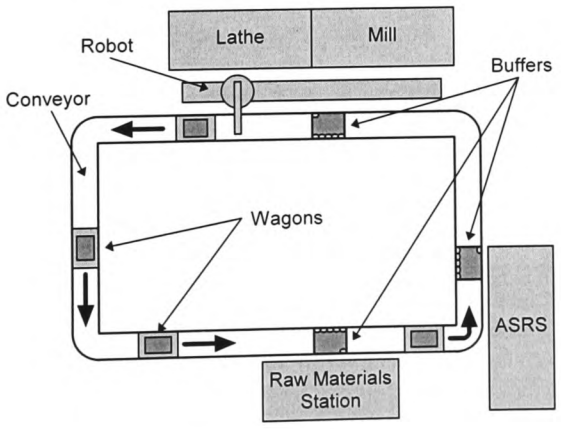


Figure 1 The CIM system

The block is milled and the cylinder lathed so that the two items fit together. Finally the finished product is stored in the automated storage and retrieval system (ASRS). The focus of this paper is on the RMS.

5.2 Use Case Scenarios

The RMS can be shown as a use case diagram, figure 2, and it can be seen that it interacts with the conveyor belt in order to perform its operations. These operations are - get a pallet from the conveyor belt (getPallet), put a pallet and block on the conveyor belt (putBlock) and put a pallet and cylinder on the conveyor belt (putCylinder).

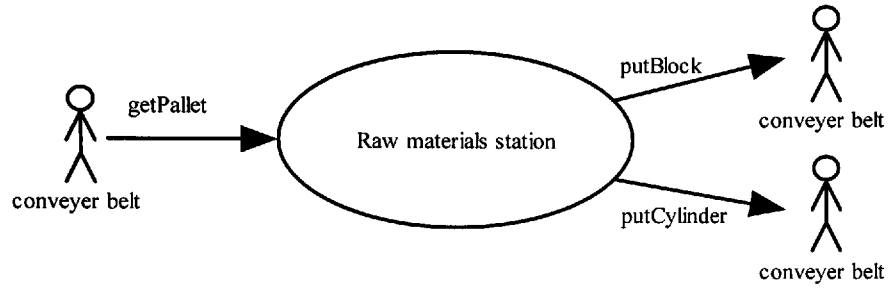


Figure 2 Use case scenarios for the raw materials station

The use case diagram in figure 2 identifies the communication between the module under consideration and the outside world and gives the basic information needed to operate the module. This can be described as the *interface* between this module and any component that needs to interact with it. This concept of *encapsulation*, also known as *information hiding*, is an important OO technique and helps achieve the idea of *loose coupling* (SOMMERVILLE, 1995), (Meyer, 1997), (PRESSMAN, 2000). The internal operations of the module are hidden from the user. In order to operate this module external users need only know about its interface, which describes the operations it performs. The internal details of how it provides a cylinder or block, or gets a pallet, are unimportant when calling these operations. Therefore, modifications made to the internals of an object should have a minimal, if any, effect on other objects in the system, as long as its interface remains unchanged. Expanding this idea gives the concept of a hardware/software object, where no distinction is drawn between the software and hardware in the module. Instead the module is thought of in terms of the operations it performs and the interface to those operations.

The RMS itself (figure 3) consists of a series of interacting objects. It contains two manipulators and two storage units. The latter contain blocks and cylinders respectively with one manipulator used to load cylinders onto a pallet waiting in the loading area, whilst the other serves the dual purpose of placing pallets onto the loading area, and populating pallets with blocks.

The whole station is controlled by programmable logic controllers (PLC's) via a series of pneumatic actuators. A description of the using the UML to create an aggregation of these actuators to form a manipulator is described in (Llewellyn *et al*, 2000).

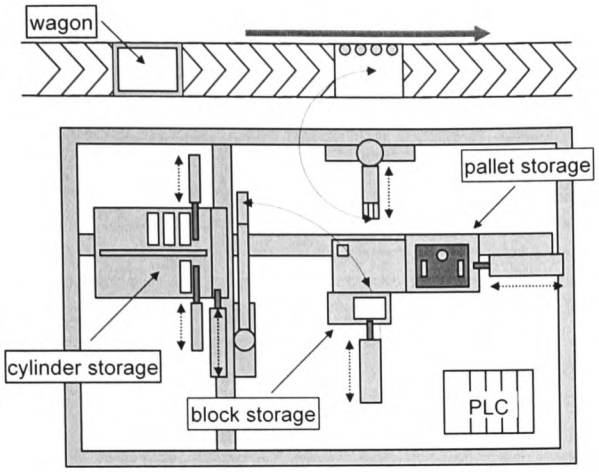


Figure 3 Layout of the raw materials station

The RMS can also be modelled with a use case diagram, to give an understanding of the behaviour of the module's interactions. The resultant diagrams are not shown for space considerations but give the following operations:

Object	**Action**
Cylinder Manipulator	
	Get cylinder from cylinder storage
	Put cylinder on loading area
Pallet Manipulator	
	Get pallet from conveyor
	Put pallet on loading area
	Get block from block storage
	Put block on loading area

Table 1 The operations of the raw materials station

These actions help in the design of the operations for each of the objects in the RMS, and further identify the interface of each object which can be used later to design the module controller. Using use case analysis is an iterative process where each module is decomposed into its component sub-modules and these in turn are modelled. Ultimately all levels of the system are modularised and enough detail is obtained to design each object fully using a bottom-up approach. Ultimately the top levels of the system are combinations of lower level objects.

5.3 Class diagrams

Focusing on the cylinder storage manipulator, it can be observed that the object is an *instance* of class manipulator, and that this class itself is a *composition* of four instances of class actuator. The actuator class has two simple methods that allow it to actuate or deactute. However, these actions carry out a different operation depending on the receiving object. For instance, an actuator in the system under consideration may take one of four types. It contains a rotary actuator, which is able to actuate right or deactuate left. It contains a horizontal actuator which is able to extend or retract, and a vertical actuator which is able to move up or down upon receiving its actuate or deactuate command. Finally it contains a gripper which when actuated opens and on deactuation closes. This demonstrates the OO concept of *polymorphism* whereby each of the classes responds differently to the same command based upon its hidden internal mechanisms. The manipulator class itself responds to commands such as move left, move right, up, down, open and close. These commands or operations form the interface to the manipulator class, with the individual actuators, and indeed their pneumatic valves and the PLC controller being encapsulated from the user.

5.4 Modelling the behavioural capabilities of an object using Petri net graphs

Having captured the class diagrams and any inheritance present in the system, it is possible to model the dynamic capabilities of the class. These are the operations that need to be invoked in order to make the class carry out its functions. In addition, the operations provide a method of altering the state or behaviour of the object. In a discrete event system (DES) such as the CIM system being considered, the state of the system at any moment in time can be captured by obtaining the states of all objects in that system. A Petri net graph allows these states to be represented visually or, if required, mathematically. The corresponding UML diagrams for capturing the dynamic and state aspects of a system are interaction and state diagrams. However, not only does this require the modelling of two separate diagrams, but neither are mathematically provable. The actuator class can be modelled using the following class/Petri net diagram:

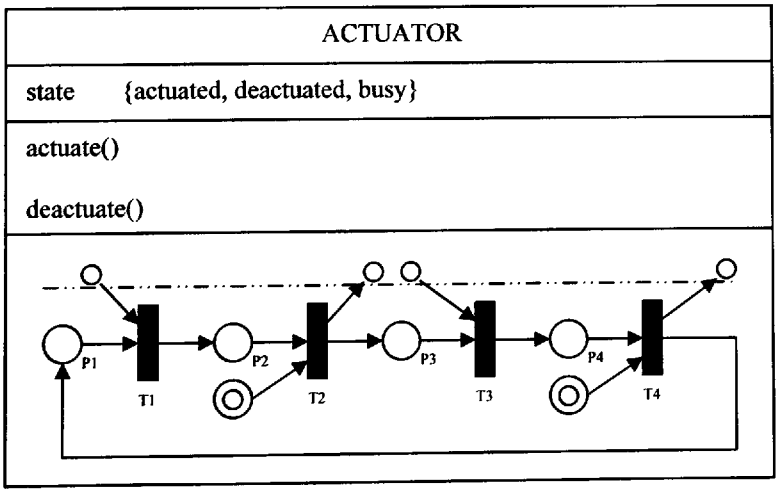


Figure 4 The complete actuator object

In the diagram smaller circles represent control places and feedback. The former are signals from the controller that invoke the method of the object. In this instance these can be either actuate or deactuate. The feedback is being sent to the controller object, with double circles representing input from external feedback sources. The dashed line represents the external (public) interface to the object.

5.5 Applying constraints to the object

The actuator class has been designed to be as generic as possible, as indeed is the resultant manipulator. It can be seen that this object can be reused in any application. To ensure the object remains as general purpose as possible the environment specific constraints are built into a separate object which acts as an intermediary between the controller, which is goal specific and the manipulator object itself. In the system under consideration, the only constraint for the raw materials manipulator is that the gripper cannot be opened when the arm is raised. Imagining the cylinders to be quite heavy, doing so could amount in considerable damage to the other objects in the system and possibly the cylinder itself.

Figure 5 shows a constrained object being used. The controller object sends a message to the manipulator via its constraint. The constraint validates the request based on the current state of the object it is constraining, and depending upon the outcome either sends the message on to the object for actioning or returns an error message to the controller.

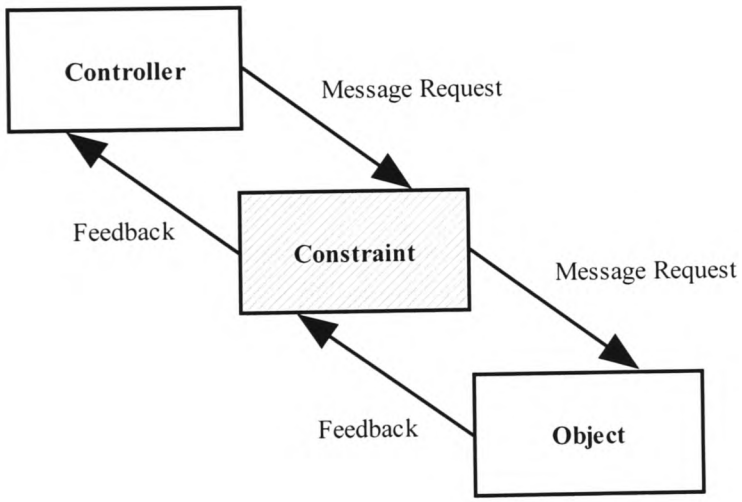


Figure 5 A constrained object

The constraint here is the intermediary between the controller and manipulator object, in other cases the constraint could be constraining a combination of objects where it is ensuring there are no conflicts between objects operating in the same environment.

6 Conclusions

It is widely accepted that manufacturing systems need to be flexible, customisable and maintainable. This is effectively addressed in the proposed OO system where individual objects can be customised and updated using the key features of UML, such as inheritance, polymorphism and encapsulation. By integrating the two types of models, the design of manufacturing systems is greatly enhanced. Manufacturing systems will be able to take advantage of the concepts of OO programming that have been widely available in software engineering for some time. Future upgrades to the resultant system will be more intuitive as manufacturing design adopts the 'plug and play' philosophy of other computer systems. The technique provides a model that can be used initially as a simulation tool and later as the basis for the automated generation of the control software. Once the initial design has been carried out many objects can be reused in future systems with no requirement for additional modelling.

7 Future Work

Whilst the idea of hardware software components (Kopetz, 1999), and Petri net modules (Stanton, 1999) has already been proposed, the communication between these modules needs translation into real systems. Petri net diagrams have been successfully converted into ladder logic (Stanton, 1999), however, as modern systems increasingly take advantage of more up to date programming languages, this idea needs to be extended to translate from Petri nets to their syntax. Thought also needs to be given to the elimination of the redundant code inherent in automated code generation from OO models (Narisawa *et al*, 1998). This paper proposes the concept of a constraint object based around the UML and Petri net graphs. Future work will expand this idea to cover a larger system that incorporates multiple objects and constraints.

Reference List

Adiga, S. (1993). *Object-oriented Software for Manufacturing Systems.* London, UK: Chapman & Hall.

Adiga, S. &Gadre, M. (1990). Object-Oriented Software Modeling of a Flexible Manufacturing System. *Journal of Intelligent and Robotic Systems*, **3**, pp. 147-165.

Booch, G., Rumbaugh, J. &Jacobson, I. (1999). *The Unified Modeling Language User Guide.* USA: Addison Wesley Longman..

Chen, K. &Lu, S. (1997). A Petri-net and entity-relationship diagram based object-oriented design method for manufacturing systems control. *International Journal of Computer Integrated Manufacturing*, **Vol. 10** (No. 1-4), pp. pp. 17-28.

Coad, P. &Yourdon, E. (1991). *Object-Oriented Analysis.* 2nd end. New Jersey, USA: Yourdon Press.

Delatour, J. &Paludetto, M. (1998). UML/PNO: A Way to Merge UML and Petri Net Objects for the Analysis of Real-Time Systems. *Lecture Notes in Computer Science*, **15** (43), pp. 511-514.

Di Giovanni, R. 1991. Hood Nets. *Lecture Notes in Computer Science*, (524), pp. 140-160.

Holloway, L. E. &Krogh, B. H. (1990). Synthesis of Feedback Control Logic for a Class of Controlled Petri Nets. *IEEE Transactions on Automatic Control*, **35** (5), pp. 514-523.

Jacobson, I., Booch, G. &Rumbaugh, J. (1999). *The Unified Software Development Process.* USA: Addison Wesley Longman.

Kopetz, H. (1999). *Do Current Technology Trends Enforce a Paradigm Shift in the Industrial Automation Market?* 7th IEEE International Conference on Emerging Technology and Factory Automation. pp. 1557-1565, Barcelona, Spain.

Lin, L., Wakabayashi, M. &Adiga, S. (1994). Object-oriented modelling and implementation of control software for a robotic flexible manufacturing cell. *Robotics & Computer-Integrated Manufacturing*, **11** (1), pp. 1-12.

Llewellyn, E. W., Stanton, M. J. &Roberts, G. N. (2000). *Towards the implementation of the Unified Modelling Language (UML) into a Computer Integrated Manufacturing (CIM) environment.* Fourteenth International Conference on Systems Engineering. pp. 398-403, Coventry, UK.

Meyer, B. (1997). *Object-Oriented Software Construction.* 2nd end. New Jersey, USA: Prentice Hall.

Narisawa, F., Naya, H. &Yokoyama, T. (1998). A Code Generator with Application-Oriented Size Optimization for Object-Oriented Embedded Control Software. *Lecture Notes in Computer Science*, **15** (43), pp. 511-514.

Pressman, R. S. (2000). *Software Engineering A Practitioners Approach.* 5th edn. London, UK: McGraw Hill.

Sommerville, I. (1995). *Software Engineering.* 5th end. London, UK: Addison-Wesley.

Venkatesh, K. &Zhou, M. (1998). Object-oriented design of FMS control software based on object modeling technique diagrams and Petri nets. *Journal of Manufacturing Systems*, **17** (2), pp. 118-136.

Warmer, J. &Kleppe, A. (1999). *The Object Constraint Language.* Reading, USA: Addison-Wesley.

Wu, B. (1995). Object-oriented systems analysis and definition of manufacturing operations. *International Journal of Production Resources*, **33** (4), pp. 955-974.

Yourdon, E. (1994). *Object-Oriented Systems Design.* New Jersey, USA: Prentice-Hall.

Zapf, M. &Heinzl, A. (2000). Approaches to integrate Petri nets and object-oriented concepts. *WIRTSCHAFTSINFORMATIK* , **42** (1), pp. 36-48.

A NINE STEP APPROACH TO DESIGNING SUCCESSFUL VISUAL PROGRAMMING APPLICATIONS

Eric Llewellyn, Martin Stanton and Geoff Roberts

Mechatronics Research Centre, University of Wales College, Newport
Allt-yr-yn Campus, PO Box 180, NP20 5XR, United Kingdom

Abstract: The following paper presents a nine-step method for overcoming the limitations of traditional models when designing visual applications. Microsoft Visual Basic is used to demonstrate the technique which can be applied to most visual programming languages. The method described takes account of factors such as an interface driven approach, the psychology of programmer commitment, the need to develop readable code, and provides a method of modularly designing detailed test documents. The method has proved to be suitable for existing visual basic developers and those wishing to move from procedural into visual programming.

Introduction

Visual programming is increasingly being adopted for software development and languages such as Visual Basic are becoming the tools of choice for applications development. This is inevitably due to their flexibility and ease of use, though their use is more usual within the software development rather than software-engineering domain. Command and control and real time systems will more likely run under languages such as C. For many applications there is an increasing need for traditional programmers to embrace this new technology and design quick, customised applications. It is inappropriate to approach the design of such systems using conventional design methodologies such as the waterfall model [2] {Sommerville 1995 #78}as these methods tend to be structured to procedural software development. Even experienced visual programmers tend to take an exploratory approach to rapid applications development (RAD) and this exploratory nature often leads to poorly designed programs with inefficient code. With exploratory programming, debugging is an ongoing affair as errors often result from the confusion caused by this approach.

Programmers often take a cursory glance at the requirements and immediately begin developing their applications with no real picture of the overall project. This means that problems are solved as they are encountered which very often leads to the creation of other problems further along the development process. The code is reworked from front to back with problems being fixed along the way, and this method is iteratively carried out until the application works satisfactorily. This can lead to frustration and dissatisfaction on the part of the programmer. In many instances the initial good intentions of the programmer to apply interesting techniques and routines to the code is destroyed by the final rush to get the application working. For all forms of programming the foundations of the application are code commenting, indenting, and naming of variables and objects. If these foundations are not correctly laid out from the outset of the project several problems can arise. Amending procedure names etc. can cause errors in the application. The programmer may have lost the inclination to do any more than is absolutely necessary towards the end of the project life cycle. In visual programming, often too much time at the outset is spent designing the interface, which is often important coding time wasted.

The following nine-step plan is suggested as a method of overcoming many of the problems identified:

Step 1 – Plan the application

Step 2 – Design a working interface

Step 3 – Assign meaningful names

Step 4 – Identify events and describe the required behaviour for each event

Step 5 – Code the easy statements

Step 6 – Formulate complex code

Step 7 – Implement the complex code

Step 8 – Test and debug the application

Step 9 – Enhance the GUI

Motivation

Programmer motivation during a software development project can be likened to a musician playing a guitar. For example, the musician strikes chord C. The note follows a path as shown in Figure 1 below:

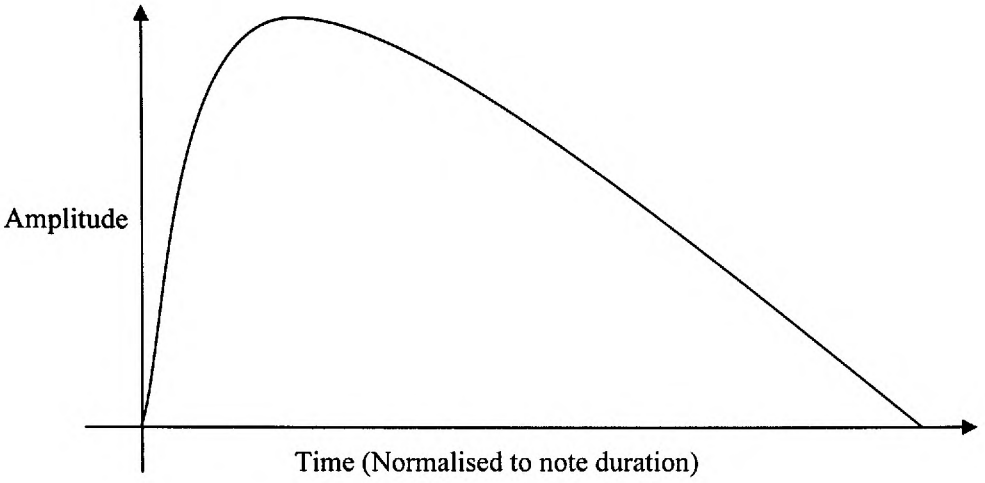


Figure 1: The rise and decay of a guitar chord

The note rises quickly to its peak and then slowly begins to fade as time progresses. The same is true for programmer motivation. It is possible to use the same graph with a different label on the y-axis to represent this.

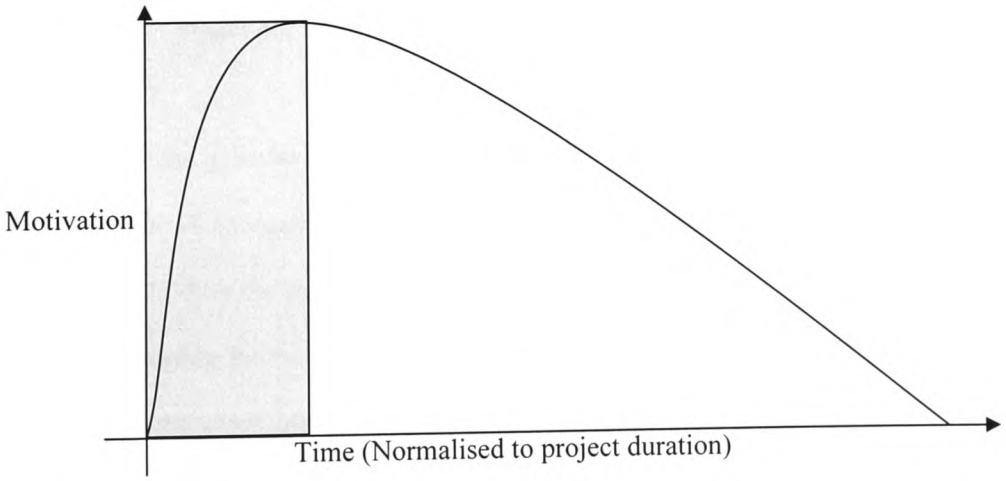


Figure 2: The rise and fall of programmer motivation

The programmer's motivation is at its highest level at the inception of the project, shown as a shaded area in Figure 2, as the project progresses and the tasks become tedious and repetitive, the motivation of the programmer fades, although there is often a slight upturn as the project reaches completion.

If this concept is taken a stage further it can be seen in Figure 3 that this cycle is actually composed of smaller, similar cycles for each task in the development of the project. Each cycle will have higher or lower motivation depending on how the individual programmer likes that particular task.

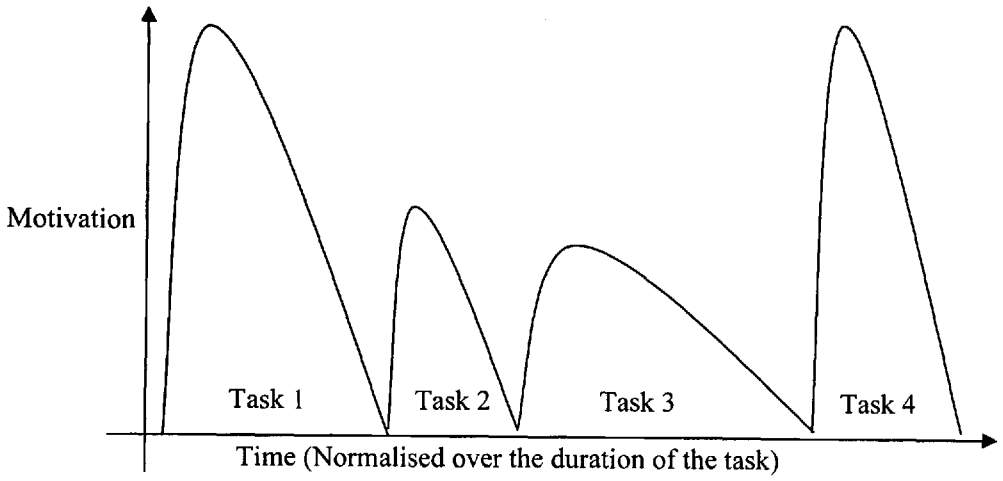


Figure 3: Programmer motivation over the duration of a project

To return to the guitarist analogy, the chord previously struck is about to lapse and yet the musician wishes to extend the note. He/she does not want to strike the note again, as this would alter the musical score. Instead the guitarist has a range of techniques such as adjusting the tremolo arm or vibrating their arm which can extend the note. Likewise if the programmer begins to tire of a task, he or she does not want to begin the task again. Instead a method is needed which takes the task in another direction. This change of direction is hopefully enough to re-motivate the programmer to

complete the current task and begin the new one. The methodology presented in this paper aims to counteract this 'waning of programmer' interest by placing the more interesting elements of development at key stages within the process. The suggested steps will aid in the generation of efficient and robust code.

Step 1 – Plan the application

Action: Understand what is required and plan the program

It seems obvious but it is important to understand the whole application problem before embarking upon any development task. In many cases a developer may skirt through the detail of a problem and begin programming in an almost top down approach. Imagine the situation where a programmer is given a lengthy brief about a change handling system that is required by a client in an amusement arcade. Having quickly glanced over the specification the program begins coding in the 'code and fix' fashion and arrives at a fully functional application. The system can accept any sterling paper denomination and can duly give a various combination of change as required by the user. The developer is pleased with his/her effort and subsequently demonstrates the application to the client. However, the client quickly draws the programmer's attention to some of the small print in the specification, the system must be able to deal with different currencies as it is to be implemented throughout their international gaming business. Whilst the code can be modified to accommodate this it may require a considerable amount of time to do so. Time which could have been spent on other tasks had the problem been thought through in detail.

At this step any available documentation should be read and understood. Interviews should be held with the client to establish anything not stated and to clear up any queries. Despite the *ad-hoc* nature of event driven programming, there will be some basic path which the application follows and this can be drawn as in the generic step

diagram shown in Figure 4 which shows the basic route through the application. The dotted lines represent possible conditions or repetitions in the program

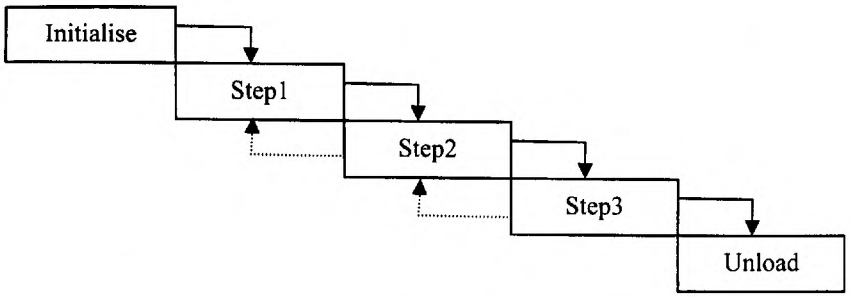


Figure 4: The generalised path of event driven software development

As applications such as Visual Basic are inherently interface oriented it should be possible to identify the type of controls, i.e. buttons, check boxes and so on, and possibly events related to these controls, i.e. mouseOver, click etc.

Step 2 – Design a Working Interface

Action: Develop a working interface in order to begin implementing the plan

Visual languages are by their very nature, and as the name suggests, based around interfaces. The effective design of such graphical user interfaces (GUI) plays a strong role in the usability of the finished product. Therefore much thought should be placed on effective design. There are many articles available which discuss the design of GUI's and how computers and humans interact ([3], [1]), however, this is beyond the scope of this paper. GUI design can be an interesting and satisfying part of the project life-cycle and many developers enjoy this aspect of the project, however, at the initial phase of the design cycle there is the danger of spending too much time on the GUI design, especially in a time constrained project where this could be at the cost of the functionality of the program.

A rough or functional interface should be developed for all forms required by the application and these should be populated with the various controls required. The planning step should have suggested some of the controls and a reasonable idea of the form hierarchy which can be developed at this step. This step can be carried out, where necessary, with the direct involvement of the client. It is not important here to adjust fonts, screen colours and the aesthetics of the forms.

Having considered the application in its entirety at step1, even the novice visual programmer will find it easy to ascertain the control types to use drawing on their experience with other Window's applications.

Step 3 – Assign meaningful names

Action: Assign meaningful names to all objects and variables in the system

To demonstrate the benefit of correct naming compare the code snippets shown below:

```
Public sub command1_click()
      a=(b/2)*c
      label1.caption=str(a)
End Sub
```

```
Public sub calcArea_click()
   strArea=(sngBase/2)*sngHeight
   lblArea.caption=str(sngArea)
End Sub
```

Whilst it can be seen that the leftmost example is halving a variable and multiplying by another, assigning the value to a third variable, it becomes apparent from the rightmost code that it is the area which is being calculated. By simply giving code, objects and events sensible names code becomes easier to decipher for anyone examining the work at a later step or indeed for the developer. Adding the *sng* to the variable names gives the reader and indication they are type single. Microsoft provides a valuable list of naming conventions, the most common of which are shown in the Table 1 below:

Data type	Prefix	Example
Boolean	bln	blnFound
Byte	byt	bytRasterData
Collection object	col	colWidgets
Currency	cur	curRevenue
Date (Time)	dtm	dtmStart
Double	dbl	dblTolerance
Error	err	errOrderNum
Integer	int	intQuantity
Long	lng	lngDistance
Object	obj	objCurrent
Single	sng	sngAverage
String	str	strFName
User-defined type	udt	udtEmployee
Variant	vnt	vntCheckSum

Table 1: Common variable names based on the Microsoft standard

This process can be carried out in parallel with step 2, it is important that all objects and variables have sensible, meaningful names at the end of this step.

Step 4 – Identify events and add comments

Action: Identify all events in the application and describe the required behaviour for each event as appropriate

Having designed a working interface for the application and with the plan from step 1, the developer at this step will have some concept of the program's overall flow. Tying the two together the events which trigger these paths become apparent i.e., buttons, mouse events etc are the user inputs which can cause actions. Likewise some actions, such as timers, that are under the control of the program itself can also invoke events.

These events should be commented with sensible statements, which can later provide guidelines for implementing in the language syntax. Where the code is simple such as "end" application the comments may be simple. Where the code is complex, the

comments can be broken down further to provide more guidance when coding. For example, an end button may invoke a message box warning containing the option to exit (Yes) or continue (No). This could be commented as follows:

```
Public Sub cmdEnd_Click()
        'User has clicked the end button
        'Display message box
        'End application if user clicks yes
End Sub
```

The first comment is a general statement as to the purpose of the subsequent code, whilst the next statements break down the task for coding. At the end of this step the comments will describe 'pseudocode' statements detailing the function of each event.

Step 5 – Code the easy statements

Action: Write the code for the easier parts of the application

Sections of code may be quite simple to implement, such as 'End' and 'Clear' buttons. However, there are different degrees to which this is completed effectively. For example, an end button can contain no more than an 'End' statement at which, by accident or choice, the user is abruptly taken from the application. A more suitable end statement displays a warning offering the user a further choice. These refinements are generally simple to implement and add much to the user friendliness of applications. At the early stage of the development when the programmer is still motivated and keen to complete the application, these should be coded. Towards the end of the life cycle the programmer motivation and/or time may preclude these statements being effectively coded.

Write all essential code so that the application has basic functionality. Where the application has multiple forms this can include opening and closing the relevant screens and probably includes a working end button.

Step 6 – Formulate complex code

Action: Outline any complex code or calculations outside of the Integrated Development Environment (IDE)

Almost inevitably the application will be required to perform some calculations or complex data handling. The comments from step 3 will have aided in the identification of the necessary steps required, which should now be outside of the IDE. Accompanying the calculations should be, where applicable, test data which reinforces the formula. By working outside of the programming environment the developer is able to pursue the most effective method of solving a problem and does not try to impose 'language terminology' on the solution. At this step he/she is free from the constraints of the programming language. It is suggested that once an effective solution to a problem has been found, the problem is to establish the best method of implementing this in the chosen language and not the other way around. The resulting test data also becomes useful in step 9 of the design.

Step 7 – Implement the complex code

Action: Code the complexities of the application

Once the complex code has been described and manually tested, a viable solution needs to be found to carry out the required tasks. Using the commentary provided from step 4 and the plan developed in step 1, this task should be more intuitive. It is however the most crucial step of development as the main functionality of the application is within this step. The main requirement of the application and the

purpose for its design will be coded during this step. It is also the point at which the programmer's motivation is usually beginning to degrade as he/she tires of the project.

Step 8 – Test and debug the application

Action: Design a test requirements document and test the application

The first priority at this step is to design a test requirements document (TRD) which is used to test the performance of the application. The document should identify each event and the associated action(s) and ensure they perform correctly. For many programmers TRD design is one of the least interesting and tedious parts of the development process and is generally disliked. However, the approach suggested here lends itself well to TRD design. At step 4 all events were identified and commented such as that shown in Figure 5. These events and comments map neatly into the TRD as shown below:

Event	Action	Check
End button clicked	Warning displayed	√
	User clicks no program continues	√
	User clicks yes program ends	√

Figure 5: An event/action test document

Taking this a stage further the TRD should also concern itself with accuracy as well as functionality. For example, an application may be required which allows the user to enter two numbers, which are added together and displayed upon pressing a button. The test for the button may be as shown in Figure 6:

Event	Action	Check
Button clicked	Result is calculated and displayed	√

Figure 6: A test documents showing a calculation event

Therefore, if the numbers 10 and 20 are entered and 30 is displayed it would be correct to check off this test as being complete. However, if the same numbers are entered and the number 40 is displayed is the test satisfied? In this case yes, a result was calculated and displayed. There was no stipulation that the result had to be correctly calculated. It is suggested that to overcome this inadequacy the wording of the test is carefully prepared and that all calculations are validated using test data. Conveniently step 8 will provide tests and answers which can now be used with the application and compared to ensure the program functions correctly. The probability is that the testing of the application will highlight any bugs in the system or inadequacies with the operation of the program. At this step these can be amended.

Step 9 – Enhance the Graphical User Interface (GUI)

Action: Enhance the GUI

By step 9 the application is well coded and commented, fully functional and thoroughly tested and the programmer is possibly eagerly awaiting the signing off of the project. After spending a large amount of time dealing with the intricacies of the code, the programmer is now free to spend as much or as little time as necessary in the pursuit of enhancing the interface.

Conclusions

This article has presented a nine-step plan to a successful visual software development. The technique will eliminate many of the problems inherent in the development of these types of applications. One of the important aspects of the method is that it takes into account the initial enthusiasm when a new project is undertaken and the wane in interest as the project progresses. Whilst much work has been carried out in the field of user psychology, this paper approaches the design from the perspective of programmer psychology. It identifies the aspects of projects that are generally poorly implemented such as 'easy code' and TRDs and offsets the unwillingness to do this by placing them in the enthusiastic period of development. The more interesting aspect of RAD programming, i.e. GUI design is placed at the end of the project when it provides a resurgence of interest before the project is finally completed. Programmers moving into the visual programming field from more procedural languages now have a structured framework in which to develop efficient and robust applications. Whilst the method outlined in this paper has been applied to Microsoft's Visual Basic, the technique can readily be applied any visual application such as Borland's Delphi or C builder. The method presented is equally applicable for students and industrialists and provides a driving force that addresses the fundamentals of visual software development.

Further Reading

[1] Schneiderman, B., Designing the user interface : strategies for effective human-computer interaction. 3rd edn. 1998. Addison-Wesley. London.

[2] Sommerville, I., Software Engineering. 6th edn. 2001. Addison-Wesley. London.

[3] Wood, L. E., User interface design : bridging the gap from user requirements to design. 1998. CRC. Boston.

A COMBINED OBJECT-ORIENTED AND STRUCTURED PETRI NET APPROACH FOR DISCRETE EVENT SYSTEM'S DESIGN

Eric Llewellyn[†], Dr Martin Stanton[‡] and Professor Geoff Roberts[†]

[†] University of Wales College, Newport. School of Computing and Engineering
Allt-yr-yn Campus. PO Box 180, Newport NP20 5XR, South Wales, United Kingdom.
email: eric.llewellyn@newport.ac.uk and geoff.roberts@newport.ac.uk

[‡] Manchester Metropolitan University, Department of Computing and Mathematics
John Dalton Building, Chester Street, Manchester, M1 5GD, England, United Kingdom.
email: m.stanton@mmu.ac.uk

Abstract

This paper proposes a method of designing discrete event systems utilising a combined object-oriented and Petri net approach. The approach allows manufacturing system designers to develop truly generic, and therefore reusable systems and components that can aid in the speed up of system design and implementation. It also provides a user-centric view of the system that can facilitate effective communication between system designers and end users.

Initially use-cases are iteratively developed until the resultant diagrams fully capture user requirements along with a suitable level of detail in order to implement the design. Subsequently the use-case scenarios provide a series of 'test cases' that enable the end system to be fully tested against the original design requirements. A series of class diagrams are then produced using the standardised notation provided by the Unified Modelling Language (UML). The resultant class diagrams provide a generic and abstracted view of the system and enable the system designer to identify all levels of modularity within the system under consideration. Careful identification of the 'interface' to each module in the system presents two major benefits to manufacturing system design: firstly it allows a large project to be concurrently developed by a team thereby reducing the time to implementation; secondly it enables manufacturing organisations to incrementally implement new systems by department or even cell level. A well-defined interface and encapsulated module mean that it is possible to combine new and existing technologies without leaving 'islands of automation'.

The approach outlined in this paper draws no distinction between hardware and software, and instead views the system as a series of events and resultant state changes that are modelled using structured Petri nets. Structured Petri nets allow a model to more closely resemble the system under consideration by extending the basic Petri net graph to include input and output places that can be used to model direct control of a system and allow for the capture of feedback. In common with the basic Petri net model the structured Petri net graphs can be used to simulate a system, enabling the system modeller to carry out 'what if' analysis on any proposed design or change. The structured Petri net approach also allows for the generation of control code, which again reduces the all-important time to implementation.