A METHODOLOGY FOR ANALYSIS AND CONTROL OF DISCRETE EVENT DYNAMIC SYSTEMS

DANIEL AZZOPARDI

Submitted for degree of Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

August 1996

The University of Aston in Birmingham

A METHODOLOGY FOR ANALYSIS AND CONTROL OF DISCRETE EVENT DYNAMIC SYSTEMS

by

Daniel Azzopardi

Submitted for the degree of

DOCTOR OF PHILOSOPHY

1996

SUMMARY

The rapid developments in computer technology have resulted in a widespread use of discrete event dynamic systems (DEDSs). This type of system is complex because it exhibits properties such as concurrency, conflict and non-determinism. It is therefore important to model and analyse such systems before implementation to ensure safe, deadlock free and optimal operation. This Thesis investigates current modelling techniques and describes Petri net theory in more detail. It reviews top down, bottom up and hybrid Petri net synthesis techniques that are used to model large systems and introduces an object oriented methodology to enable modelling of larger and more complex systems. Designs obtained by this methodology are modular, easy to understand and allow re-use of designs.

Control is the next logical step in the design process. This Thesis reviews recent developments in control of DEDSs and investigates the use of Petri nets in the design of supervisory controllers. The scheduling of exclusive use resources is investigated and an efficient Petri net based scheduling algorithm is designed and a re-configurable controller is proposed.

To enable the analysis and control of large and complex DEDSs, an object oriented C++ software tool kit was developed and used to implement a Petri net analysis tool, Petri net scheduling and control algorithms. Finally, the methodology was applied to two industrial DEDSs: a prototype can sorting machine developed by Eurotherm Controls Ltd., and a semiconductor testing plant belonging to SGS Thomson Microelectronics Ltd.

Key words: Discrete event dynamic systems (DEDS), Object oriented methodologies, OMT, Petri nets, scheduling, supervisory control.

This Thesis is dedicated to my parents, Jane and Tony, to my sister Judith and her family and to my brother Matthew and his family, for their support and encouragement

ACKNOWLEDGEMENTS

I would like to acknowledge the help given to me by the following:

Dr. D.J. Holding for supervising my research work, guiding the development of this Thesis and for the invaluable discussions we had throughout my stay at Aston University.

Dr. G.F. Carpenter and Dr. J.S. Sagoo for reading my Thesis and making suggestions for improvements in my research work.

Dr. A.P. Wilton for his help with the Sun workstations used for the software development related to this research work and for his suggestions on the presentation of experimental results.

Dr. J. Jiang of Eurotherm Controls Ltd. for allowing me to use C++ classes that he developed and for supplying information regarding the prototype can sorting machine developed by Eurotherm.

Mr. G. Genovese of SGS-Thomson Microelectronics (Malta) Ltd., for the information and discussions regarding the micros semiconductor testing plant described in this Thesis.

The EPSRC for sponsoring this research work and my research fellowship at Aston University (Grant No. GR/J09352)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction to this Thesis

This Thesis develops a methodology[1] for the design, analysis and optimal control of discrete event dynamic systems (DEDSs). A DEDS is a dynamic system that evolves with abrupt changes, at possibly unknown and irregular intervals. DEDSs are encountered in many fields of engineering, including manufacturing, robotics, traffic management, logistics and computer and communication networks [Sobh *et al.* 94]. This Chapter introduces the basic theory of DEDSs, describes the purpose of the research and identifies the contributions made in this Thesis.

1.2 Discrete Event Dynamic Systems

The IEEE Standard Dictionary of Electrical and Electronic Terms defines a system as "a combination of components that act together to perform a function not possible with any of the individual parts". Any type of system has a set of inputs, $\mathbf{u}$, a set of outputs, $\mathbf{y}$, and a state, measurable over a period of time $[t_0, t_f]$, where $t_0, t_f \in \Re^+$. The state of a system is defined as the smallest set of variables (state variables) such that knowledge of these variables at time, $t = t_0$, together with the knowledge of the system inputs, $\mathbf{u}(t)\ \forall t \geq t_0$, completely determines the behaviour of the system $\forall t \geq t_0$ [Ogata 90]. The set of equations required to specify the state $\mathbf{x}(t)\ \forall t \geq t_0$, given $\mathbf{x}(t_0)$ and the function $\mathbf{u}(t)$, $\forall t \geq t_0$, are called the **state equations** and the **state space** of a system, $\mathbf{X}$, is the set of all possible values that the state may take.

A DEDS belongs to a class of systems that is described by means of the system classification illustrated in Fig. 1.1. Referring to this classification, a static system is one in which the output, $\mathbf{y}(t)$, is independent of past values of the input, $\mathbf{u}(t_1)$, $t_0 < t_1 < t, \forall t \in \Re^+$ whereas a dynamic system is one in which the output depends on past values of the input.

[1] A design methodology is a process for the organised production of a design using a collection of predefined techniques and notations [Rumbaugh *et al.* 91]

Furthermore, dynamic systems can be classified as **time-varying** or **time-invariant** systems. For time-varying systems, the input/output relationship of a system, **g**, depends on time, hence, $\mathbf{y} = \mathbf{g}(\mathbf{u}(t), t)$. In linear time-invariant systems, the function **g**, is linear, otherwise the system is non-linear.

If the state space of a non-linear system is described by a discrete set and state changes (events) are only observed at discrete points in time, the system is known as a **discrete state** system. An event can be thought of as occurring instantaneously and causing transitions from one discrete state value to another.

Discrete state systems are further subdivided into **time-driven** and **event-driven** systems. In time-driven systems state transitions are synchronised to a clock, whereas in event-driven systems it is the occurrence of asynchronously generated discrete events that forces instantaneous state transitions. In between event occurrences the state remains constant.

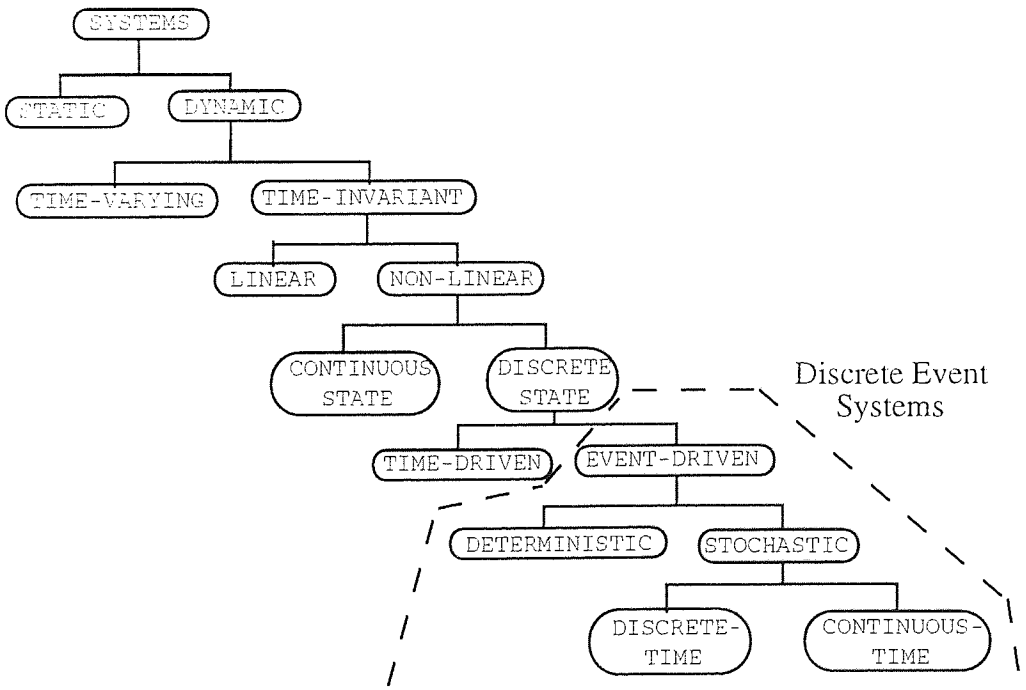


Fig. 1.1 Systems classification [Cassandras 93]

Based on the system classification above (Fig. 1.1), a **DES** is a *discrete-state, event-driven* system [Cassandras 93]. Furthermore, a **DEDS** is a DES whose output depends on past values of the input. DEDSs are complex systems because they exhibit characteristics such as concurrency, conflicts, non-determinism and system deadlocks. Current theories and techniques for modelling and analysis of DEDSs are discussed in Chapter 2 of this Thesis.

Definition 1.1

Mathematically, a DEDS is defined by Ramadge and Wonham [89] as the state automaton [Hopcroft and Ullman 79] described by the five-tuple: $\mathbf{G} = (\mathbf{E}, \mathbf{X}, \Gamma, \mathbf{f}, \mathbf{x}_0)$, where,

$\mathbf{E}$ is a countable set of events

$\mathbf{X}$ is a countable state space

$\Gamma(x)$ is the set of feasible or enabled events, $\forall x \in \mathbf{X}, \Gamma(x) \subseteq \mathbf{E}$

$\mathbf{f}(\mathbf{x}, \mathbf{e})$ is a state transition function, $\mathbf{f}: \mathbf{X} \times \mathbf{E} \rightarrow \mathbf{X}, \forall \mathbf{e} \in \Gamma(\mathbf{x}), \mathbf{x} \in \mathbf{X}$

$\mathbf{x}_0 \in \mathbf{X}$, is an initial state

$\mathbf{G}$ is interpreted as a device that starts in the state $\mathbf{x}_0 \in \mathbf{X}$, and executes state transitions, generating a sequence of events. Events are considered to occur spontaneously and instantaneously.

1.3 Modelling and analysis of DEDS

A model of a DEDS is a representation of the features of a system that are considered to be important for its correct operation. Obtaining a model of a system is the first step towards understanding how the system works and, if the model is an accurate representation of the system, it can be utilised to analyse the system's dynamic behaviour. The mathematics of traditional control theory [Healey 75, Ogata 90], including differential and difference equations, has been developed over the centuries to model and analyse continuous processes that are observed in nature. However, the rapid advances in computer technology have resulted in a widespread use of DEDSs that are mostly man-made and are overwhelmingly complex [Cassandras 93]. The increasing complexity of man-made systems makes intuitive solutions inadequate [Ramadge and Wonham 89], therefore it is necessary to use some form of state transition structure (for example, automata [Hopcroft and Ullman 79] or Petri nets [Peterson 81]), or a set of algebraic equations [Hoare 85] or a logical calculus such as temporal logic [Manna and Pnueli 83] for their analysis and design. According to Ramadge and Wonham [89], no single modelling approach will suffice for all problems of interest and every approach has its own applications, virtues and limitations. A large capital is involved in designing and operating DEDSs and some of these systems are safety critical [Leveson 86], so it is important to model and analyse new systems, prior to their implementation, to ensure safe and optimal operation. According to Cassandras [93]:

"New models and methodologies are needed not only to enhance design procedures, but also to prevent failures (which can indeed be catastrophic at this level of complexity) and to deliver the full potential of these systems."

Since this Thesis is about control of DEDSs, including safety critical applications, Chapter 2 reviews current modelling techniques to select one that is well proven, allows automation of analysis for convenience and provides a compact graphical representation of the model to facilitate visualisation of the problems.

1.4 Synthesis of a DEDS model

Once the modelling technique that is best suited to the problem is selected, the issue of "how to build a system that behaves as we desire" needs to be addressed [Cassandras 93]. Design techniques have been developed to aid the designer to synthesise models of complex systems. These include top down, bottom up and object oriented design.

Top down design [Wirth 71], also known as stepwise refinement, is a technique for the design of a system by moving from a general statement of what the system does to detailed statements about specific tasks that are performed. This approach is often referred to as a "divide and conquer" approach because each statement is decomposed into more specific statements in a step by step fashion. Sometimes the top down approach can be so abstract that it is hard to find a starting point [McConnell 93]. Therefore, an alternative approach is bottom up design [Agerwala and Choed-Amphai 78]. In this approach one needs to identify the low level capabilities that the system needs to have, then to identify the common aspects of low level components and group them. This process is then repeated on the next higher level.

The difference between the two approaches is that the top down decomposition works from the general problem and breaks it into smaller more manageable problems, whereas the bottom up approach starts with smaller more manageable problems and works towards a general solution. Both approaches have their strengths and weaknesses. The strength of top down design is that people find it easy to break up a big problem into smaller components and a weakness is that the top function of a system might be difficult to identify [McConnell 93]. Another disadvantage is that many systems are not naturally hierarchical, so they are difficult to decompose. The most serious weakness is that top down functional design requires a system to be described by a single function at the top, which is a dubious requirement for many modern event-driven systems [McConnell 93].

One of the strengths of bottom up composition is that it typically results in early identification of implementation details. However, it is difficult to use exclusively since most people are better at breaking down a concept into smaller concepts than they are at taking small concepts and making one big one [McConnell 93]. Another disadvantage of bottom up design is that it is not possible to build a system using building blocks without actually knowing what the final product will look like, therefore it can only be used in conjunction with top down design.

Both bottom up and top down approaches concentrate on functional abstraction, and have produced incomplete specifications and designs for complex systems [Firesmith 93]. In order to facilitate the design of complex systems, produce more understandable designs and specifications, facilitate the transition between design and implementation and enable software re-use, several researchers including Seidewitz [89], Coad and Yourdon [90] and Firesmith [93], have advocated a paradigm shift towards object oriented (OO) techniques.

In OO techniques [Booch 94] one refers to classes of objects. A class of objects is defined as a collection of attributes and operations that are able to manipulate the values of these attributes. An object is an instance of a class and the values of the object's attributes can only be changed by executing the operations defined in the class it belongs to. Object oriented design (OOD) is the process of identifying real-world objects and classes of objects, identifying the operations on the classes and objects, and then building a system from those objects [McConnell 93]. The encapsulation of properties and operations within an object makes it possible to treat the object as a 'black box'. This is in contrast to traditional functional design techniques where data structures and functions are defined separately and are only loosely connected. Encapsulation prevents small changes to an object from having a large ripple effect on the whole system. One can define a new class, starting off from another class, by means of inheritance. The inherited class would contain all the attributes and operations of the super class but would have some additional attributes and operations. Therefore inheritance makes it possible to re-use previously designed classes. The application of top down, bottom up and object oriented methods to synthesise DEDS models is investigated in Chapter 3 of this Thesis.

1.5 Control of DEDSs

Control is the next logical step to the design process [Cassandras 93]. The adaptation of traditional control theory to DEDSs was only recently pioneered by Ramadge and Wonham [87]. Their strategy was to model the DEDS on a state automaton [Hopcroft and Ullman 79] and completely describe its behaviour by the language [Hopcroft and Ullman 79] generated by the automaton. System requirements and specifications were also assumed to be specified as languages.

To control a DEDS [Defn. 1.1], it is assumed that certain events of the system can be disabled (not allowed to occur) when desired [Ramadge and Wonham 89]. Thus, the set of events, $\mathbf{E}$, is partitioned into the set of controllable events, $\mathbf{E}_c$ and the set of uncontrollable events, $\mathbf{E}_u$, where $\mathbf{E} = \mathbf{E}_c \cup \mathbf{E}_u$. Uncontrollable events occur spontaneously. However, the occurrence of controllable events depends on the enabling/disabling action of a system controller defined by the function $\gamma(w)$, where w is the sequence of events that have been observed up to the current state. In automata theory [Hopcroft and Ullman 79] a sequence of events is referred to as a string.

A controller of this type is known as a supervisor [Ramadge and Wonham 89], the fundamental purpose of which is to provide closed loop control to force the system to behave as specified under a variety of operating conditions [Cassandras 93]. The supervisor (Fig. 1.2) converts the input string, w, into a controlled string, w_c. Feedback is provided by observing the state sequence resulting from w_c either by directly observing every new state or by observing the output, $y = g(x,e)$.

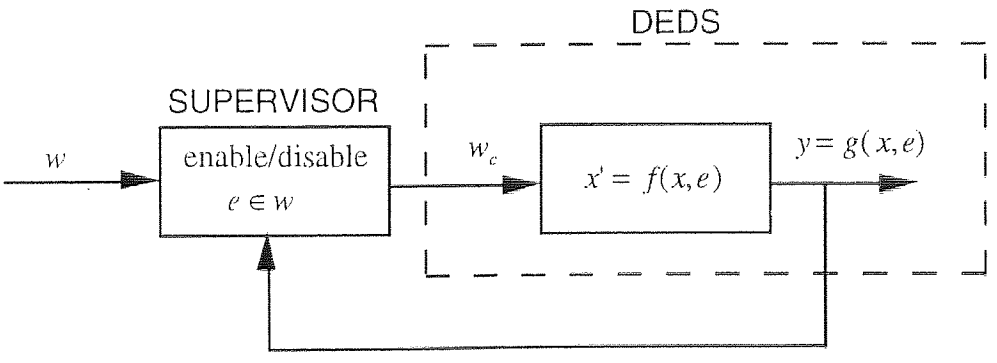


Fig. 1.2 Supervisory Control of DEDSs [Cassandras 93]

This method for supervisory control, as described in [Ramadge and Wonham 87], is limited by the use of a state automaton (to represent the system) which is adversely affected by the state-explosion phenomenon [Zhou and DiCesare 93]. The feasibility of constructing supervisors for DEDSs and methodologies for supervisor design are still under development [Cassandras 93] and the basic theory of supervisory control, its limitations and a novel supervisory controller design are presented in Chapter 4 of this Thesis.

1.6 Optimisation of the system

Exclusive-use shared resources are components that are present in most DEDSs. Examples of such resources are: production units in process plants, machines in flexible manufacturing systems (FMSs); processors, communication channels and storage devices in computer systems; cash machines in a bank; and so on. Their common feature is that they are shared by different users or processes, but can only be used by one user or process at a time. One of the problems raised by the presence of shared resources in a DEDS is that the overall performance of the system depends on the order in which they are allocated to the users or processes. It is necessary to schedule the allocation of resources if one is to ensure the optimal control of a DEDS. In Chapter 4 traditional approaches to scheduling resources are introduced, followed by the presentation of an efficient scheduling algorithm to ensure optimal control.

1.7 Contributions of this Thesis

This Thesis presents an object oriented methodology for the design, analysis and optimal control of DEDSs based on Object Modelling Technique (OMT) [Rumbaugh *et al.* 91], Petri net theory [Murata 89] and supervisory control theory [Ramadge and Wonham 87]. Four main contributions are presented:

(i) An object oriented methodology for synthesis of DEDS models [1.7.1]
(ii) A fast Petri net based scheduling algorithm [1.7.3]
(iii) A re-configurable supervisory controller for DEDSs [1.7.2]
(iv) A Petri net software tool kit [1.7.4]

These contributions are assessed in Chapter 6 by considering their application to two industrial DEDSs.

1.7.1 An object oriented methodology for modelling DEDSs

This Thesis presents an object oriented methodology for synthesis of DEDS models . The technique is a modification of OMT [Rumbaugh et. al 91] and complements the currently available Petri net synthesis techniques. The OMT model consists of three separate models; the object model, describing the objects in the system and their relationship; the dynamic model, describing the interactions among objects in the system; and the functional model, describing the data transformations of the system. This Thesis shows that the dynamic model can be represented by a controlled Petri net [Krogh 87], and that there is a direct link between the three OMT models, thus enabling construction of the complete model by following a step by step approach. This makes the dynamic model more understandable and the design engineer more confident that it accurately represents the behaviour of the system. This methodology has been applied to two classical problems in Chapter 3 and to two industrial DEDSs in Chapter 6. The methodology described in this thesis has resulted in the publication of a conference paper [Azzopardi *et al.* 96].

1.7.2 A fast Petri net based scheduling algorithm

This contribution is an improvement over the Petri net [Peterson 81] based scheduling algorithm published in [Azzopardi and Lloyd 94a]. The algorithm uses a branch and bound algorithm applied to the timed Petri net [Sifakis 78] model of the plant. However, to improve the efficiency of this algorithm and to make the algorithm applicable to larger systems, heuristics are used to reduce the search space. The major improvement introduced in this Thesis involves reducing the Petri net, effectively removing all the uncontrollable events [Ramadge and Wonham 87] from the model to obtain a further reduction in the search space of the scheduling algorithm. The improvement in the rate of convergence of the algorithm is backed up by experimental results presented in Chapter 4 of this Thesis.

1.7.3 A re-configurable supervisory controller for DEDSs

In DEDSs, such as computer or production systems, shared resources could break down at unknown intervals. As a result, jobs must be re-routed to make use of the available shared resources whenever such a situation occurs. In the event of a breakdown, the state-transition structure on which the supervisory controller is based, would not be a correct representation of the plant, unless all possible failures are included in the model.

It is not practical to model all possible failures in the supervisory controller because the additional states that need to be introduced would result in high CPU and memory requirements and make the system difficult to analyse. Therefore, in Chapter 4, a Petri net model-based controller is proposed. It uses state feedback to detect changes in the set-up and incorrect response of the system. It is re-configurable to accommodate to these changes. The design of the re-configurable supervisory controller was published in a workshop paper [Azzopardi and Holding 95].

1.7.4 A Petri net software tool kit

It is necessary to use computer programs to be able to analyse the dynamic model of large DEDSs and to implement scheduling and control algorithms, due to the large state space associated with these systems. In this Thesis, an object oriented C++ Petri net software library was implemented and has been used for the analysis and implementation of all the examples discussed in this Thesis. The class structure and facilities of the software library are described in Chapter 5.

1.8 Layout of the Thesis

This Thesis is organised into seven chapters. Chapter 2 presents a review of techniques for modelling DEDSs and introduces Petri net theory, which is most relevant to the research work presented in this Thesis. Chapter 3 discusses top down and bottom up Petri net synthesis techniques for modelling large systems. Since these techniques concentrate on functional abstraction they are limited to smaller sized DEDSs. To overcome this limitation, Chapter 3 presents a modified version of OMT (an object oriented design methodology) to synthesise models of large, industrial DEDSs.

Chapter 4 tackles the problem of optimising a DEDS by developing an efficient Petri net based scheduling algorithm to enable re-scheduling of the resources in the case of fluctuation in processing times. Chapter 4 also presents an introduction to the basics of supervisory control for DEDSs, the use of Petri nets for supervisory control and a novel re-configurable closed-loop controller for DEDSs.

Chapter 5 describes a Petri net software tool kit, implemented in object oriented C++, that enables the implementation of analysis tools for DEDSs, Petri net based scheduling algorithms and Petri net based control algorithms. Chapter 6 applies the design methodology to two industrial DEDSs and Chapter 7 summarises the methodology, assesses the contributions made in this Thesis and makes suggestions for further work.

Chapter 2

Modelling and analysis of DEDSs

2.1 Introduction

The increasing complexity of modern industrial DEDSs, makes intuitive solutions for their design and control inadequate. Therefore, according to leading researchers in the field, including Ramage and Wonham [89] and Cassandras [93], one must use a state transition structure (such as automata [Hopcroft and Ullmann 79] or Petri nets [Petri 62]), or a set of algebraic equations [Hoare 85] or a logical calculus such as temporal logic [Manna and Pnueli 83] for their analysis and design.

This Chapter investigates current techniques for modelling DEDSs including automata, state charts [Harel 87], Petri nets, temporal logics and process algebras [Hoare 85] in an attempt to choose the technique that is most suited to this research work. Petri net theory was found to be the most suitable technique because of the ease with which it is able to model DEDS characteristics; it is well developed and research in this field is intense; it allows automation of analysis to facilitate analysis of larger systems; it provides a system evolution mechanism that is useful for analysis and for implementing supervisory control; it provides a compact graphical representation and it models system precedence relations and timing information that are necessary for performance optimisation. Hence Petri net theory and Petri net extensions that are particularly relevant to this research are described in further detail in this Chapter

2.2 Techniques for modelling DEDSs

Briefly, a model of a DEDS is a mathematical representation of the features of the system, that are considered to be important for its operation. Several techniques for modelling DEDSs have been developed and are reviewed in [Davis 88, Joseph and Goswami 89, Scholefield 90, Ostroff 92b]. In this section, the more relevant modelling techniques are discussed.

2.2.1 Finite State Automata

One of the modelling techniques used to study the behaviour of DEDSs is based on theories of *languages* and *automata* [Hopcroft and Ullman 79]. A DEDS [Definition 1.1] has an underlying event set, **E**, associated with it, which is thought of as the alphabet of a language, and event sequences are thought of as words in the language.

Definition 2.1 A *language*, L, defined over an alphabet **E**, is a set of strings formed from events in **E**. A language may be thought of as a formal way to describe the behaviour of a DEDS by specifying all admissible sequences of events the DEDS is capable of following [Cassandras 93].

Definition 2.2 A finite state automaton is a device that is capable of generating a language in accordance to well defined rules and is defined by Hopcroft and Ullman [79] as a five-tuple $(\mathbf{E}, \mathbf{X}, \mathrm{f}, x_0, \mathbf{F})$, where

- $\mathbf{E}$ is a finite alphabet,
- $\mathbf{X}$ is a finite set of states,
- f is a state transition function, $\mathrm{f}: \mathbf{X} \times \mathbf{E} \rightarrow \mathbf{X}$,
- x_0 is the initial state, $x_0 \in \mathbf{X}$,
- $\mathbf{F}$ is a set of final states, $\mathbf{F} \subseteq \mathbf{E}$.

State automata are represented graphically by means of state transition diagrams (or *state diagrams* for short). A state diagram is a directed graph consisting of circles that represent states and arcs that denote events.

Example 2.1 Consider the automaton $(\mathbf{E}, \mathbf{X}, \mathrm{f}, x_0, \mathbf{F})$, [Cassandras 93], where

$$\mathbf{E} = \{\alpha, \beta, \gamma\}$$
$$\mathbf{X} = \{x, y, z\}$$
$$\mathrm{f}(x, \alpha) = x, \ \mathrm{f}(x, \beta) = \mathrm{f}(x, \gamma) = z$$
$$\mathrm{f}(y, \alpha) = x, \ \mathrm{f}(y, \beta) = \mathrm{f}(y, \gamma) = y$$
$$\mathrm{f}(z, \beta) = z, \ \mathrm{f}(z, \alpha) = \mathrm{f}(z, \gamma) = y$$
$$x_0 = \mathrm{x}$$

This is graphically represented by the state diagram of Fig. 2.1, where the initial state, {x}, is marked by an arrow and the set of final states, {x,z} is indicated by shaded circles.

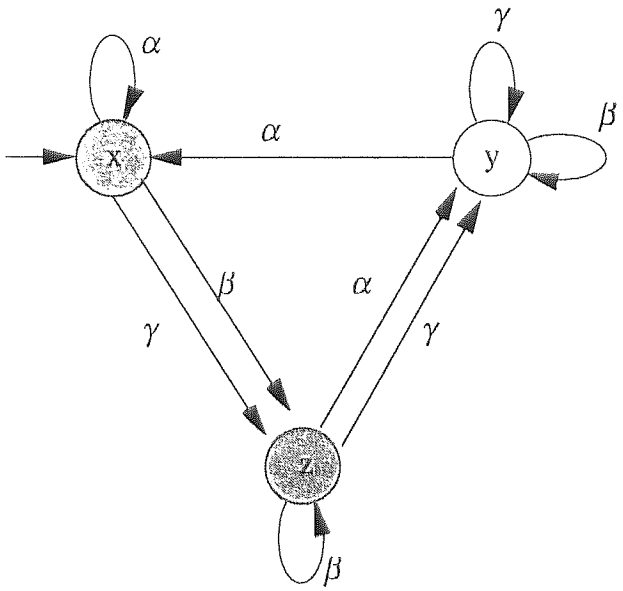


Fig. 2.1 A state diagram

Although state automata have been used in applications such as telephony [Davis 88], their application is limited to smaller applications because they are non-compositional and suffer from state-space explosion [Scholefield 90], where the number of states increases exponentially, resulting in an unstructured and chaotic diagram. The deficiencies in state automata have been partially overcome by the development of Statecharts [Harel 87] which are described in the following section.

2.2.1.1 Statecharts

Statecharts is a visual formalism developed by Harel [87], in an attempt to resolve the problems associated with state diagrams. Statecharts can have several layers so that one can "zoom-in" and "zoom-out" depending on the level of abstraction that is required, and provide a mechanism for communication between concurrent state machines to enable compositional design of large systems.

Rounded rectangles represent states and encapsulation is used to express the hierarchical relation between states. Events are represented by arrows which can optionally have a condition (guard) associated with them. This is illustrated in the statechart of Fig. 2.2 which has three states A, B and C. Event α, for example, transfers the system from state B to state A. Since event β takes the system to state B from either states A or C, the latter are "clustered" into a new "super-state" D. Then, state D represents the exclusive-or (XOR) of states A and C.

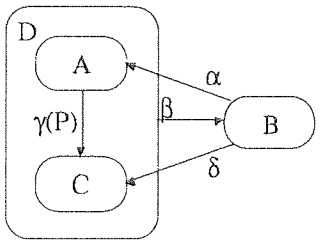
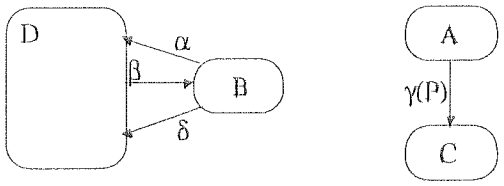


Fig. 2.2 Example of a statechart

Zooming-in is achieved by looking 'inside' D resulting in the diagram of Fig. 2.3 (b). Zooming-out is done by removing the states inside D and abstracting Fig. 2.2 to Fig. 2.3 (a). This facility enables the system designer to view the dynamic model at the level of abstraction that is desired and makes the diagram less chaotic than traditional state diagrams.



(a) (b)

Fig. 2.3 Zooming-in and zooming-out

Fig. 2.4 illustrates a system with concurrent states, where states B and C are concurrent with states E, F and G. The state Y represents the AND operator on states A and D. This feature of statecharts results in a more compact graphical representation of concurrent systems. A traditional state diagram would have required 6 states to model the system Y of Fig. 2.4.

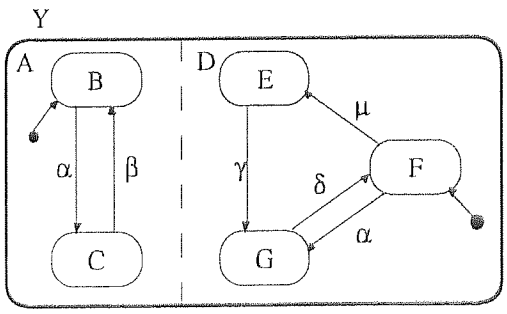


Fig. 2.4 Modelling concurrency

A software package called STATEMATE [Harel *et al.* 90], has been written to provide a working environment for the development of complex DEDS. It makes use of statecharts' semantics for representation of the behavioural model of the system and allows the user to step through the system evolution.

STATEMATE also provides facilities for simulation of the system and analysis of the dynamic behaviour of the system. Analysis is done by means of brute force methods (exploring all possible states) and gives information on reachability of states, non-determinism, deadlock and event traces.

Statecharts' semantics are incomplete and imprecise, resulting in several problems that are described in detail in [DerBeek 94]. To overcome these problems, the semantics of Statecharts have been altered and DerBeeck [94] describes 20 different variants of Statecharts.

2.2.2 Petri nets

A Petri net [Petri 62] is a graphical notation with an underlying mathematical theory for modelling and analysing DEDSs. Graphically, Petri nets can be used to visualise the evolution of the system. As a mathematical theory, it can be applied to set up the state equations of a system to analyse its dynamic behaviour.

Definition 2.3 A marked Petri net is defined by Murata [89] as a five-tuple $Z = (P, T, I, O, m)$, where:

P	is a finite set of *places*.
T	is a finite set of *transitions*, with $P \cup T \neq \emptyset$ and $P \cap T = \emptyset$;
$I: P \times T \rightarrow \{0,1\}$	is the transition *input function* that specifies the arcs directed from places to transitions;
$O: P \times T \rightarrow \{0,1\}$	is the transition *output function* that specifies the arcs directed from transitions to places;
$m: P \rightarrow \mathbf{N}$	is the marking representing the number of tokens in a place. ($\mathbf{N}$ represents the set of natural numbers)

In Petri nets, the set of states is represented by a set of places. The set of events is represented by a set of transitions. The active states of the system are illustrated by tokens in the relevant places. Graphically, places are depicted as circles, transitions as filled bars and tokens as dots in the circles that represent places. The places and transitions in the Petri net are linked by means of directed arcs. The set of places linked by directed arcs into a transition represents the input function, whilst the set of places linked by arcs directed out of transitions represent the output function. Fig. 2.5 illustrates a Petri net graph.

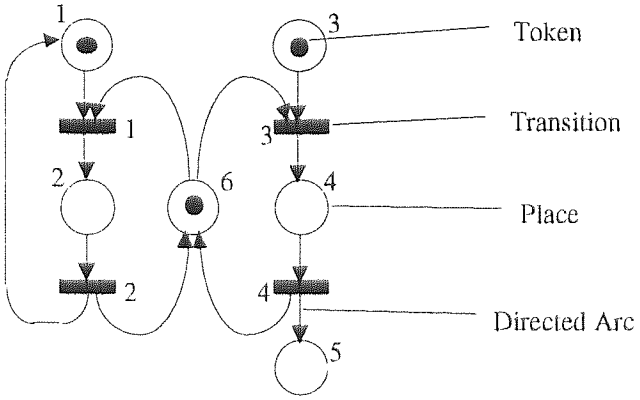


Fig. 2.5 A Petri net graph

The marking of a Petri net represents the state of the system that it models. Tokens flow through the Petri net, representing the evolution of the system, according to the rule:

Definition 2.4 A transition $t \in T$ is said to be enabled iff each input place of t contains at least one token. An enabled transition, $t \in T$, may fire at marking m', yielding the new marking, m, where $m(p_i) = m'(p_i) + O(p_i,t) - I(p_i,t)$ for $i = 1,2,...,\#(P)$

In addition to showing how the modelled system evolves, Petri net theory can be used to analyse its behaviour for desirable properties such as boundedness, liveness and reversibility [Murata 89, David and Alla 92, Desrochers and Al-Jaar 95]. Boundedness implies that the number of states that a system may enter is finite, liveness indicates that there is no deadlock and reversibility shows that the system has cyclic behaviour.

According to Ostroff [92], Petri net theory was one of the first formalisms to deal with concurrency, non-determinism and causal connections between events. A problem with ordinary Petri nets is that large nets are difficult to analyse. However Petri net theory is the subject of intense research and methods for analysis of large nets have been developed and will be discussed in this Thesis.

2.2.3 Temporal Logics

An alternative approach is to use classical logic [Hamilton 78], including propositional (statement) and predicate (first order) logic to model DEDSs. In propositional logic, simple statements are denoted by capital letters (A, B, C, ...) and the connectives listed in Table 2.1 are used to form compound statements (propositions).

not A	$\neg A$
A and B	$A \wedge B$
A or B	$A \vee B$
if A then B	$A \rightarrow B$
A if and only if B	$A \leftrightarrow B$

Table 2.1 Connectives of classical logic

In predicate logic, statements have a subject and a predicate. Roughly, the subject is the thing about which the statement is making an assertion and the predicate refers to a property which the subject has. For example, the statement "there is at least one sheep that is black" would be represented by the predicate logic statement $(\exists x)(P(x) \wedge W(x))$ where, the symbol $(\exists x)$ means that "there exists at least one object x such that" and $P(x)$ and $W(x)$ mean "x is a sheep" and "x is black" respectively.

When using classical logic to specify a DEDS, the states of the system are not individually enumerated, but groups of states are characterised by means of logical propositions. When a proposition is true, it means that the state of the system belongs to the corresponding group. Valette [95] pointed out that there are two main difficulties associated with using classical logic to specify DEDS: (i) It is difficult to define the meaningful groups and the corresponding propositions; (ii) Classical logic is inadequate to reason about time and state evolution because, once a proposition has been proved true, it has to remain true, otherwise an inconsistency would be detected.

In order to reason about systems in which changes occur, modal logic [Hughes and Cresswell 68] is used. This logic has the special operators which specifies change: (i) $\Diamond$ (possibility operator), and (ii) $\Box$ (necessity operator). The semantics of modal logic are denoted by the Kripke structure [Kripke 63] which considers a system to be composed of a set of worlds (W) and an accessibility relation (R). W represents all the possible states that a system may be in, and R defines how the system changes from one world to another.

In order to reason about concurrent systems, it is necessary to use some kind of temporal logic [Manna and Pnueli 83]. Temporal logic is a form of modal logic where R is interpreted as the passage of time and σ is a sequence of states. Formulas of temporal logic are constructed using the temporal operators: $\Box$, $\Diamond$, $\mathbf{O}$, $\mathcal{U}$, which are called the 'always', 'eventually', 'next' and 'until' operators respectively and are interpreted over the Kripke structure as:

$\sigma = s_0, s_1, \ldots$

$R(s_n, s_{n+1})$

where

s_n, where $n \geq 0$, represents a state, and

$R(s_n, s_{n+1})$ is the accessibility relation.

The semantics of the temporal operators are defined below. For a state sequence σ, let the sequence $\sigma^{(k)}$ be the k-shifted sequence given by $\sigma^{(k)} = s_k, s_{k+1}, \ldots$, then,

(i) if w is a classical formula (constructed from propositions or predicates and logical operators listed in Table 2.1), containing no temporal operators, then,

$\sigma \models w$ iff $s_0 \models w$

in this case w can be interpreted over a single state in σ, which is the initial state s_0,

(ii) the temporal operator $\Box$ (always) is defined as:

$\sigma \models \Box w$ iff $\forall k \geq 0, \sigma^{(k)} \models w$

which means that $\Box w$ holds on σ iff all states in σ satisfy w,

(iii) the temporal operator $\Diamond$ (eventually) is defined as:

$\sigma \models \Diamond w$ iff $\exists k \geq 0, \sigma^{(k)} \models w$

which means that $\Diamond w$ holds on σ iff at least one state in σ satisfies w,

(iv) the temporal operator **O** (next) is defined as:

$\sigma \models \mathbf{O}w$ iff $\sigma^{(1)} \models w$

which means that $\mathbf{O}w$ holds on σ iff $\sigma^{(1)}$ satisfies w,

(v) the temporal operator $\mathcal{U}$ (until) is defined as:

$\sigma \models x \mathcal{U} y$ iff $\exists k \geq 0$ such that

$\sigma^{(k)} \models y$, and $\forall i, 0 \leq i \leq k, \sigma^{(i)} \models x$

which means that $x \mathcal{U} y$ holds on σ iff at some time y holds, and until then, x holds continuously.

There are several types of temporal logic that differ in the way in which they model time. These include branching time, linear time and partial order temporal logics. Branching temporal logic views time as a tree like structure, in which, at each node, the future has several possible alternatives. Linear time temporal logic views time as having only one possible future whereas partial order temporal logic considers the partial ordering of events in time.

When using temporal logic to model DEDSs it is often necessary to explicitly refer to a state transition graph to define the worlds (states), where the propositions are consistent with the classical logic framework. According to Valette [95], all reasoning about state changes has to be based on a state-transition graph and the typical search for invariants (propositions which are true for all the states), although useful for formal proof of computer program correctness [Galton 87], including safety and liveness properties (defined in [Alpern 85]), is useless for specification of the system evolution.

2.2.4 Process Algebras

Process Algebras consist of a set of processes and operators that provide process construction. The process algebra defines a set of equations which provide a proof theory. One such algebraic approach is communicating sequential processes (CSP) developed by Hoare [78] and described in detail in [Hoare 85]. It is a notation for specification and verification of systems of concurrent processes and makes use of three fundamental semantic models: trace model, failure model and stability model.

A trace is a finite sequence of events, where an event is an action performed by a process and is considered to be instantaneous. A process, P, is formally defined by the trace model as:

$$P \quad \textbf{sat} \quad S(tr)$$

where S(tr) is a specification stating the behaviour of this process in terms of traces.

A failure model is used to distinguish between deterministic and non-deterministic behaviour, by specifying a process in terms of refusal sets. A refusal set, such as refusals(P), is a set of event sequences which is refused by process P. For a non-deterministic process, both traces and refusals are used to specify its behaviour, therefore:

$$P \quad \textbf{sat} \quad S(tr, ref)$$
$$\Rightarrow \forall (tr, ref) \mid \begin{pmatrix} tr \in traces(P) \\ \wedge\ ref \in refusals(P / tr) \end{pmatrix}$$

which is equivalent to the failure model, defined as:

$$failures(P) = \{ (tr, ref) \mid tr \in traces(P) \wedge ref \in refusals(P / tr) \}$$

The stability model is used to model internal transactions of a process and to investigate whether a particular process has stabilised at some internal event.

CSP processes communicate by sending messages via channels. A process, P, consists of a pair of communicating processes A and B (denoted by (A||B)) which synchronise on a communicating event, C, where process A sends an output message, M, to process B. This is illustrated in Fig. 2.6.

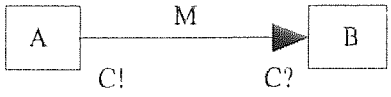


Fig. 2.6 CSP model of communication [Lau 91]

CSP provides a process algebra to specify a process in terms of its set of traces and refusals. Several proof systems exist [Olderog 86] whereby the trace model can be used to prove safety properties whilst the failure model can be used to prove liveness properties. A drawback of using CSP to specify and verify DEDS, pointed out by Lau [91], is that the verification technique used by CSP is based on an axiomatic approach and to construct a full mathematical proof requires an in depth understanding of the different inference systems. Another disadvantage of CSP is that it is event based where the state of a system is only a derived notion and is not specified explicitly (as in state automata, Petri nets and temporal logic).

2.2.5 Comparison of modelling techniques

Several authors [Peterson 81, Cassandras 93] have compared the modelling capabilities of state automata and Petri nets. Cassandras [93] states that the choice of using state automata or Petri nets is often subject to personal biases however, in this section, the most important modelling features are compared.

First of all, a state automaton can be always be represented by a Petri net. The automaton $(\mathbf{E}, \mathbf{X}, \mathrm{f}, x_0, \mathbf{F})$ can be represented by the Petri net Z = (**P**, **T**, **I**, **O**, m_0), where;

$$\mathbf{P} = \mathbf{X}$$

$$\mathbf{T} = \{(x, x') : x \in \mathbf{X}, x' = \mathrm{f}(x, e) \forall e \in \Gamma(x)\}$$

$$\mathbf{I} = \{(x, t) : x \in \mathbf{X}, t \in \mathbf{T}\}$$

$$\mathbf{O} = \{(t, x) : x \in \mathbf{X}, t \in \mathbf{T}\}$$

and the initial state, x_0, is represented by marking the corresponding places in the Petri net, (m_0).

According to Cassandras [93] a disadvantage of Petri nets is that the graphical representation of realistically sized DEDSs has many places, transitions and links, resulting in a spaghetti-like diagram. However, in such a situation, the graphical representation is still useful since one can focus on the section of the net that is of interest. On the other hand, state automata of a simple system can consist of an infinite number of nodes. Consider the queuing system in Fig. 2.7, where 'a' represents the arrival of a customer to the queue and 'd' represents a departure of a customer after being processed. The processor is only capable of handling one job at a time.

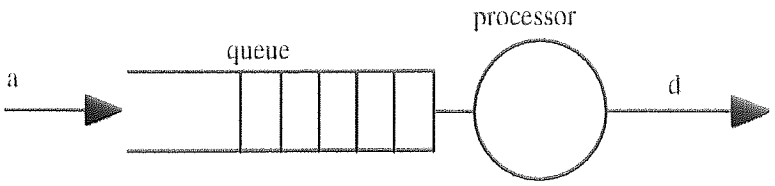


Fig. 2.7 A queuing system

The state diagram representing this system consists of an infinite number of nodes, whereas a Petri net can describe the same system without having to explicitly show all possible markings. The equivalent models are illustrated in Fig. 2.8 and clearly show the advantage of using the compact notation of Petri nets.

Another advantage of using Petri nets over state diagrams is that the Petri net structure allows the modular design of a system. Suppose that we want to model a system consisting of two systems, say, system 1 and system 2. If the two systems are modelled as state automata with state spaces $\mathbf{X}_1$ and $\mathbf{X}_2$ respectively, the state space of the whole system, $\mathbf{X}$, consists of all possible combinations of the individual system states (i.e. $\mathbf{X} = \mathbf{X}_1 \times \mathbf{X}_2$). This means that combining multiple systems rapidly increases the complexity of a state diagram. On the other hand, Petri nets allow the modular design of systems, in which the individual component Petri nets are combined by adding places and transitions to represent the coupling effects between the systems. The subject of Petri net synthesis for large systems is well documented [Jeng and DiCesare 93] and is covered in more detail in Chapter 3.

The deficiencies in state diagrams have been partially overcome by the development of Statecharts which, however, have incomplete and imprecise semantics, resulting in 20 different variants of Statecharts. Another disadvantage of using Statecharts is that analysis of the system behaviour is done by means of brute force methods, which limits the size of systems that can be analysed within a reasonable time.

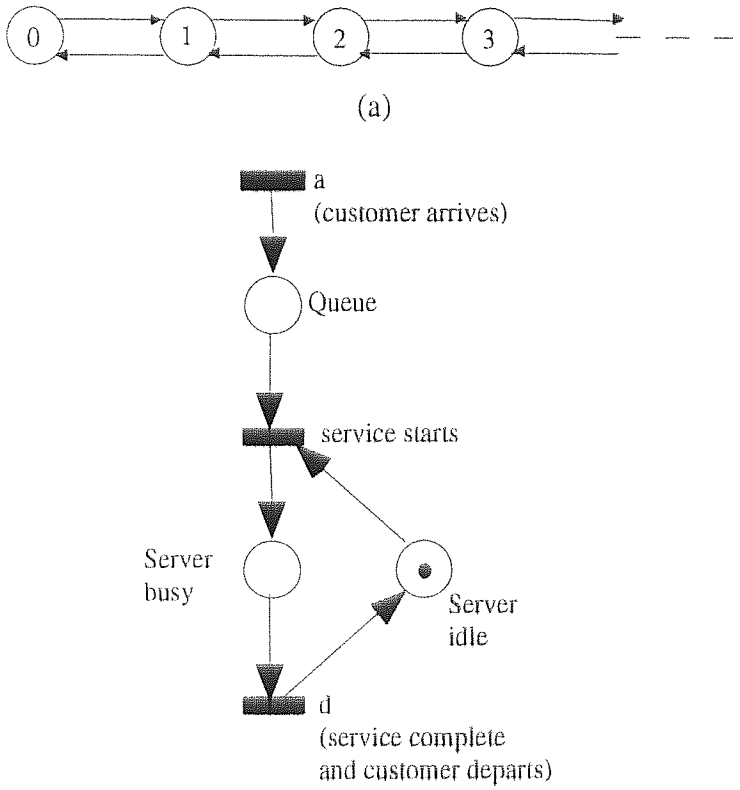


Fig. 2.8 Comparison of state diagram and Petri net

The formal methods (temporal logic and process algebras) described in the previous sections are complementary. For example, temporal logic is good at describing properties that concern the whole system, such as liveness and safety. However, according to Valette [95] it is not suitable for specifying the system evolution, also, according to Ostroff [92] temporal logic specifications are relatively unstructured and could benefit from the more structured notations of process algebras. The use of formal methods results in specifications that are difficult to understand and according to Hall [90], a high level of mathematical skill is needed to carry out a fully formal development (of a DEDS) that includes proofs, and it is unrealistic to expect the majority of software engineers to be able to do proofs easily.

The advantages of Petri theory, over the techniques described in this Chapter are:

- The ease of modelling DEDS characteristics including: concurrency, asynchronous and synchronous features, conflicts, mutual exclusion, precedence relations, non-determinism and system deadlocks,
- Analysis can be performed by using mathematically based computer software,
- It is well developed and backed up by over 25 years of intense research and a large Petri net research community,

- It allows a concise and intuitive graphical representation of the model to aid visualisation of the problem and facilitate communication of the problem amongst the system designers,
- It is useful for performance optimisation (scheduling exclusive-use resources) because the model contains both the system precedence relations and timing information.

For the reasons mentioned above, Petri net theory is the theory adopted for the research work presented in this Thesis. It is described in more detail in the following sections.

2.3 Petri net Theory

A Petri net (defined in Section 2.2.2) is a graphical notation with an underlying mathematical structure [Definition 2.3] for modelling and analysing DEDSs. As a graphical notation, a Petri net is used to visualise DEDS characteristics [Zhou and DiCesare 93] such as: synchronisation, concurrency, conflicts, resource sharing, precedence relations, non-determinism and system deadlocks. As a mathematical tool, Petri net theory is used to define the state equations of a system and to analyse its dynamic behaviour [Murata 89, David and Alla 92, Zurawski and Zhou 94].

There are three approaches for analysis of behavioural and structural properties of the Petri net model: The coverability graph approach [2.3.3], the incidence matrix approach [2.3.4] and the Petri net reduction approach [2.3.5].

2.3.1 Behavioural properties

The most important behavioural properties of a DEDS are reachability, boundedness, liveness, deadlock states, quasi-liveness, home state, and reversibility [David and Alla 92, Zurawski and Zhou 94, Desrochers and Al-Jaar 95]. It is important to note that behavioural properties of a system depend on its initial conditions. Therefore, a system that is, say, live, under a particular initial condition is not necessarily so under another initial condition.

Definition 2.6

A marking m_r is said to be *reachable* from a marking m_0 iff there exists a sequence of transitions that transforms m_0 to m_r [Peterson 81, Murata 89]. That is, m_r is reachable from a marking m_0 iff $\exists S: m_0[S > m_r$, where S is a sequence of transitions and $m_0[S > m_r$ means that m_r is reachable from m_0 by firing the sequence of transitions, S.

This property can be used to verify, for example, that the system can reach a desirable state, or that it will never reach a hazardous state, from its initial conditions.

Definition 2.7

A place is *k-bounded* if the number of tokens in that place never exceeds k in any of the reachable markings of the Petri net [Peterson 81, Murata 89]. That is, a place $p \in P$ is *k-bounded* iff $\exists k \in \mathbb{N} : m(p) \leq k$, $\forall m \in R(Z, m_0)$ where $R(Z, m_0)$ is the set of markings of the marked Petri net Z, reachable from initial marking m_0.

A marked Petri net Z is k-bounded iff all its places are k-bounded. That is, Z is k-bounded iff $p \in P$ is k-bounded, $\forall p \in Z$. Furthermore, Z is *safe* iff it is 1-bounded.

If a Petri net is used to model storage buffers or production units that have finite capacity, k, then the k-boundedness of a place representing a buffer or production unit ensures that there will be no overflow [Murata 89].

Definition 2.8

Z is *live* if for any marking reachable from m_0, it is ultimately possible to fire any transition of the net by progressing through some further firing sequence [Peterson 81, Murata 89].

Liveness guarantees deadlock free operation. A transition T_j is said to be *quasi-live* if for initial marking m_0, there is at least one firing sequence which contains T_j [David and Alla 92].

Definition 2.9

A marking m_h is a home state for an initial marking m_0, if for every reachable marking there exists a sequence of transitions that transforms this marking into m_h [David and Alla 92]. That is, m_h is a home state for an initial marking m_0, if $\forall m_i \in R(Z, m_0)$, $\exists S_i : m_i[S_i > m_h$.

A Petri net is reversible if the home state is the initial marking [David and Alla 92]. That is, Z is reversible iff $m_0 \in R(Z, m), \forall m \in R(Z, m_0)$.

Reversibility is an important property that means that the system will finally return to its initial state from any reachable state. This property is directly related to the automatic error recovery problem [Narahari and Viswanadham 85, Zhou and DiCesare 93].

2.3.2 Structural Properties

Structural properties of a Petri net, unlike behavioural properties, are independent of the initial marking. The following structural properties are of interest:

Definition 2.10

A marked Petri net, Z, is structurally bounded if it is bounded [Definition 2.7] for any finite initial marking [Desrochers and Al-Jaar 95, Murata 89].

Definition 2.11

Z is structurally live if it is live [Definition 2.8] for any finite initial marking [Desrochers and Al-Jaar 95, Murata 89].

Definition 2.12

A trap is a set of places such that every transition that inputs from one of these places also outputs to one of these places. So, once a place in a trap has a token, there will always be a token in at least one of the places in the trap [David and Alla 92, Desrochers and Al-Jaar 95].

Definition 2.13

A siphon is a set of places such that every transition that outputs to one of these places also inputs from one of these places. This means that once all places in a siphon have no token, there will never be a token in any one of the places in the siphon [David and Alla 92, Valette 95].

To illustrate the significance of siphon structures in Petri net models, consider the siphon structure, comprising the set of places {p2, p3, p12, p13, p22, p23}, illustrated in Fig. 2.9: If all the places are empty, none of the transitions may be fired because all of them have at least one input place belonging to the net. Therefore, once empty, no sequence of firing of transitions could possibly re-introduce a token into the siphon resulting in deadlock.

This situation is reachable if it is possible to empty the siphon by firing transitions ta and te respectively. Therefore the liveness of Petri nets that contain siphon structures is dependent on the initial marking and on whether there can be a situation where the siphon is emptied.

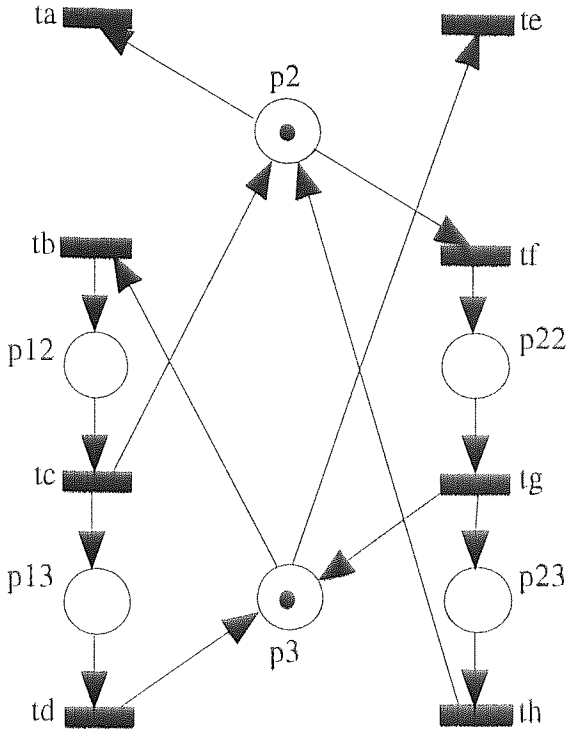


Fig. 2.9 Example of a siphon [Valette 95]

Definition 2.14

Parallel Mutual Exclusion (PME) places [Zhou and DiCesare 91] are used to model exclusive-use shared resources in a DEDS. In order to define a PME, places are categorised as:

- The set of A-places, P_A, representing processes
- The set of B-places, P_B, representing resources (1-bounded)
- The set of C-Places, P_C, representing buffers (n-bounded)

An elementary path [Berge 62], EP, is a sequence of nodes $x_1, x_2, x_3, \ldots x_n, n \geq 1$, such that $\exists$ an arc $(x_i, x_{i+1}) \forall i \in \mathbb{N}_{n-1}$, where $\mathbb{N}_{n-1}$ is the set of natural numbers {1, 2, ... , n-1}. An A-path between nodes x and y is an elementary path between node x and node y, EP(x,y), each of whose nodes, except for x and y is either an A-place or a transition.

Then, given a marked Petri net, Z = (P, T, I, O, m_0), $P = P_A \cup P_B \cup P_C$, a k-parallel mutual exclusion (Fig. 2.10) is defined as a k-PME=(p_E,D) such that:

1. $p_E \in P_B$ with $m_0(p_E) = 1$, D is a set of transition pairs, $D = \{(t_{a1}, t_{b1}), (t_{a2}, t_{b2}), \ldots, (t_{ak}, t_{bk})\}, k \geq 1$, satisfying the following conditions:

(a) All transitions in D must be different, except for t_{ai} and t_{bi}, therefore, $t_{ai}, t_{bi} \in T$, $t_{ai} \neq t_{bj}$, $t_{ai} \neq t_{aj}$, and $t_{bi} \neq t_{bj}$ when $i \neq j, \forall i, j \in \mathbb{N}_k$, where $\mathbb{N}_k = \{1, 2, \ldots, k\}$

(b) There is one input arc from p_E to each t_{ai} and one output arc from t_{bi} to p_E for $1 \leq i \leq k$ but no other arcs related to p_E, therefore, $I(p_E, t_{ai}) = O(p_E, t_{bi}) = 1$, $I(p_E, t_{bi}) = O(p_E, t_{ai}) = 0$ when $t_{bi} \neq t_{ai}, \forall i \in \mathbb{N}_k$;
$I(p_E, t_u) = O(p_E, t_u) = 0$ when $t_u \notin T_a \cup T_b$, where
$T_a = \{t_{ai}, i \in \mathbb{N}_k\}$, and $T_b = \{t_{bi}, i \in \mathbb{N}_k\}$;

(c) Any elementary path between t_{ai} and a C-place has to contain t_{bi} for $1 \leq i \leq k$, therefore, $\forall i \in \mathbb{N}_k, p \in P_C$, if $p \in EC(t_{ai}), t_{bi} \in EC(t_{ai})$;

(d) Any elementary circuit including t_{ai}, the shared resource place p_E, but no other B-places or C-places, must include t_{bi} for $1 \leq i \leq k$, therefore, $\forall EC(t_{ai})$, if $EC(t_{ai}) \cap (P_B \cup P_C) = \{p_E\}$, $t_{bi} \in EC(t_{ai})$; and

(e) Each transition on an elementary path between t_{ai} and t_{bi} should be on one A-path between t_{ai} and t_{bi} for $1 \leq i \leq k$, therefore, if $t \in EP(t_{ai}, t_{bi})$, t is on an A-path $EP(t_{ai}, t_{bi})$

2. If a resource place is not initially marked, then its output transition is never enabled. That is, the output transitions of p' are never enabled $\forall p' \in P_B \cup P_C, m \in R(Z, m_0)$ if the initial marking $m_0(p') = 0$ and

$$m_0(p) = \begin{cases} 0; & \text{if } p \in P_A \\ \geq 1; & \text{if } p \in P_B \cup P_C - \{p'\} \end{cases}$$

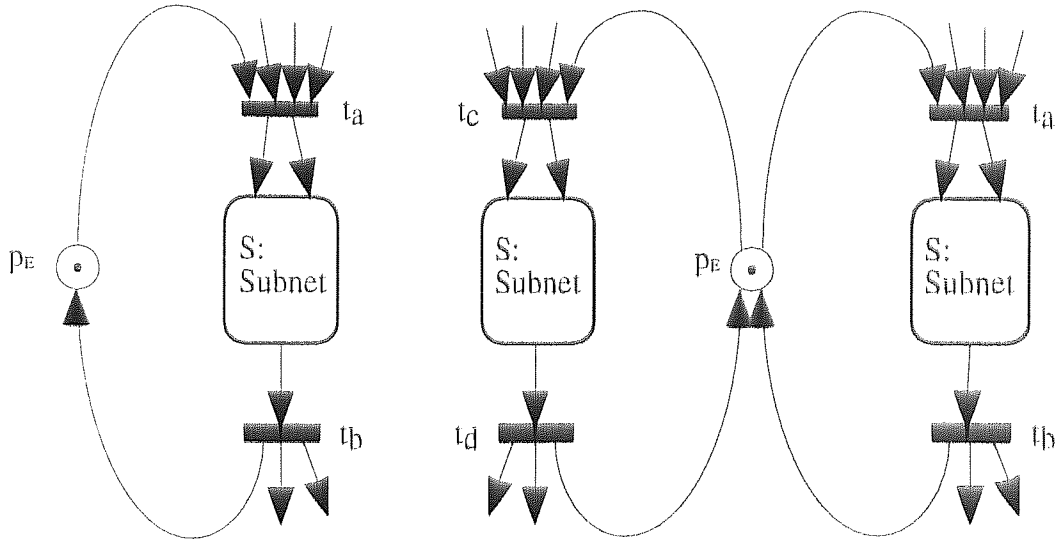
3. Given a marked Petri net, Z, if $p_1 \neq p_2$, and if $\exists EP(p_1, t)$ and $EP(p_2, t)$ are A-paths, then their first intersection occurs at a transition.

4. Tokens for different processes cannot be mixed together, that is, there is no A-path between t and t', $\forall t \in T_b, t' \in T_a$. Where T_a and T_b are the sets of transitions associated with processes a and b respectively.

5. Each process must have an equal opportunity to compete for and acquire the resource. Therefore, $\exists m_0$, and a sequence of fireable transitions g that contains no transition in T_a, such that $m_0[g >$ enables $t, \forall t \in T_a$.

6. The shared resource must be eventually released from the process, therefore, $\forall m_0$, if t_{aj} fires at $m \in R(Z, m_0)$, then $\forall t \in T$ if $EP(t_{aj}, t) \neq 0$ and $t_{bj} \notin EP(t_{aj}, t)$, t can be enabled and $\forall$ fireable g_j containing no t_{bj}, $\exists h_j$, such that $m[t_{aj} g_j h_j >$ enables t_{bj}; if $m(p_E) = 1$ and t_u is enabled, then $t_u \notin T_b$.

Zhou and DiCesare [93] studied the conditions under which Petri nets containing PME structures are live, bounded and reversible.



(a) (b)

Fig. 2.10 (a) A general case 1-PME, (b) A general case 2-PME

Definition 2.15

A sequential mutual exclusion (SME) [Zhou and DiCesare 93] is a mutual exclusion where two transitions, say, t_{a1} and t_{a2} can be enabled simultaneously only after t_{a1} fires from some initial marking before t_{a2}. SMEs are used to model exclusive-use shared resources that are used in more than one stage of a sequential process.

Given a marked Petri net Z = (P, T, I, O, m_0), $P = P_A \cup P_B \cup P_C$, a sequential mutual exclusion (SME) exists iff

1. $\exists (p_E, D)$ such that $D = D^1 \cup D^2 \cup \ldots \cup D^L, L \geq 2, k_i = |D^i|$

   $D^i = \left\{ \left(t^i_{aj}, t^i_{bj} \right), j \in \mathbb{N}_{k_i} \right\}$, satisfying:

   (a) $\left(p_E, D^i \right)$ forms a k_i-PME if all arcs of p_E related to transitions in $D - D^i$ are deleted, therefore, $\left(p_E, D^i \right)$ is a k_i-PME in Z_i = (P, T, I, O, m_0) that results from Z with $I(p_E, t) = O(p_E, t) = 0$, $\forall t \in D - D^i$;

(b) For each transition t in a PME with a higher index, there is a transition t' in a PME with a lower index such that an A-Path exists between t and t', therefore, $\forall t \in T_a^j, 1 \le i < j, \exists t' \in T_a^i$, such that the A - path EP(t' ,t) $\neq 0$ where $T_a^i = \left\{ t_{av}^i, v \in \mathbf{N}_{k_i} \right\}$;

(c) There is no A-path from D^{i+1} to D^i, therefore, if an A-path $\mathrm{EP}(t_{bu}^i, t_{av}^j) \neq 0$ for $u \in \mathbf{N}_{ki}, v \in \mathbf{N}_{kj}$, then i<j; and

(d) Any elementary path containing t_{bu}^i and t_{bv}^j have t_{av}^j when i<j, therefore, if $\mathrm{EP}(t_{bu}^i, t_{bv}^j) \neq 0$ for $u \in \mathbf{N}_{ki}, v \in \mathbf{N}_{kj}$, i<j, then $t_{av}^j \in \mathrm{EP}(t_{bu}^i, t_{bv}^j)$.

2. There exist sequential relations among different groups of processes, therefore, $\forall m_0$, g_i and $i \in \mathbf{N}_{L-1}$, if $m_0[g_i >$ enables $t_{aj}^{i+1}, \forall j \in \mathbf{N}_{k_{i+1}}$, then $\exists u \in \mathbf{N}_{k_i}, \ni \#(g_i, t_{au}^i) = \#(g_i, t_{bu}^i) \ge 1$. Where $\#(g_i, t_{bu}^i)$ is the number of times t_{bu}^i appears in g_i.

3. Each subnet Z_i has the property that once the transition t_{av}^i in D^i fires at marking m, a sequence of transitions can be found to enable the other transition t_{bv}^i from any marking reachable from m in Z_i. Hence, $\forall m_0$, if t_{av}^i fires at $m \in R(Z_i, m_0)$, then $\forall$fireable g_j containing no $t_{bv}^i, \exists h_j$ containing no transition in T_a^j, $1 \le j < i$ if i>1, such that $m[t_{av}^i g_j h_j >$ enables t_{bv}^i in Z_i.

The Petri net in Fig. 2.11 is an example of an SME in which transitions t1 and t3 are both enabled but cannot fire simultaneously.

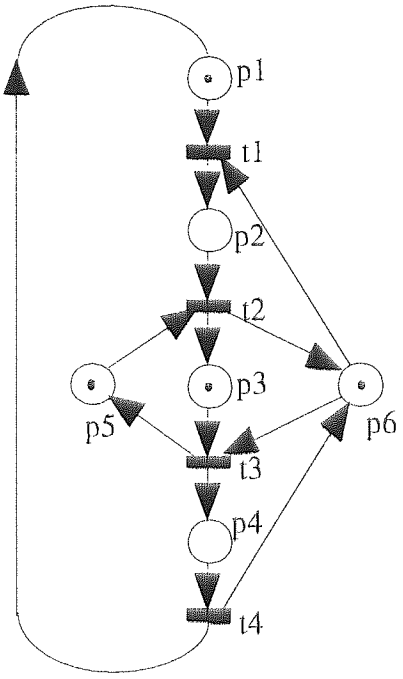


Fig. 2.11 Example of an SME

2.3.3 The Coverability Graph

A reachability tree is a graph of all possible markings that the Petri net may reach starting off from a specific initial marking. The nodes of this graph represent the marking of the Petri net whilst the arcs represent the firing of a transition. The reachability tree is actually the state-transition diagram equivalent of the Petri net. As with state automata [2.2.1], the reachability tree suffers from the state explosion phenomenon [Murata 89], when it continues to grow indefinitely because of unbounded or cyclic behaviour of the system.

To overcome this problem, a coverability graph is used. This is obtained by merging nodes which correspond to the same marking and therefore contains a finite number of nodes [David and Alla 92]. In the case of unbounded places, the marking in the coverability graph is represented by an 'ω', which can be interpreted as 'infinity'.

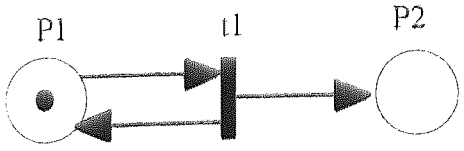


Fig. 2.12 An unbound Petri net

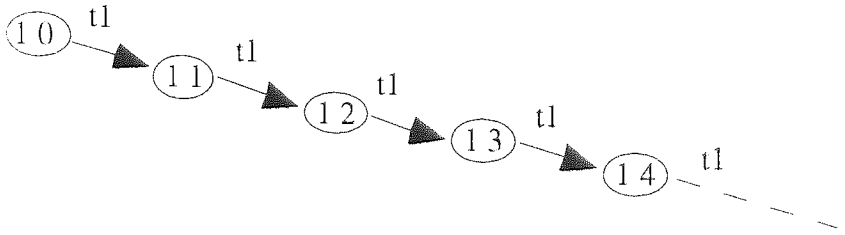


Fig. 2.13 An infinite reachability tree

Consider the unbound Petri net in Fig. 2.12. The reachability tree (Fig. 2.13) is infinitely large, whereas the coverability graph (Fig. 2.14) consists of only two nodes, {1,0} and $\{1, \omega\}$, where $\omega = \{1, 2, \ldots, \infty\}$.

Fig. 2.14 A coverability graph

Using a coverability graph it is possible to analyse the Petri net for behavioural properties by means of an exhaustive search. However, for an unbounded Petri net, liveness and reachability cannot be verified from the coverability tree alone due to a loss of information brought about by the use of the 'ω' notation [David and Alla 92].

2.3.4 The incidence matrix

An n-place, m-transition Petri net graph can be described by means of an $n \times m$ incidence matrix [Murata 89], **A**, and a marking vector **m**. The incidence matrix describes the interconnection between places and transitions of the Petri net and the marking vector indicates the places that are initially marked. These are defined as:

Definition 2.16

$$\mathbf{A}[i,j] = \begin{cases} 0; & \text{if there is no arc linking place } i \text{ and transition } j \\ -1; & \text{if place } i \text{ is an input place to transition } j \\ 1; & \text{if place } i \text{ is an output place from transition } j \end{cases}$$

$$\mathbf{m}[i] = \begin{cases} 0; & \text{if there are no tokens in place i} \\ 1; & \text{if there is a token in place i} \end{cases}$$

The incidence matrix can be used to generate both the coverability graph and P- and T-invariants, which in turn can be used to verify certain structural properties such as conservative components and repetitive components [David and Alla 92].

2.3.4.1 Definition of P- and T- invariants

For an ordinary Petri net with incidence matrix $\mathbf{A}$:

Definition 2.17

A P-invariant is an $(n \times 1)$ non-negative integer vector $\mathbf{x}$, satisfying the equation

$$\mathbf{x}^T\mathbf{A} = 0, \text{ where } n = \#(P)$$

The non-zero elements of the vector $\mathbf{x}$ represents the set of places in which the total number of tokens is constant.

Definition 2.18

A T-invariant is an $(m \times 1)$ non-negative integer vector $\mathbf{y}$ satisfying the equation

$$\mathbf{A}\mathbf{y} = 0, \text{ where } m = \#(T)$$

The non-zero elements of the vector $\mathbf{y}$ represent a set of transitions which, if fired in a particular order, bring the net to the same marking that it was in prior to firing the transitions.

From the P- and T-invariants of a Petri net, the following properties can be proved about the dynamic behaviour of a DEDS:

Boundedness If all the places in a pure Petri net (i.e. a net that does not contain self-loops) are included in its set of P-invariants, and the initial marking, m_0, is bounded, then the net is bounded [Desrochers and Al-Jaar 95].

Liveness and boundedness A pure Petri net is live and bounded if all the places are included in the P-invariants, all the P-invariants are marked and none of the siphon structures (if any) are cleared [Desrochers and Al-Jaar 95].

Conservative components The set of places in a P-invariant is referred to as a conservative component because the number of tokens marking this set of places is a constant [Murata 89]. Thus P-invariants can be used to prove mutual exclusion of states [David and Alla 92], which is often required to prove safety properties of a system.

Repetitive components The set of transitions in a T-Invariant is referred to as a repetitive component since, by firing the transitions in a T-invariant, starting from an initial marking, m_0, the resultant marking will be m_0 [David and Alla 92].

Solution of the equations in definitions 2.17 and 2.18 involves solving a set of homogeneous simultaneous equations and if $m=n$, then the equations can be solved by using Cramer's rule. Solving these equations present the following problems:

(i) If the determinant of the incidence matrix, **A**, det(**A**) $\neq$ 0 then there is only one solution, with all the unknowns equal to zero. This is often called the trivial solution.

(ii) If det(**A**)=0, then there will be infinitely many solutions other than the trivial solution, which means that at least one of the equations can be obtained from the others.

(iii) If $m \neq n$, the determinant of matrix A is undefined. In this case, if $m>n$ there are an infinite number of solutions and if $m<n$, there are no solutions at all.

2.3.4.2 Graphical Solution of P-invariants

To simplify the procedure of obtaining the P-invariants, David and Alla [92] present a graphical method that consists of two reduction rules that preserve the P-invariants of the Petri net.

2.3.4.2.1 R_a - Self loop transition This reduction rule removes self loop transitions whilst preserving the P-invariants of the Petri net. That is, transitions that are connected to a place which is both an input and an output place. Consider the Petri net in Fig. 2.15 (a). It can be reduced by:

(i) Suppressing arcs forming a self-loop (Fig. 2.15 (b))
(ii) Suppressing a transition if it is isolated (Fig. 2.15 (c))

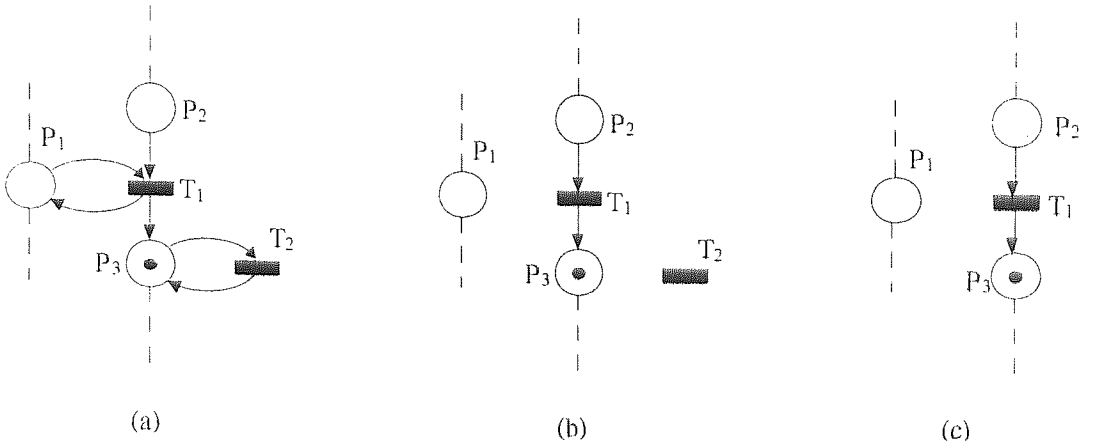


Fig. 2.15 Reduction R_a (self-loop transition)

2.3.4.2.2 R_b - Pure transition This reduction rule operates on pure transitions. A Transition T_j, is pure if it has at least one input and one output. The reduction, illustrated in Fig. 2.16, is done by performing the following:

1. T_j is suppressed
2. Place $P_i + P_k$ is associated with every pair of places (P_i, P_k) such that $P_i \in {}^{o}T_j$ and $P_k \in T_j^{o}$, where ${}^{o}T_j, T_j^{o}$ represent the set of input and output places of T_j respectively. The number of tokens in $P_i + P_k$ = sum of tokens which were initially in (P_i, P_k).
3. The input transitions of $P_i + P_k$ are the input transitions of P_i and P_k except for T_j. The output transitions of $P_i + P_k$ are the output transitions of P_i and P_k except T_j.

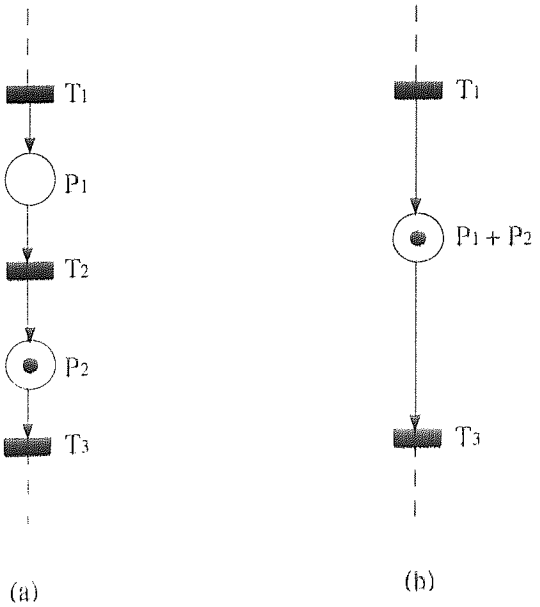


Fig. 2.16 Reduction R_b (pure transition)

The reduction rules described above can be extended to and expressed as an efficient algorithm [Martinez and Silva 82] to find all the invariants of a Petri net. These are implemented in Chapter 5 of this Thesis.

Using a combination of these reductions it is possible to obtain the invariants of ordinary Petri nets. This is illustrated in Fig. 2.17. Starting with a 5-place, 4-transition Petri net we apply reduction rule R_b three times and R_a once, resulting in a 2-place Petri net from which we can deduce that P1+P2+P4 = 1 and P1+P3+P5 = 2. This means that the number of tokens in the loop P1,P2,P4 is equal to one and that the number of tokens in the loop P1, P3 and P5 is equal to two at all times.

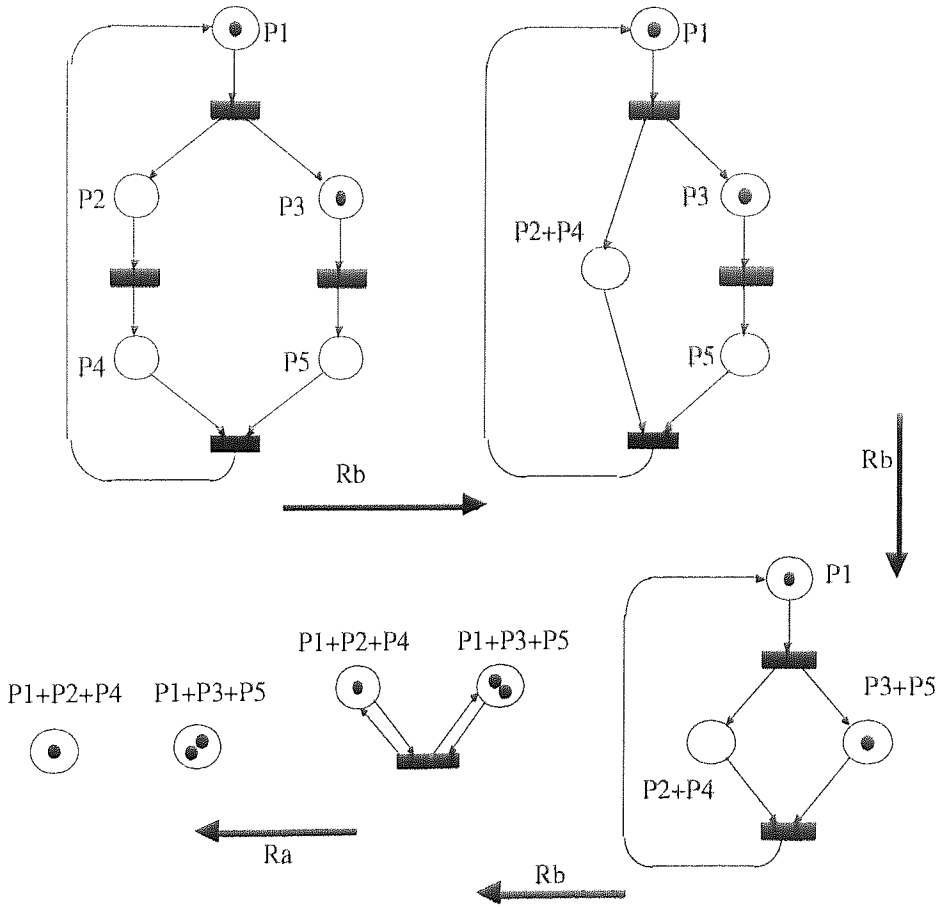


Fig. 2.17 Calculation of P-Invariants [David and Alla 92]

2.3.5 Reduction rules

Petri net models of large systems tend to become very complex. Generating the coverability graph and calculating P- and T- invariants using matrix techniques can become very time consuming and impossible to compute manually.

For this reason, several researchers, including Hack [72], Bertholet [86], Lee-Kwang *et al.* [87], Murata and Komoda [87] have devised reduction rules that, whilst reducing the number of places and transitions in the Petri net, preserve certain properties of the original Petri net. These can then be verified by using coverability graph analysis on the reduced Petri net. The following sections describe reduction rules that preserve liveness, quasi-liveness, home-state, conservativeness and boundedness.

2.3.5.1 R1 - Substitution of a place

A place P_i can be substituted (suppressed) if:

1. The output transitions of P_i have no input places other than P_i.
2. Place P_i is pure (P_i is pure if there is no transition that is both an input and an output transition to P_i).
3. At least one output transition is not a sink transition (a sink transition is one that has no output places).

Substitution is done by removing the place and its output transition as shown in Fig. 2.18a and Fig. 2.18b below. In the case that the place to be substituted is tokenised, then the token is placed in the place following the output place following the output transition.

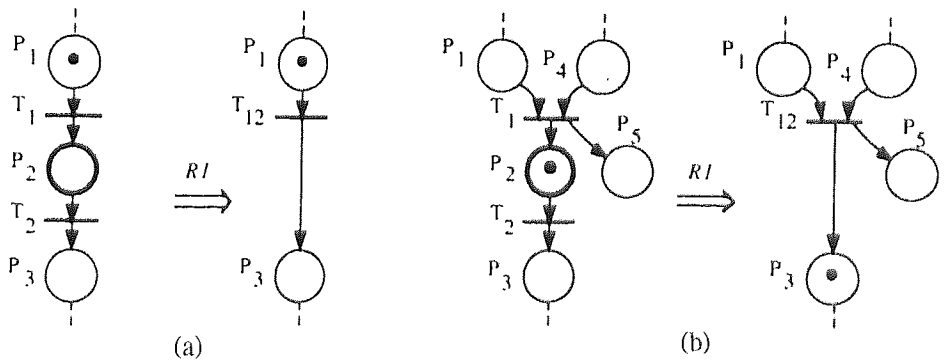


Fig. 2.18 Reduction Rule R1 [David and Alla 92]

In the case that the place to be substituted has more than one input transition or more than one output transition, then, substitution is done by removing the place and re-structuring the links specified by the directed arcs connected to this place as shown in Fig. 2.19 and Fig. 2.20.

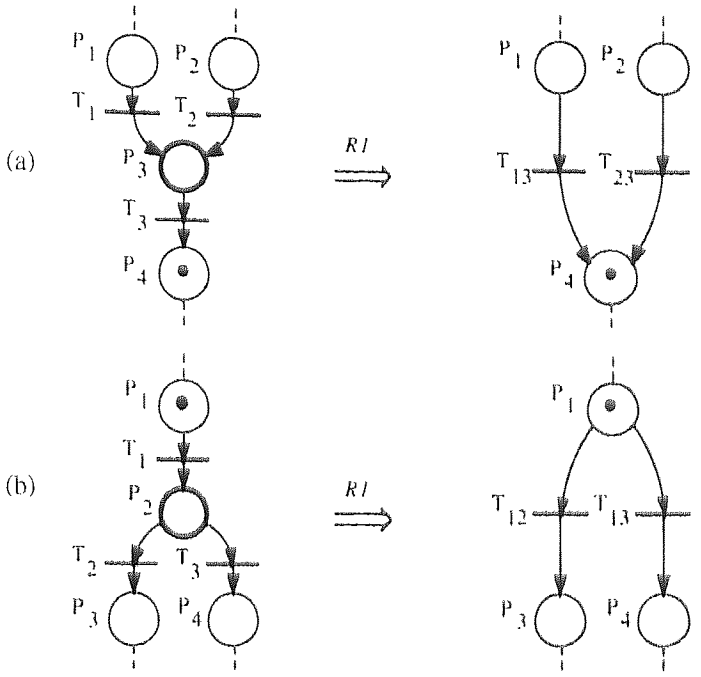


Fig. 2.19 Substitution of a 2-input or 2-output place [David and Alla 92]

2.3.5.2 R2 - Neutral Transition

A transition T_j is neutral iff ${}^{o}T_j = T_j^{o}$. A neutral transition and its interconnecting arcs can be suppressed iff a transition $T_k \neq T_j$ exists such that $\text{post}(p_i, t_k) \geq \text{pre}(p_i, t_j)$ for every place $p_i \in {}^{\circ}T_j$ [David and Alla 92]. This is illustrated in Fig. 2.21.

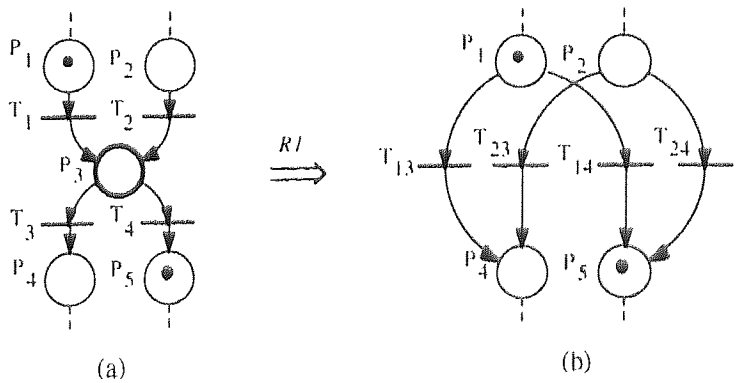


Fig. 2.20 Substitution of a place with more than one input and output [David and Alla 92]

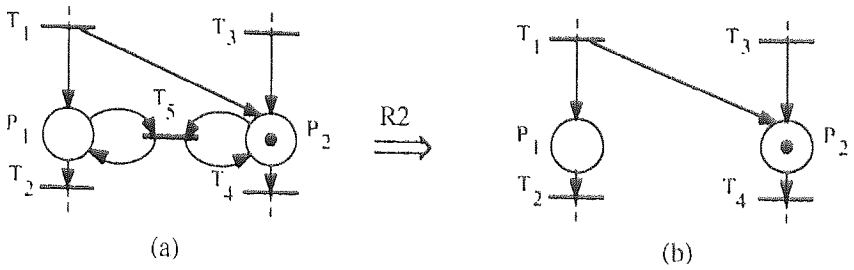


Fig. 2.21 Reduction rule R2 [David and Alla 92]

2.3.5.3 R3 - Identical transition

Transitions T_i and T_j are identical iff ${}^{o}T_i = {}^{o}T_j$ and $T_i^{o} = T_j^{o}$. In such a case one of the transitions together with its corresponding arcs can be suppressed. This is illustrated in Fig. 2.22 below.

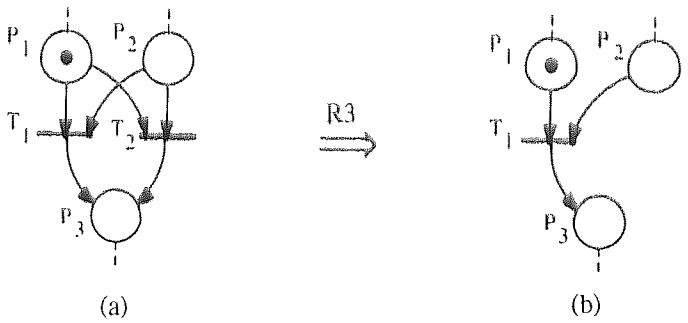


Fig. 2.22 Reduction rule R3 [David and Alla 92]

2.3.6 Petri net extensions

In the following sections, Petri net extensions that have been developed to suit particular applications relevant to this research are described.

2.3.6.1 Timed Petri nets

To be able to analyse how a DEDS evolves in time, it is essential to include the concept of time within the Petri net model. Timed Petri nets are an extension of Petri nets that are able to incorporate time within the model. There are two conventions related to timed Petri nets, one has a time delay associated with the places in the Petri net and the other has time delay associated with the transitions. Sifakis [78] showed that they can be functionally equivalent, however, both conventions have been widely used.

Dubois and Stecke [83] were the first to use Petri nets to analyse control problems of production systems by using timed transition Petri nets. Barad and Sipper [88] used timed place Petri nets to model a FMS. A timed place Petri net has two main advantages over a timed transition Petri net, namely: (a) it preserves the usual Petri net convention of instantaneous events; (b) it is less ambiguous, since the marking of the Petri net during the time that a process is in execution, represents the current state of the system.

A timed place Petri net is defined as a six-tuple, $Z_{timed} = (P, T, I, O, \theta, m)$ where P, T, I, O are as previously defined for an ordinary Petri net and θ: $P \rightarrow \Re^+$ is a delay vector, whose *i*th element represents the time associated with the *i*th place. A transition $t \in T$ is enabled iff $\forall p \in I(t)$, $m(p) > 0$ for a time, $\delta \geq \theta(p)$.

2.3.6.2 Controlled Petri nets

Krogh [87] developed controlled Petri nets (CPNs) for the solution of a class of control problems as described in [Holloway and Krogh 90]. A CPN is a five-tuple $Z_C = \{P, T, \xi, \chi, \beta\}$, where P is the set of places, T is the set of transitions, $\xi = (P \times T) \cup (T \times P)$ is the set of directed arcs connecting places and transitions, χ is the finite set of control places, represented by empty squares and β is the set of arcs associating control places with transitions. A place $p \in P$ or control place $c \in \chi$ is said to be an input to a transition $t \in T$ if $(p,t) \in \xi$ or $(c,t) \in \beta$, respectively. The set of places or control places which are inputs to a transition $t \in T$ is denoted by ${}^{(p)}t$ or ${}^{(c)}t$. The set of controlled transitions $T_C \subseteq T$ is defined as the set of transitions $t \in T$ for which ${}^{(c)}t \neq \varnothing$. The set of transitions that are not in T_C is called the set of uncontrolled transitions. A control $u: \chi \rightarrow \{0,1\}$ assigns a binary token count to each control place. A transition $t \in T$ is said to be state enabled under a marking m if $\forall p \in {}^{(p)}t, m(p) \geq 1$. A controlled transition $t \in T_C$ is said to be control enabled under a control u if $u(c) = 1, \forall c \in {}^{(c)}t$.
The control place $c \in \chi$ is said to be disabling if u(c)=0, and is said to be enabling if u(c)=1. A controlled transition $t \in T_C$ is enabled if it is both state enabled and control enabled.

2.3.6.3 High level Petri nets

High level Petri nets, including predicate/transition nets [Genrich and Lautenbach 81], coloured Petri nets [Jensen 81] and nets with individual tokens [Reisig 82] were developed to model complex DEDS. They can be considered to be a structurally folded version of a regular Petri net and produce a smaller and more manageable graphical representation for very complex systems [Murata 89]. To illustrate the transition firing rule for high level Petri nets, consider the net illustrated in Fig. 2.23 (a). It consists of one transition, four places and four labelled arcs. The arc label dictates how many and which kinds of coloured tokens will be removed from or added to a place on firing of the

relevant transition. For example, when the transition in the net of Fig. 2.23 (a) fires, P_1 loses two tokens of the same colour, x, P_2 loses two tokens of different colours, <x,y> and <y,z>, P_3 gets one token of the colour <x,z> and P_4 gets one token of the colour e (a constant). This is illustrated in Fig. 2.23 (b).

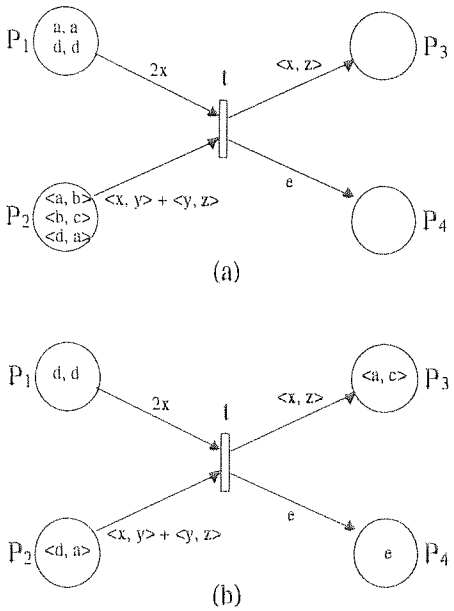


Fig. 2.23 Illustrating the firing rule for high level Petri nets

2.3.6.4 Continuous Petri nets

Timed Petri nets [2.3.6.1] are well suited for the quantitative evolution of a system. However, in the case where the times associated with places differ greatly in length, then, the number of reachable states explodes [Alla 95]. This limitation can be overcome by the use of timed continuous Petri nets [David and Alla 90], which are presented informally, below.

In a continuous Petri net, the marking of a place is a real number and, at each time, an associated firing speed v(t) is associated with each transition. This transition is continuously fired, which means that between the instant t and $t+\delta t$, a quantity of marking, $v\delta t$ is removed from the input place and added to the output place of this transition. Continuous places are represented as two concentric circles and transitions are represented as rectangles. To illustrate the application of continuous Petri nets, consider the simple production line, consisting of two buffers and a machine, illustrated in Fig. 2.24 (a). It is represented by a timed transition Petri net as shown in Fig. 2.24 (b). In this model the delay, d=0.5, represents the processing time of Machine 1, the availability of which is represented by place P_3.

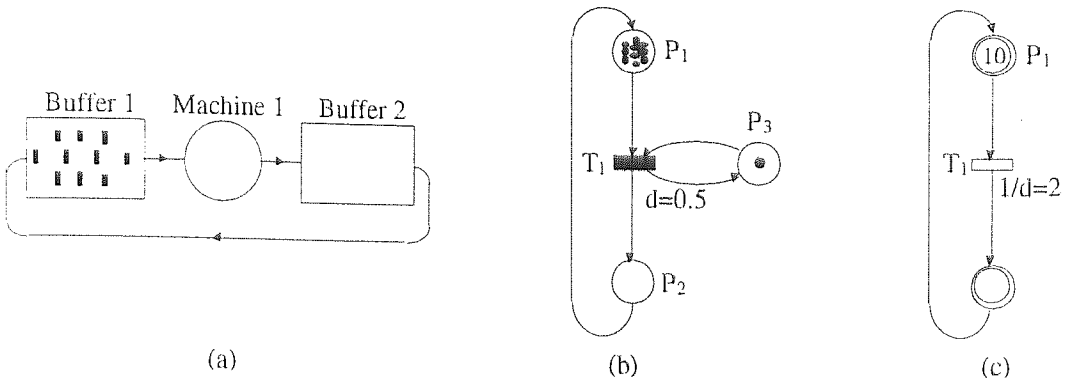


Fig. 2.24 Continuous Petri nets

The discrete event model (Fig. 2.24 (b)) is transformed into a continuous Petri net in the following way: The integer number of tokens in P_1 is replaced by a real number and a firing speed, v=1/d=2, is associated with transition T_1. The evolution of the continuous Petri net is as described above.

2.3.6.5 Hybrid Petri nets

As defined in Section 1.2, DEDSs evolve from one state to the next via instantaneous events. The states that a DEDS may be in could be of a continuous nature, such as, "drum is rotating". Such systems are commonly referred to as hybrid systems. Therefore, Petri nets have been extended to combine the concepts of Petri nets and continuous Petri nets to form hybrid Petri nets [Le Bail *et al.* 91]. Consider the example in Fig. 2.25, which represents a system consisting of two tanks (continuous places P_1 and P_2) and a valve (continuous transition T_1) which can be open (P_3 is marked) or closed (P_4 is marked). When P_3 is marked and Tank 1 is not empty, transition T_1 fires continuously at a speed that represents the fluid flow. When the marking of the continuous place P_2 is equal to 85.6, indicating that Tank 2 is full, then T_3 is enabled and fires, thus closing the valve. This example shows how the interaction of the continuous and discrete event parts of a hybrid system can be modelled on a hybrid Petri net.

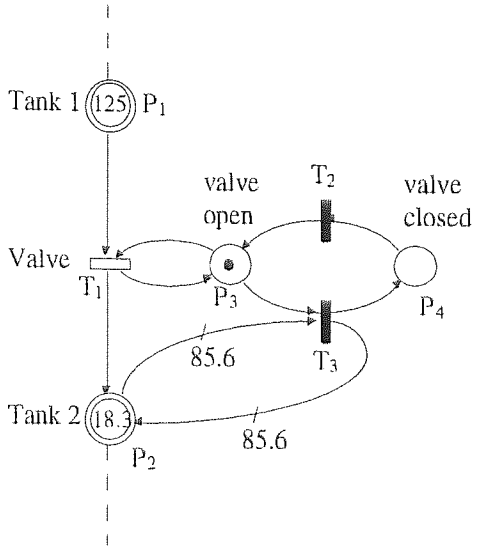


Fig. 2.25 Hybrid Petri nets

2.4 Conclusion

This Chapter discussed techniques for modelling DEDSs and selected Petri net theory as the modelling technique that is most suited to the research presented in this Thesis. It described Petri net theory for analysis of structural and behavioural properties of DEDSs by means of three main techniques, including coverability graph analysis, incidence matrix analysis and analysis by means of reduction rules.

The following Chapter will investigate currently available Petri net synthesis techniques that are used for modelling large and complex DEDSs and will introduce a novel object oriented methodology for Petri net synthesis based on OMT [Rumbaugh 91].

Chapter 3

Synthesis of DEDS models

3.1 Introduction

Modelling practical DEDSs such as independently-driven, multi-axis, high-speed industrial machines or complex manufacturing systems, usually results in a Petri net with a large number of places and transitions [Desrochers and Al-Jaar 95]. This implies a large state space and incidence matrix, which makes it computationally difficult to check for liveness, boundedness and reversibility using the conventional Petri net analysis techniques described in Chapter 2. To overcome this problem, design techniques have been developed to synthesise well behaved Petri net models of complex systems. These include bottom up [Agerwala and Choed-Amphai 78], top down [Valette 79], and hybrid [Zhou and DiCesare 93] Petri net synthesis techniques.

Since a Petri net model is an abstract representation of the system, there is a 'semantic gap' [Cooling 91, Fraser *et al.* 91] between the informal specification of a system and its model. Researchers including Yourdon and Constantine [79], DeMarco [78], Jackson [83], Ward and Mellor [85] have developed structured methods to bridge the 'semantic gap' between the informal specification of a system and its model. However, these methods concentrate on functional abstraction, and have produced incomplete specifications and designs [Firesmith, 93]. In order to facilitate the design of complex systems, produce more understandable designs and specifications, facilitate the transition between design and implementation and enable software re-use, several researchers including Booch [91], Rumbaugh *et al.* [91], Shlaer and Mellor [92], Firesmith [93] have advocated a paradigm shift towards object oriented (OO) techniques.

This chapter reviews Petri net synthesis techniques and object modelling technique (OMT) [Rumbaugh et. al 91], a well established object oriented methodology. It presents a modification to OMT to facilitate the design of complex DEDSs and to improve the representation and analysis of the dynamic model of the system. This modification enables the construction of a complete DEDS model by following a step by step approach.

3.2 Bottom up Petri net synthesis

In bottom up synthesis one identifies the low-level capabilities the system needs to have, then identifies the common aspects of low level components (sub-systems) and groups them to form a larger sub-system. This process is then repeated on the next level up until the system is completely described.

In bottom up Petri net synthesis techniques [Agerwala and Choed-Amphai 78, Narahari and Viswanadham 85, Krogh and Beck 86], sub-systems are modelled separately. The sub-system models are normally small and easy to verify. At each step of the synthesis procedure, the interactions between the sub-systems are considered and represented by merging common places/transitions of the sub-system models, resulting in a larger sub-system. Analysis for the required properties is done after each step so that final analysis is simplified.

Agerwala and Choed-Amphai [78] presented a set of synthesis rules which allowed the bottom up construction of large Petri nets, based on smaller sub-nets that shared a common place. At each synthesis step, sub-nets are merged in such a way that a set of places are merged into a new place. This is called a one-way merge, and is defined below:

For an unmarked Petri net, Z=(P,T,I,O), select a set of places to be merged $P_m \subseteq P$, such that:

1. If p_i and p_j are input places to the same transition then they cannot be merged. Therefore $\forall p_i, p_j \in P_m, \left(I(p_i,t)=1 \wedge I(p_j,t)=1\right) \Rightarrow p_i = p_j$.
2. If p_i and p_j are output places to the same transition then they cannot be merged. Therefore $\forall p_i, p_j \in P_m, \left(O(p_i,t)=1 \wedge O(p_j,t)=1\right) \Rightarrow p_i = p_j$.

Then construct the net Z'=(P',T',I',O'), such that

1. $T' = T$
2. $P' = (P - P_m) \cup \{p\}$, where $p \notin P$
3. I' and O' are obtained by replacing every occurrence of each $p_i \in P_m$ in I and O by p

A one-way merge is illustrated in Fig. 3.1 where P3 and P6 are merged into P3.

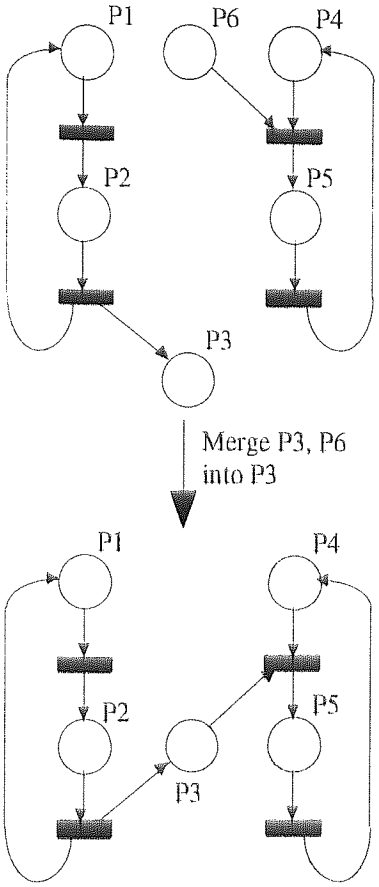


Fig. 3.1 Example of a one-way merge [Jeng and DiCesare 93]

Agerwala and Choed-Amphai [78] presented the following theorem which shows that after every one-way merge, the P-invariants of the resultant net can be derived from the P-invariants of the sub-nets.

Theorem

Consider a Petri net Z=(P,T,I,O) on which the one-way merge operation is applied, resulting in the Petri net Z'=(P',T',I',O'), all the P-invariants [2.3.4.1] of Z', PI', are obtained from the P-invariants of Z, PI, as follows:

PI' $\subseteq$ P is a P-invariant of Z' iff $\exists$PI on Z such that:

1. if $P_m \subseteq PI$, then $PI' = (PI - P_m) \cup \{p\}$
2. if $P_m \cap PI = \varnothing$, then $PI' = PI$

Consider the Petri net, Z, (Fig. 3.1) consisting of places {P1, P2, P3, P4, P5, P6}. The P-invariant supports of Z, PI = ({P1,P2}, {P4,P5}). Z' is formed by merging P3 and P6 into P3, therefore P_m = {P3, P6}, satisfying the condition: $P_m \cap PI = \varnothing$. Then the P-invariant supports of Z', PI' = PI = ({P1,P2}, {P4,P5}). This can be verified by solving the equations of Definition 2.17 in [2.3.4.1]. In this particular case, since not all the places are included in the set of P-invariant supports, one can not use the invariant method to verify liveness or boundedness.

Narahari and Viswanadham [85] presented a systematic bottom up approach to synthesise the Petri net model of a FMS. They obtained the final model by representing every machine operation by a separate Petri net and then combining these nets by sharing places. They extended the theorems of Agerwala and Choed-Amphai [78] to verify for existence/absence of deadlock (liveness), conservativeness and boundedness after each synthesis step.

Krogh and Beck [86], proposed a bottom up synthesis approach for synthesising live and 1-bounded Petri nets. Their method, unlike the bottom up approaches of Agerwala and Choed-Amphai [78] and Narahari and Viswanadham [85], is based upon sharing simple elementary paths [Berge 62] in which no place or transition appears more than once. They defined two types of simple elementary paths:

1. A solitary transition path (STP), which is a simple elementary path terminated at both ends by a place, for which each transition in the path has only one input and one output place.
2. A solitary place path (SPP), which is a simple elementary path terminated at both ends by a transition, for which each place in the path is an input place for only one transition and an output place for only one transition.

To illustrate the synthesis procedure, consider the three simple elementary circuits (SECs) in Fig. 3.2, where an SEC is a Petri net of finite length with coincident initial and final places. First (a) and (c) are combined along a common STP {P1, t1, P2}, then the resultant net is combined with (b) along a common SPP {t2, P3, t3}, resulting in the Petri net (d). The Petri net obtained in this fashion will be live and 1-bounded if there is exactly one token in each of the P-Invariants of the system (a standard invariant analysis result [David and Alla 92]). Krogh and Beck [86] show that after each synthesis step, the P-invariants of the combined net can be easily calculated.

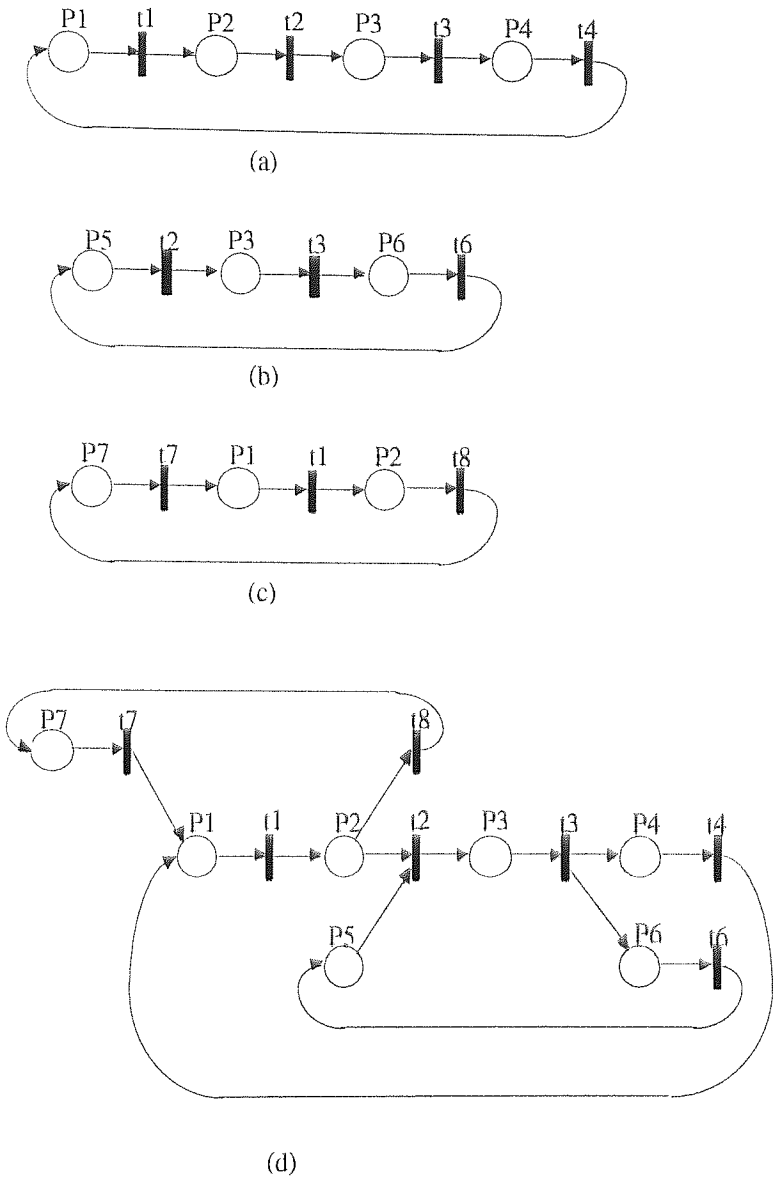


Fig. 3.2 (a)-(c) Three SECs, (d) Three SEC's combined along an STP and an SPP [Jeng and DiCesare 93]

The bottom up Petri net synthesis techniques described above rely on the invariant method to analyse the properties of the combined net after each synthesis step. The disadvantage of using invariant analysis is that P-invariants do not convey complete information about the Petri net, making it difficult to analyse the net for properties such as liveness and reversibility. Except for the method of Krogh and Beck [86], the methods do not guarantee that the resultant net will preserve important properties.

Bottom up design has the advantage of describing the system in terms of sub-systems, which have real-life correspondences such as robots and machines. However, it is difficult to use exclusively since most people are better at breaking down a large concept into smaller ones [McConnell 93].

Also, it is not possible to build a system solely by using building blocks, without actually knowing what the final product will look like. Therefore it can only be used in conjunction with top down design which shall be described in the following section.

3.3 Top down Petri net synthesis

Top down design [Wirth, 71] is a technique for design of a system by moving from a general statement of what the system does to detailed statements about specific tasks that are performed. This approach is often referred to as a "divide and conquer" approach. In top down Petri net synthesis, one starts off with a simple Petri net, showing the top-level behaviour of the system. Then, transitions and places are replaced by a more detailed subnet in a step-by-step fashion, so that a large Petri net can be synthesised. This method is also referred to as stepwise refinement of transitions and places [Valette 79, Suzuki and Murata 82, 83]. Valette [79] showed that using well formed blocks [3.3.1] to replace a place or transition in a safe, live and reversible Petri net preserves these properties in the larger net.

Top down Petri net synthesis has the limitation, however, of not guaranteeing the correct behaviour of a concurrent system if shared resources are involved. The reason is that in these systems the interactions among the sub-systems are coupled throughout all levels of refinement, which makes it difficult to specify the system using the top down approach [Jeng and DiCesare 93]. This has led to the development of a hybrid synthesis approach [Zhou and DiCesare 93], which overcomes this deficiency.

3.3.1 Well formed blocks

A Petri net with one initial transition labelled t_{ini} and one final transition labelled t_{fin} is called a block. Consider the Petri net Z' = (P', T', I', O', m_o') (Fig. 3.3). It is obtained from a block Z = (P, T, I, O, m_o), by adding a place p_o, called an idle place, such that:

- The only output transition of p_o is t_{ini},
- The only input transition of p_o is t_{fin},
- $m_o' = m_o + \{ p_o \}$.

Then, the following definitions can be given:

- The block Z is k-bounded iff the associated Petri net Z' is k-bounded,
- The block Z is live iff the associated Petri net Z' is live.

A block Z is said to be well formed [Valette 79] iff the associated Petri net Z' is such that:

- Z' is live
- m_o' is the only marking in the set of reachable markings from m_o' such that the idle place is not empty
- The only transition enabled by m_o' is the initial transition, t_{ini}.

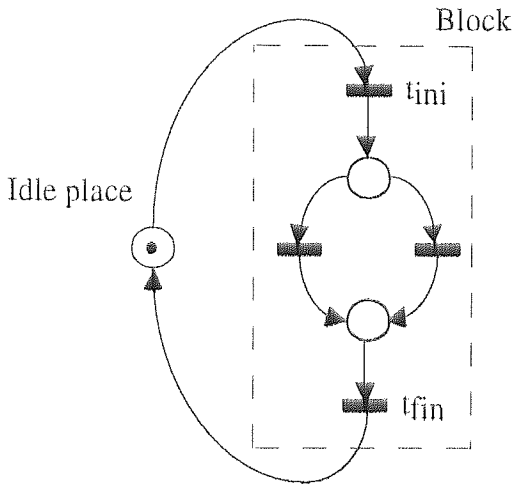


Fig. 3.3 Well formed block with idle place

Examples of well formed blocks are illustrated in Fig. 3.4 below.

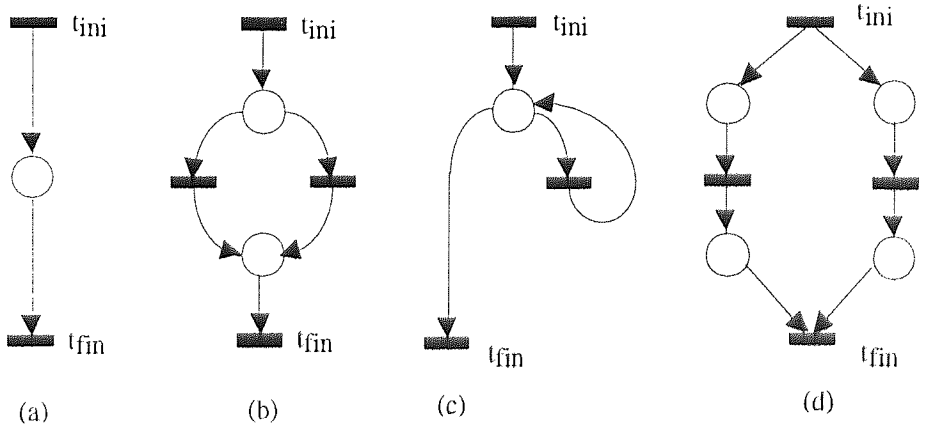


Fig. 3.4 Well formed blocks: (a) "Sequence" block; (b) "if-then-else" block; (c) "do-while" block; (d) "fork-join" block [Valette 79]

3.4 Hybrid Petri net synthesis

Hybrid Petri net synthesis [Zhou and DiCesare 93] uses top down design followed by a bottom up approach. This synthesis approach, although very similar to top down design, guarantees a safe, live and reversible Petri net model when shared resources are involved.

This is achieved by modelling shared resources as parallel mutual exclusion (PME) places [Defn. 2.14] and sequential mutual exclusion (SME) places [Defn. 2.15]. Hybrid Petri net synthesis of Zhou and DiCesare [93] is summarised below;

- From knowledge of the system, an abstract Petri net model is formulated to describe the important system interactions. This initial net should have all the desirable properties required for the final net.

- Stepwise refinement is used to add the required amount of detail. Each successive refinement will retain the desirable properties of the initial net if well-formed blocks [Valette 79] are used.

- Resource places and the shared resource places are added in the form of PME [Defn. 2.14] and SME [Defn. 2.15] places as required.

This approach inherits the advantages and disadvantages of top down design. Its main weakness is that formulating the top function of a system, which is the first step of the hybrid synthesis technique, is difficult in realistically sized systems. Another disadvantage is that many systems are not naturally hierarchical, so they are difficult to decompose and represent by means of well formed blocks. The most serious weakness is that this technique requires a system to be described by a single Petri net at the top, which is a dubious requirement for many modern event-driven systems [McConnell 93].

3.5 Bridging the semantic gap

The previous sections described methods for synthesis of well behaved Petri nets. Using these methods one can synthesise Petri nets which would exhibit the required properties of liveness, boundedness and reversibility, therefore not requiring time consuming analysis. However, since a Petri net is an abstract representation of the system, there is a semantic gap (defined in [Cooling 91, Fraser *et al.* 91]) between the informal specification of a system and its model. Thus it is possible for information to be lost in the process of translation of the informal requirements into a Petri net representation.

When synthesising Petri net models of complex industrial systems or safety critical systems, designers need to feel confident that their model accurately represents the system. To help bridge the semantic gap, researchers (including Yourdon and Constantine [79], DeMarco [78], Jackson [83], Ward and Mellor [85], Yourdon [89]) have developed structured methods . However, according to Firesmith [93], since these

methodologies concentrate mainly on functional abstraction they have produced incomplete specifications and designs because, as noted by Seidewitz [89] "Functional analysis and specification techniques actually sacrifice closeness to the problem domain in order to allow a smooth transition to functional design methods". Functional decomposition methods have very limited software re-use [Firesmith 93] and, since changes in requirements are mostly related to functions rather than objects, small changes to functional decomposition designs may have a large ripple effect [Rumbaugh *et al.*, 91]. To enable the design of more complex software systems, to enable re-use of designs, to produce more understandable designs and specifications, researchers (including Booch [86, 91], Meyer [88], Shlaer and Mellor [88], Coad and Yourdon [90], Jacobsen [87], Rumbaugh *et al.* [91] and Firesmith [93]) have advocated a paradigm shift towards object oriented methodologies.

Booch [86] described the fundamental concepts of object oriented software development and explained that object oriented development is fundamentally different from functional approaches to design. Object oriented software decomposition more closely models a person's perception of reality, whilst functional decomposition is only achieved through a transformation of the problem space. Therefore, a design that is developed using an object oriented engineering approach is more understandable, extensible and maintainable. In OO techniques [Booch 86], one refers to classes of objects. A class of objects is defined as a collection of attributes and operations that are able to manipulate the values of these attributes. To enable re-use of designs, one can define a class, starting off from another class, by means of inheritance. The inherited class (subclass) contains all the attributes and operations of the superclass together with additional attributes and operations.

An object is an instance of a class and the values of the object's attributes can only be changed by executing the operations defined in the class it belongs to. Encapsulation of properties and operations within an object makes it possible to treat the object as a 'black box' and prevents small changes to an object from having a large ripple effect on the whole system.

Object-Oriented Analysis (OOA) strives to understand and model a particular problem from a user-oriented or expert's perspective, in terms of objects and classes, with an emphasis on modelling the real-world. The product, or resultant model, of OOA specifies a complete system, a complete set of requirements and an external interface of the system to be built. These are often obtained from a domain model (e.g. FUSION, [Jacobsen 87]), scenarios [Rumbaugh *et al.* 91], or use-cases [Jacobsen 87].

3.6 Comparison of OO methodologies

Rumbaugh *et al.* [91] defined a software engineering methodology as a process for the organised production of software, using a collection of techniques and notational conventions. A methodology is usually presented as a series of steps, with techniques and notation associated with each step. The more established OO methodologies, are those of Booch [86], Shlaer and Mellor [88] and Rumbaugh *et al.* [91] and are briefly introduced below.

Booch [91] extended previous Ada-oriented work to the entire OO design area. Booch's methodology includes models to describe the object, dynamic and functional aspects of a software system. Shlaer and Mellor [88] described a complete methodology for object-oriented analysis which breaks down analysis into three phases: Static modelling of objects, dynamic modelling of states and events, and functional modelling. Shlaer and Mellor [88] added OO techniques to the traditional structured analysis principles of Yourdon and Constantine [79]. Yourdon [89] provided a critique, but only referred to their earlier work. According to Rumbaugh *et al.* [91] a major flaw with the Shlaer and Mellor methodology is the excessive preoccupation with relational databases and database keys. The OMT [Rumbaugh *et al.* 91] methodology consists of three phases: analysis, system design and object design. An object model is augmented with a dynamic model and a functional model to describe all aspects of the system. The OMT analysis phase is the development of a model of what the system is supposed to do, whilst the design phase consists of optimising and refining the object model, dynamic model and functional model until they are detailed enough for implementation.

Fundamentally, the methodologies of Booch [86, 91], Shlaer and Mellor [88] and Rumbaugh *et al.* [91] are very similar, the main difference being the graphical notation that they use. In fact, currently[1] there is a move to combine OMT and Booch methodology to form a "unified approach". The Shlaer and Mellor methodology is an approach to analysis, whereas OMT is an object oriented approach spanning from analysis to design and implementation, and for this reason it was chosen to be the object oriented methodology to be adopted in this Thesis.

[1] Object Expo, London, UK, September '95

3.7 The OMT methodology

OMT uses three models to describe a system: the object model, describing the objects in the system and their relationship; the dynamic model, describing the interactions among objects in the system; and the functional model, describing the data transformations of the system. The methodology is introduced in the following sections.

3.7.1 The object model

The OMT object model consists of object diagrams which describe the static structure of objects in the system, including their identity, their relationships to other objects, their attributes and operations that can change the values of these attributes. The object model provides the foundation for the dynamic and functional models. OMT object modelling notation combines object-oriented concepts (class and inheritance) with information modelling concepts (entities and associations). It is an enhanced form of entity-relationship (ER) diagram [Chen 76] that is used in information modelling and database design. The basic notation is described in the following sections and a more complete description can be found in [Rumbaugh *et al.* 91].

3.7.1.1 Modelling objects and classes

Figure 3.5 summarises the object modelling notation for classes. A class is represented by a box with three regions. The topmost region contains the name of the class. The second region contains a list of attributes, with optional additional information including their data type (e.g. integer, Boolean, real number, etc.) and initial value. The bottom region contains the list of operations that can modify the values of the attributes. Each operation may be followed by additional details such as an argument list and return type.

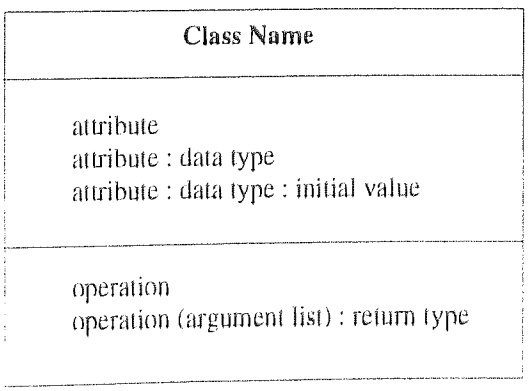


Fig. 3.5 Object modelling notation for classes

3.7.1.2 Modelling links and associations

Links and associations establish relationships among objects and classes. An association describes a group of links with a common structure. A link, which is an instance of an association, is a physical or conceptual connection between object instances. Associations and links often appear as verbs in the problem statement. A one-to-one association between classes are represented as shown in Fig. 3.6.

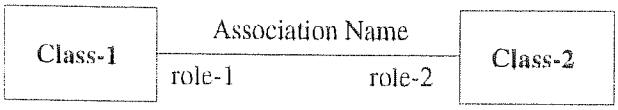


Fig. 3.6 One-to-one association

Multiplicity of associations specifies how many instances of one class may relate to a single instance of an associated class. The terminology used in OMT to model multiple associations is shown in Fig. 3.7.

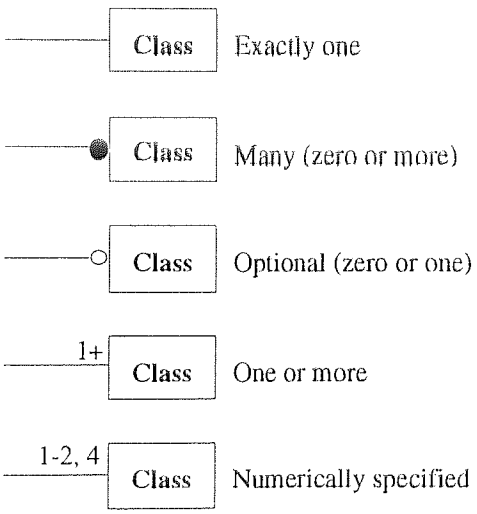


Fig. 3.7 Illustrating multiple associations

3.7.1.3 Modelling inheritance

Inheritance is a method of abstraction for sharing similarities among classes whilst preserving their differences. The notation for inheritance is a triangle connecting a superclass to its subclasses as shown in Fig. 3.8.

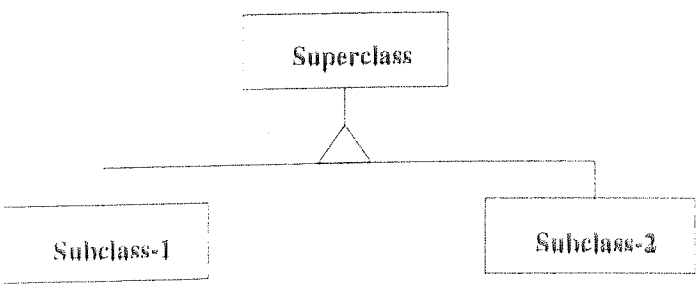


Fig. 3.8 Illustrating inheritance

3.7.2 The dynamic model

The OMT dynamic model describes the aspects of a system that are concerned with time and the sequencing of operations. These include sequences of events, states that define the context of events and the organisation of states and events. OMT dynamic models are represented graphically by means of state diagrams [Section 2.2.1], showing the state and event sequences that can occur in every class. The state diagrams used in OMT have been extended to enable nested state diagrams (for more compact notation) and to model synchronisation and concurrency as in statecharts [Section 2.2.1.1]. These extensions are described in the following sections and a more complete description can be found in [Rumbaugh *et al.* 91].

3.7.2.1 Nesting state diagrams

To allow for a more compact representation of state diagrams, the OMT notation uses nested state diagrams. Thus, an activity in a state can be expanded as a lower-level state diagram, with each state representing one step of the activity. The notation for nesting state diagrams is shown in Fig. 3.9 where the superstate consists of substate-1 followed by substate-2.

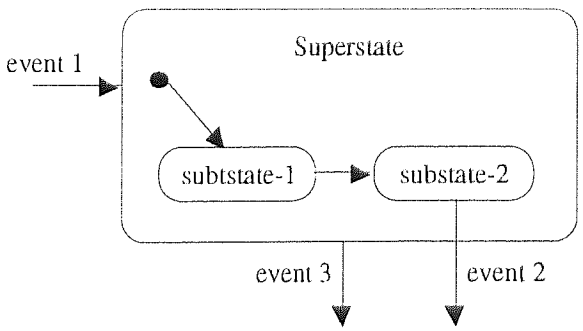


Fig. 3.9 Nesting state diagrams

3.7.2.2 Concurrency

OMT state diagrams have been extended to model concurrency. Concurrency within an object can occur when the object can be partitioned into subsets of attributes or links, each of which has its own subdiagram. This is illustrated in Fig. 3.10 where the superstate consists of four substates which are partitioned into two concurrent subdiagrams.

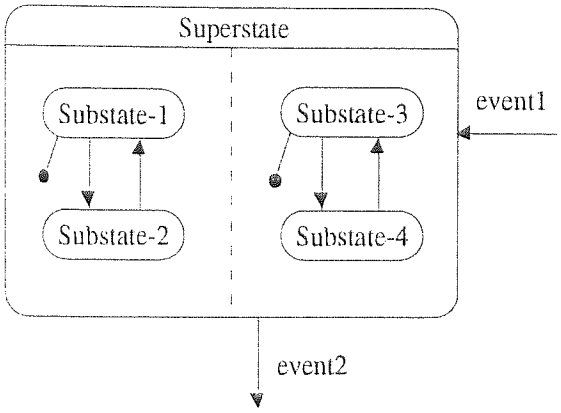


Fig. 3.10 Concurrent state diagrams

3.7.2.3 Synchronisation

In OMT, the synchronisation of events is shown by an arrow with a forked tail as shown in Fig. 3.11, where the concurrent events, event3 and event4, are synchronised events resulting from concurrent processes.

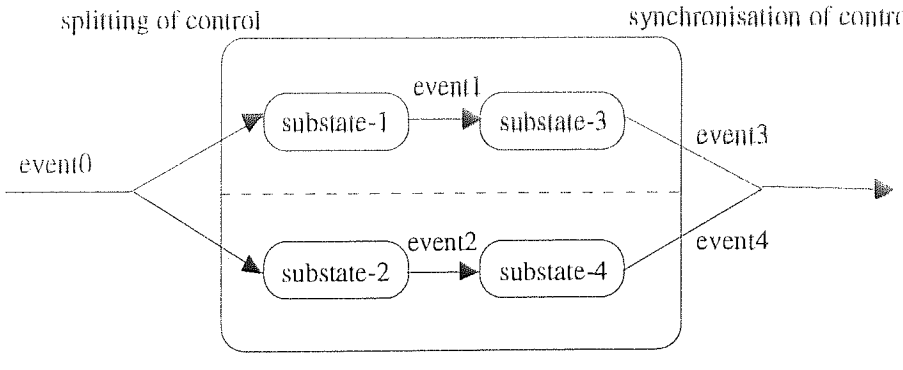


Fig. 3.11 Synchronisation of control

3.7.3 The functional model

The functional model shows how values are computed, identifies constraints between objects and specifies the optimisation criteria. Functional dependencies are illustrated by means of data flow diagrams [Rumbaugh *et al.* 91] and functions are expressed in various ways including natural language, mathematical equations and pseudo code. The processes on a data flow diagram correspond to states defined in the state diagrams. The flows on a data flow diagram correspond to objects or attribute values defined in an object diagram.

3.7.4 Analysis

In the analysis stage, the OMT designer is concerned with understanding and modelling the system and the domain in which it operates. The first step of the analysis phase consists of obtaining a problem statement which provides a conceptual overview of the proposed system. This, together with dialogue with the design engineers and background knowledge, is used to obtain a formal model of the three essential aspects of the system: the objects and their relationships, the dynamic flow of control, and the functional transformation of data. Rumbaugh *et al.* [91] give a step by step guide for the analysis stage, resulting in an analysis document, consisting of a problem statement, an object model, a dynamic model and a functional model. The steps are summarised below:

3.7.4.1 Write or obtain an initial description of the problem.

The first step in the design of a DEDS is to state the requirements. The typical contents of a problem statement are the problem scope, application context, assumptions, performance and safety requirements.

3.7.4.2 Build the object model

To build an object model, one first identifies the classes of objects that are specified in the problem statement or that are implicit in the application domain. Objects include physical entities (e.g. machine) and concepts (e.g. trajectory) which often correspond to nouns in the problem statement. The next step is to identify associations between the classes. Associations are dependencies between two or more classes and often correspond to stative verbs or verb phrases in the problem statement (e.g. communicates with, inserts into, drives). This step is followed by identifying the object attributes for each class of objects. Attributes are properties of individual objects (e.g. name, velocity, position) and usually correspond to nouns followed by possessive phrases (e.g. "the position of the cursor") or adjectives (e.g. stationary, rotating). The next step is to organise classes by using inheritance to share common structures and to test the access paths through the object model diagram. Finally, since the object model is rarely correct after a single pass, it is usually necessary to go through the object modelling procedure again.

3.7.4.3 Develop the dynamic model

The dynamic model is more important for interactive systems than for static systems (e.g. databases). The first stage in developing the dynamic model is to prepare scenarios of typical interaction sequences that occur between the different objects in the system and to identify external events. The scenarios are represented as event traces on an event flow diagram.

The next stage is to develop a state diagram for each class that has important dynamic behaviour, where every scenario corresponds to a path through the state diagram. The final stage is to check for consistency and completeness of events shared among the state diagrams.

3.7.4.4 Construct the functional model

The procedure for constructing the functional model consists of identifying input and output values, and using data flow diagrams to show functional dependencies. The next step is to describe each function in natural language, mathematical equations, pseudo code or some other appropriate form. Finally, the constraints between the objects are identified and the optimisation criteria are specified.

3.7.4.5 Verify, iterate and refine the three models

According to Rumbaugh *et al.* [91]. most analysis models require more than one pass to complete so it is necessary to verify, iterate and refine the three models to remove inconsistencies within and across the models. Thus the OMT analysis document consists of a problem statement, an object model, a dynamic model and a functional model.

3.7.5 System design

In the system design phase, the overall architecture of the system is determined. Using the object model as a guide, the system is organised into sub-systems. Concurrency is organised by grouping objects into concurrent tasks and decisions are made about inter-process communication, data storage and implementation of the dynamic model. Also, priorities are established for making design trade-offs. Rumbaugh *et al.* [91] give a step by step guide for the system design stage, resulting in a system design document which consists of the structure of the basic architecture for the system and the high level strategy decisions.

3.7.6 Object design

In the object design phase, the analysis models are elaborated, refined and then optimised to produce a practical design. During the object design phase, there is a shift in emphasis from design to implementation. First, the basic algorithms are chosen to implement each major function of the system. Based on these algorithms, the structure of the object model is then optimised for efficient implementation.

The design must also take into account concurrency and dynamic control flow as determined by the system design document. The implementation of each attribute and association is determined and the sub-systems are packaged into modules. Rumbaugh *et al.* [91] give a step by step guide for the object design stage, resulting in a design document consisting of a detailed object model, a detailed dynamic model and a detailed functional model.

3.8 Modification to OMT for analysis of DEDSs

As described in Section 3.7.2, the OMT dynamic model is represented by a state diagram. However, any system described by means of a state diagram can be represented by a Petri net. Moreover, a state diagram for a simple system can consist of an infinite number of nodes and, although Petri nets can be graphically complex, they are of finite size and their graphical structure is useful to visualise the behaviour of the system. Unlike state diagrams, Petri nets are modular and larger nets can be formed by simply merging places or transitions. Also, Petri nets are able to model asynchronous, concurrent processes whereas these are more difficult to model with state automata. In this Thesis, the following additions and modifications are made to OMT to make it more suitable to modelling DEDSs and to improve the representation and analysis of the dynamic model.

- The object model must include the set of controllable events [3.8.1]
- The dynamic model is represented by a Petri net with control places [3.8.2]
- The state of the control places is driven by functions specified in the functional model [3.8.3]

3.8.1 Object model must include the set of controllable events

Normally, in the OMT object modelling phase [3.6.1], the designer would first identify the different objects that make up the system and classify them in a list of classes. The next step would be to add the list of attributes and operations to the object models. There is no fixed rule on which attributes and operations should be included in the model, this is usually subjective. However, since the purpose of modelling the DEDS is to analyse and control it, it is necessary to include the set of controllable events [Ramadge and Wonham 87] in the list of operations of the class definitions. As a direct consequence of this requirement, it is also necessary to include the pre-conditions and post-conditions in the attribute list of the class definitions.

3.8.2 Dynamic model represented by a Petri net with control places

OMT is modified so that the dynamic model of the objects will be represented by a controlled Petri net [Section 2.3.6.2] instead of state diagrams. The Petri net model representing the dynamic behaviour of the various objects can be obtained directly from the object model and problem statement by taking the following steps:

- Class associations represented in the object model are represented by transitions that model the communication between the objects. These transitions are termed object communication transitions (OCTs).
- A controllable event is represented by a transition with a control place as one of its inputs.
- Binary attributes representing the pre-conditions and post-conditions of the controllable events are represented by Petri net places.

3.8.3 Control places driven by outputs of functions in the functional model

The functions that constrain and optimise the performance of the system are defined as algebraic equations or logical statements, the outputs of which drive the state of the control places. Every control place of the dynamic model must have a function associated with it to ensure that the behaviour of the system is constrained to satisfy the behaviour specified in the problem statement.

3.8.4 Synthesising the Petri net model

To synthesise the complete Petri net model of the system, the appropriate number of objects are instantiated. These are then "hooked" together by merging the object communication transitions. Table 3.1 shows the relationship between the three models of the modified OMT methodology.

Object model	Dynamic model	Functional model
Binary attributes	Petri net places	-
Controllable events	Controlled transitions	functions
Associations	Object communication transitions	-

Table 3.1 Relationship between the three OMT models

Therefore, the set of binary attributes defined in the object model is represented by a set of Petri net places in the dynamic model. The set of controllable events defined in the object model is represented by the set of controlled transitions in the dynamic model and functions in the functional model. The associations between classes defined in the object model are represented by object communication transitions in the dynamic model.

3.9 Applying OMT methodology to analyse DEDSs

This section illustrates the use of OMT, as modified in the previous section, by application to two classical DEDSs: The Dining Philosophers problem [Courtois *et al.* 71, Peterson 81, Hoare 85], and a Drum-Slider system [Sagoo and Holding 90, Jiang 95]. The advantages of using OMT to analyse the system and Petri nets to represent the dynamic model of DEDSs will be highlighted.

3.9.1 The dining philosopher problem

Problem Statement

There are five philosophers around a circular table with five forks on it. A philosopher may only be eating or thinking at any one time. Each philosopher has access to two forks, one on either side. Each fork is shared by two philosophers and may be either on the table or in use by a philosopher. To avoid a deadlock situation (documented in [Hoare, 85]), a philosopher must pick up the two forks simultaneously to eat. Initially all the philosophers are thinking.

Object model

In this problem two types of object are identified; a philosopher and a fork. Therefore the OMT object model consists of a fork class and a philosopher class. The following associations exist between the classes: a philosopher can be the right diner or left diner of a fork, whilst a fork can be the left utensil of the philosopher to its right or the right utensil of the philosopher to its left. The philosopher may either have two forks or none at all. The classes and associations between them are shown in the object model illustrated in Fig. 3.12.

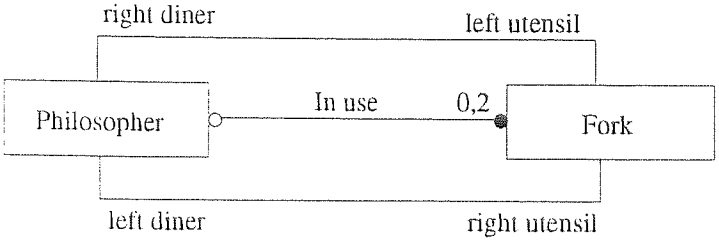


Fig. 3.12 Dining Philosophers object diagram

Using the problem statement as a guide and standard OO techniques [Booch 94], the attributes of the objects that are considered to be important for the correct functioning of the system are listed in the class definitions. Since we assume that the philosophers make their own decisions as to whether they want to eat or think, there are no controllable events in the system. The philosopher class has two attributes `Thinking` and `Eating`, with the initial condition being that the philosopher is thinking. The fork has an attribute `Available` that is initially true, meaning that initially it is on the table and available to be picked up by one of its neighbouring philosophers. These attributes are of a Boolean type and are listed in the class definition in Fig. 3.13

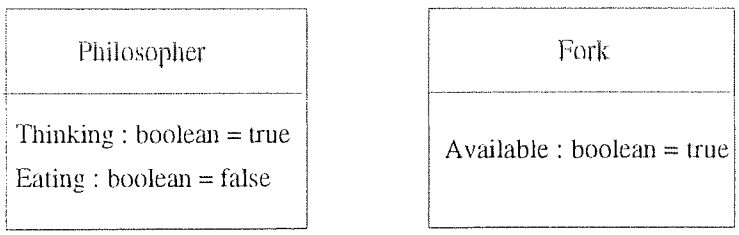


Fig. 3.13 Attributes of Philosopher and Fork classes

Thus the object model of the Dining Philosophers problem consists of the object model diagram (Fig. 3.12) and the class definitions (Fig. 3.13).

Dynamic model

In the first stage of obtaining the dynamic model, the events between objects are identified and represented by labelled Petri net transitions. These transitions are termed "object communication transitions" (OCTs). In this problem, the philosopher objects interact with the forks objects by picking both forks or releasing them. Thus the OCTs for the philosopher and fork classes are as shown in Fig. 3.14.

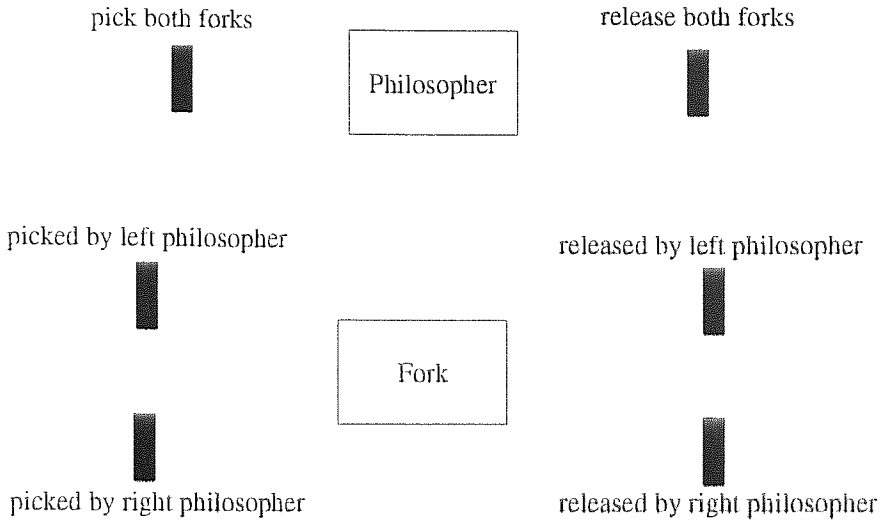


Fig. 3.14 Illustrating the object communication transitions OCTs

The second stage of dynamic modelling involves developing a Petri net model for each class to describe its important dynamic behaviour. The objects' binary attributes, listed in the object model, are represented by Petri net places in the dynamic model and events are represented by transitions. The places and transitions are then linked by directed arcs to represent the dynamic behaviour described in the problem statement. The resulting dynamic models are illustrated in Fig. 3.15.

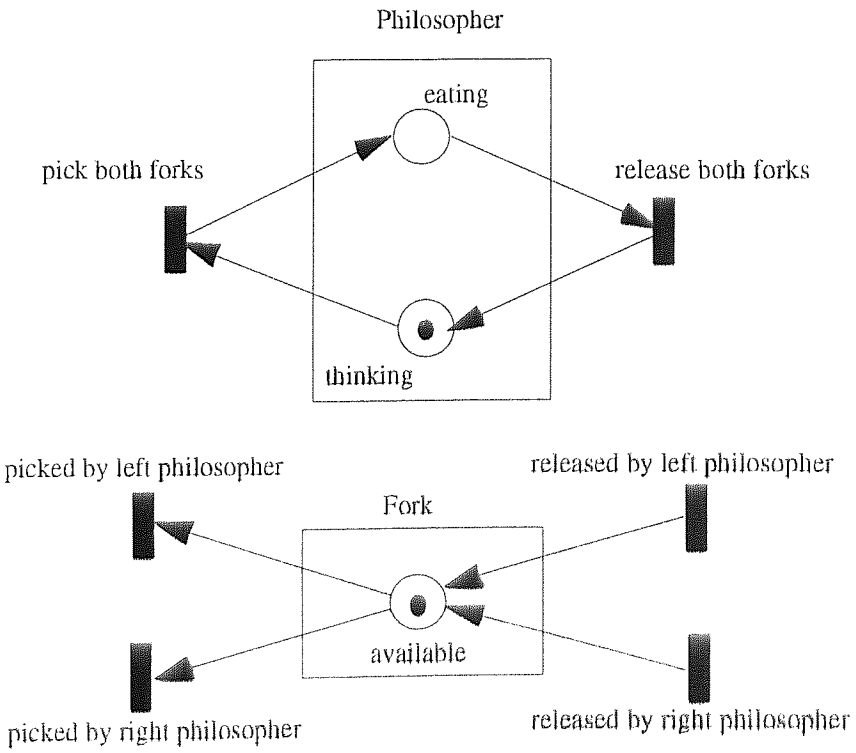


Fig. 3.15 The class dynamic models

The dynamic model describing the system is obtained by instantiating the required number of objects and "hooking" them together by merging the relevant OCTs as shown in Fig. 3.16. In this example, five fork objects and five philosopher objects are instantiated and hooked together by merging their common OCTs (Fig. 3.14). Therefore, the OMT dynamic model consists of the Petri net models representing the dynamic behaviour of the classes and the object interactions (OCTs).

Functional model

This system has no external inputs or outputs and does not have any functions that modify data values. Therefore it does not have a functional model associated with the objects that comprise the system.

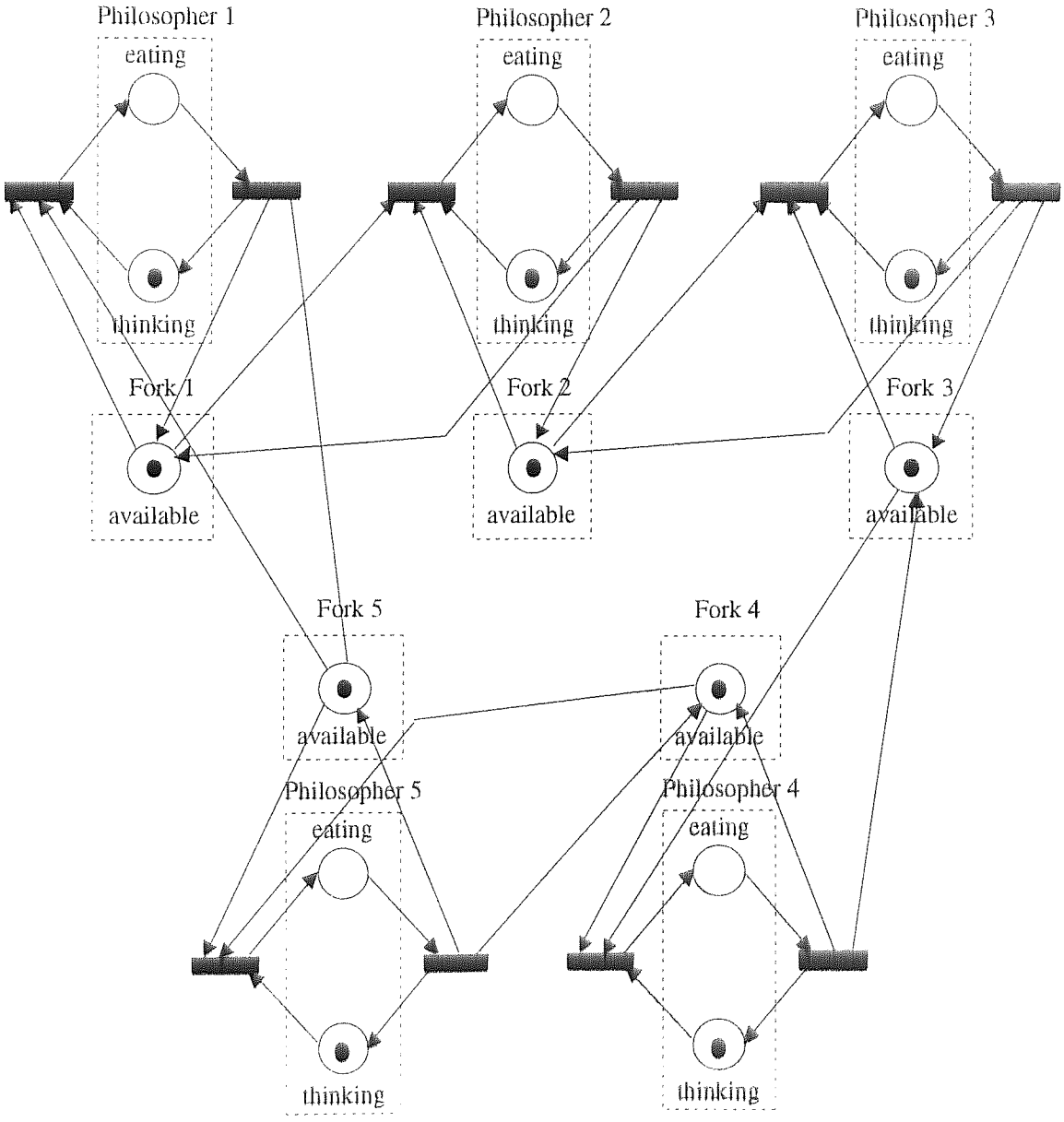


Fig. 3.16 Dining Philosophers Petri net

Behavioural analysis

The Petri net was analysed using a Petri net analysis tool (developed as part of this research work [Chapter 5]) and was shown to be live and deadlock free. Therefore the final analysis document for the dining philosophers problem consists of the problem statement, object model (Figs. 3.12 and 3.13), object communication model (Fig. 3.14) and the Petri net model (Fig. 3.16).

3.9.2 A Drum-Slider system

This section will illustrate the application of OMT analysis, as modified in this Thesis, to model a drum-slider system that was investigated by Sagoo and Holding [90] and later by Jiang *et al.* [95], who used a rule-based approach to model the system.

Problem Statement

The system consists of a drum and transfer slider, illustrated in Fig. 3.17. The drum rotates in steps of 22.5°. The slider movement may be described as being simple harmonic. It inserts into the drum slots when the drum is stationary and in the correct position. The drum starts to rotate again once the slider has reached the safe point (point (c) on Fig. 3.17) on its withdrawal stroke. If on the approach of the slider to the drum, at the decision point (point (b) on Fig. 3.17), the drum is still rotating, then the slider must abort its motion and return to its home position (point (a) on Fig. 3.17). The home position is that of maximum withdrawal. The system is initially at rest with the drum stationary and in its correct position and the slider at its home position.

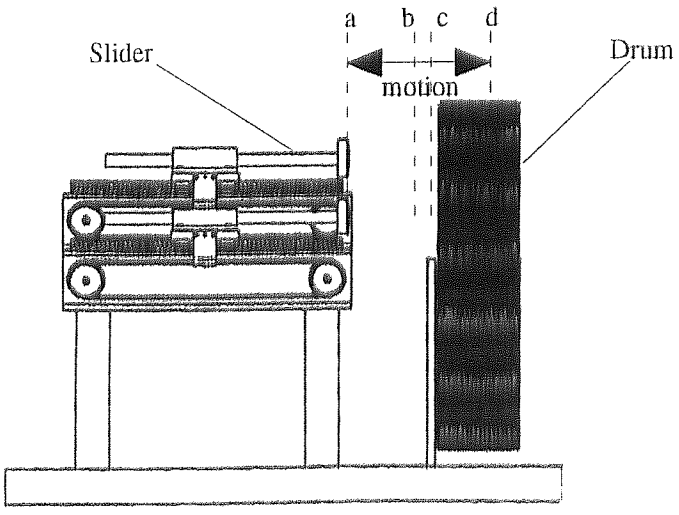


Fig. 3.17 The drum-slider mechanism

Object model

In this problem two types of object are identified; a drum and a slider. Therefore the OMT object model consists of a drum class and a slider class. The relationship of the classes is one-to-one and the slider interacts with the drum by inserting into the drum or withdrawing from the drum. Adding the associations between classes results in the object model shown in Fig. 3.18.

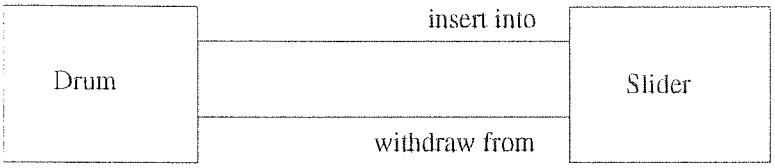


Fig. 3.18 Drum-Slider object diagram

Using the problem statement as a guide, the attributes of the objects that are considered to be important for the correct functioning of the system and the controllable events are listed in the class definitions. The drum has two binary attributes: `rotating` and `stationary`, with the initial condition being that the drum is stationary. The controllable event associated with the drum is `start rotating`. The event that stops the drum rotating is an automatic event and therefore it is not necessary to include this event in the class definition of the drum.

The movement of the slider is simple harmonic and is therefore a continuous motion. However, in this problem we are interested in controlling the interactions of the drum and slider which are discrete events. Therefore, the simple harmonic motion can be mapped into a sequence of states that highlight the important parts of the slider motion, these being: `At home position`, `inserting before decision point`, `at decision point (inserting)`, `past decision point`, `at safe point (withdrawal)`, `past safe point` (Fig. 3.19).

The controllable event associated with the slider is the `abort` event which makes the slider abort its inward stroke if the drum is still rotating when the slider has passed the decision point. Thus the class definitions are shown in Fig. 3.20.

Therefore the object model of the Drum-Slider problem consists of the object model diagram (Fig. 3.18) and the class definitions (Fig. 3.20).

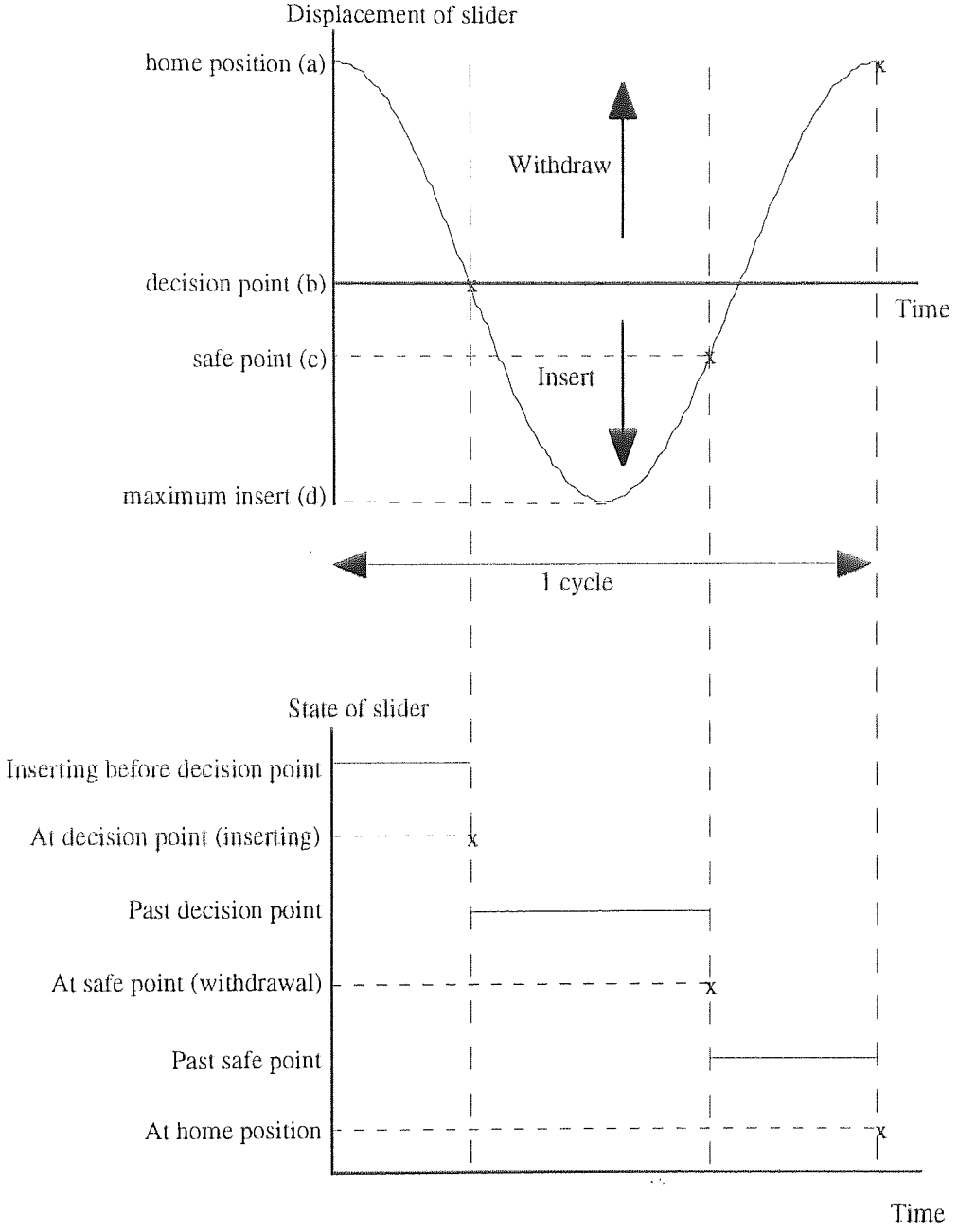


Fig. 3.19 Mapping a continuous motion to a set of discrete states

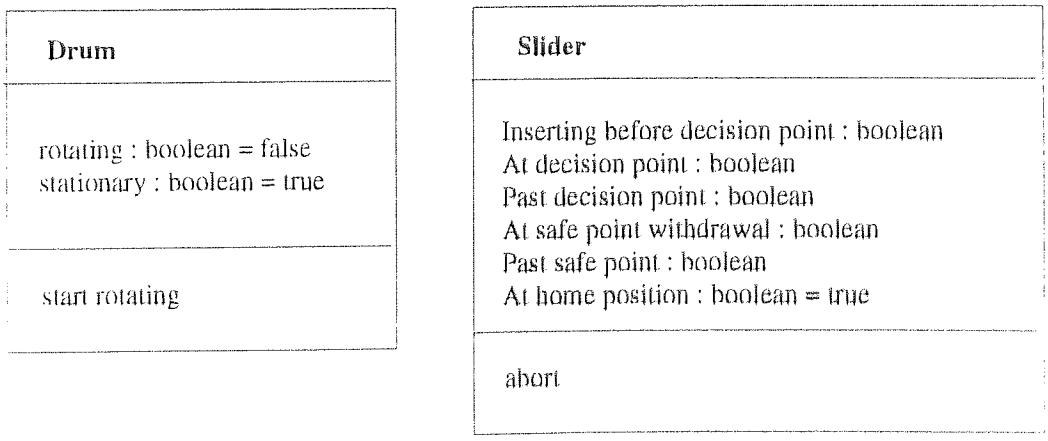


Fig. 3.20 Drum and Slider class definition

Dynamic model

In the first stage of obtaining the dynamic model, the events between objects are identified and represented by labelled Petri net transitions (Fig 3.21). In this problem, the slider object interacts with the drum by inserting into it and withdrawing from it.

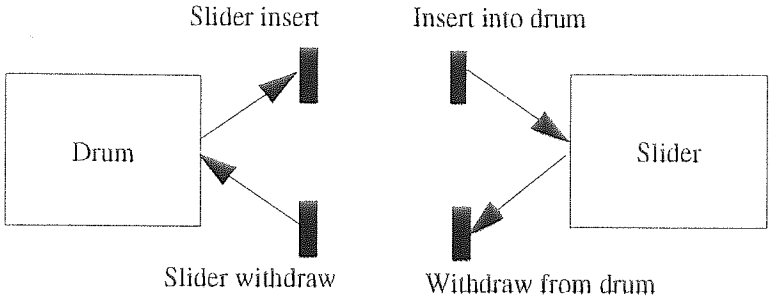


Fig. 3.21 Illustrating the Object Communication Transitions (OCTs)

As in the previous example, the second stage involves developing a Petri net model for each class to describe important dynamic behaviour. The objects' binary attributes in the object model are represented by Petri net places in the dynamic model and events are represented by transitions. The places and transitions are then linked by directed arcs to represent the dynamic behaviour described in the problem statement. The resulting dynamic models are illustrated in Fig. 3.22.

The dynamic model describing the system is obtained by instantiating the required number of objects and "hooking" them together by merging the relevant OCTs, illustrated in Fig. 3.23. In this example one drum and one slider object is instantiated and hooked together by merging the appropriate OCTs (Fig. 3.21).

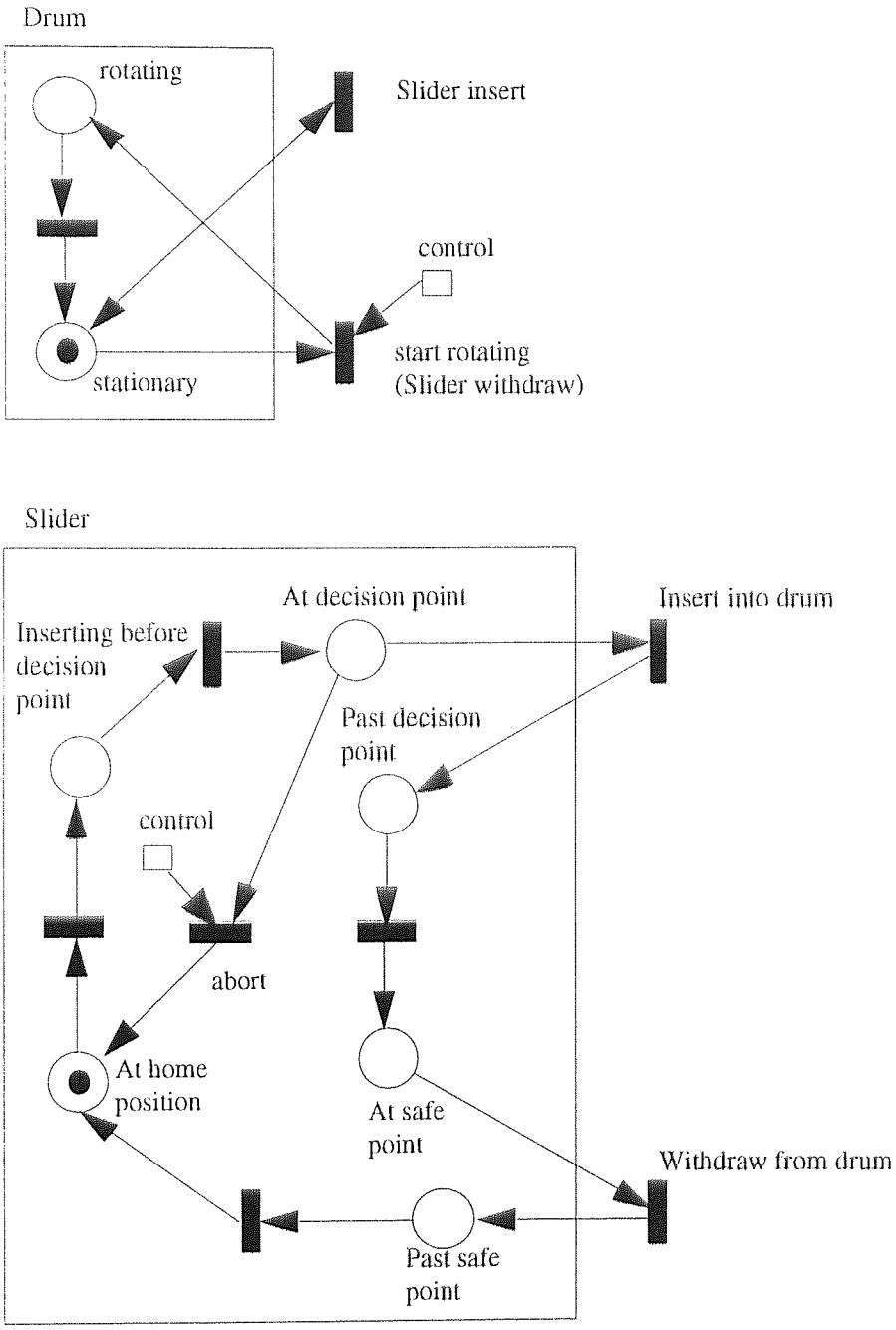


Fig. 3.22 The object dynamic models

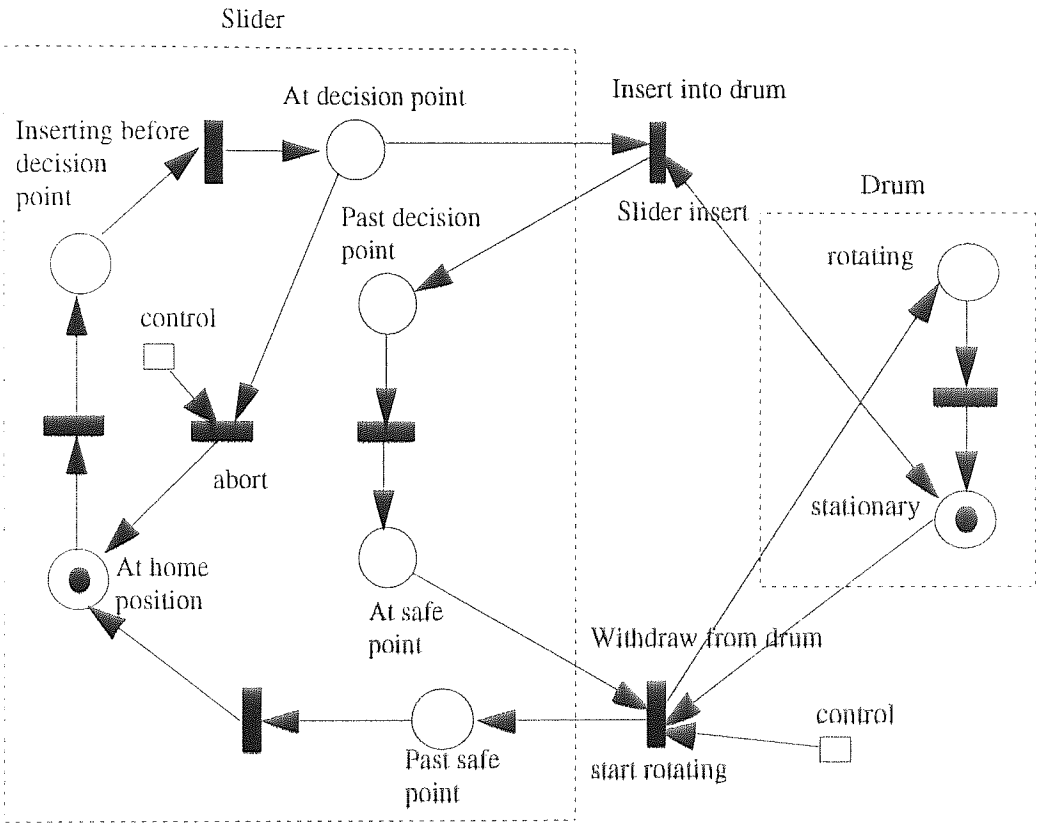


Fig. 3.23 The Drum-Slider Petri net

Therefore, the dynamic model consists of the OCTs, the Petri net models representing the dynamic behaviour of each class, and the Petri net representing the whole system

Functional model

The functional model defines the constraints of the system operation. The output of the functions change the state of the control places in the Petri net model. This system has no external inputs or outputs. However, it has two functions that are defined in the object model. These are the `start rotating` operation of the drum object and the `abort` operation of the slider object. The Boolean values that determine whether the function is performed or not are represented by control flows (dashed line) in the data flow diagram illustrated in Fig. 3.24.

For the event `start rotating` to be enabled, the drum must be stationary and the slider must have reached the safety point on its withdrawal stroke. Since the `start rotating` event does not effect the state of the slider, this is represented by a dashed line. The function changes the state of the drum from `stationary` to `rotating`, so these states are represented by solid lines.

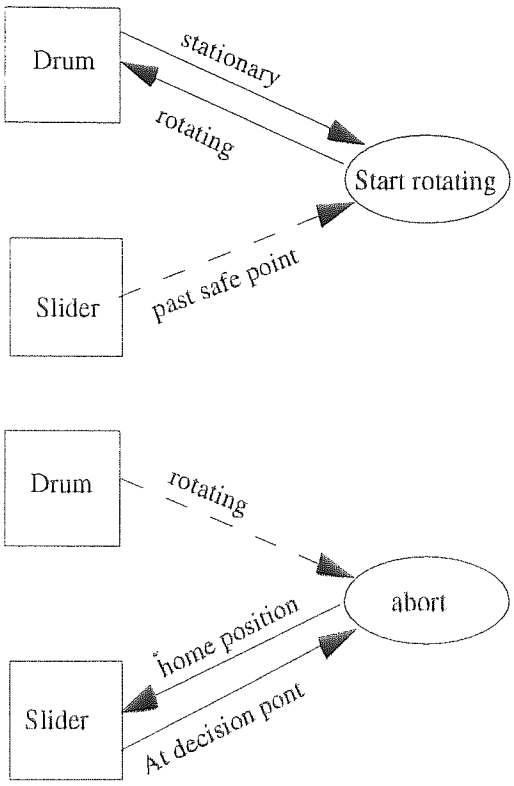


Fig. 3.24 Functional diagram for drum-slider problem

Similarly, for the event `abort` to be enabled, the slider must be at the decision point and the drum must be rotating. Since the `abort` event does not affect the motion of the drum, then the input to the `abort` enabling function is shown with a dashed line. The function changes the state of the slider from `At decision point` to `At home position`, these states are represented by solid lines.

From the problem statement, the condition for the drum to `start rotating` is defined as (note that the "." notation normally used in OO terminology is adopted. i.e. `drum.stationary` means "drum is stationary"):

```
drum.stationary ∧ piston.At safety point →
drum.start rotating
```

Since the states `stationary` and `At safe point` are pre-conditions of the transition representing `start rotating` in the Petri net model (Fig. 3.23) the state of the control place, `drum.control` = 1.

The other condition specified in the problem statement is related to the `abort` event which is defined as:

```
drum.rotating ∧ slider.At decision point → slider.abort
```

Since `drum.rotating` is not a pre-condition of the transition representing `slider.abort` in the Petri net model (Fig. 3.23), the state of the control place, `slider.control = drum.rotating`

Behavioural analysis

The Petri net was then analysed using a Petri net analysis tool (developed as part of this research work [Chapter 5]) and was shown to be 1-bounded, live and deadlock free. Using concurrency sets [Skeen 83] and P-invariants that were generated by the Petri net analysis tool, the following safety properties of the system were verified:

1. There can never be a situation where the slider will travel beyond the decision point if the drum is not stationary and in the correct position.

2. There can never be a situation where the drum will start rotating before the slider has withdrawn past the safety point.

The use of the modified OMT has resulted in a complete analysis document for the problem, consisting of the problem statement, object model, the Petri net model, the functional model and Petri net analysis results.

3.9.3 Evaluation of the methodology

In the previous two sections the modified OMT methodology was applied to two well documented problems. The Petri net model representing the dining philosopher problem is equivalent to that obtained by Peterson [81] and the Petri net representing the drum-slider problem is similar to that obtained by Sagoo [92] shown in Fig. 3.25. However in both examples, the object oriented methodology re-used designs by grouping identical objects into classes and resulted in modular and more understandable designs. Another advantage of the Petri net represention of the drum-slider problem (Fig. 3.23) over that of Sagoo [92] is that it has fewer places and transitions and a smaller state space.

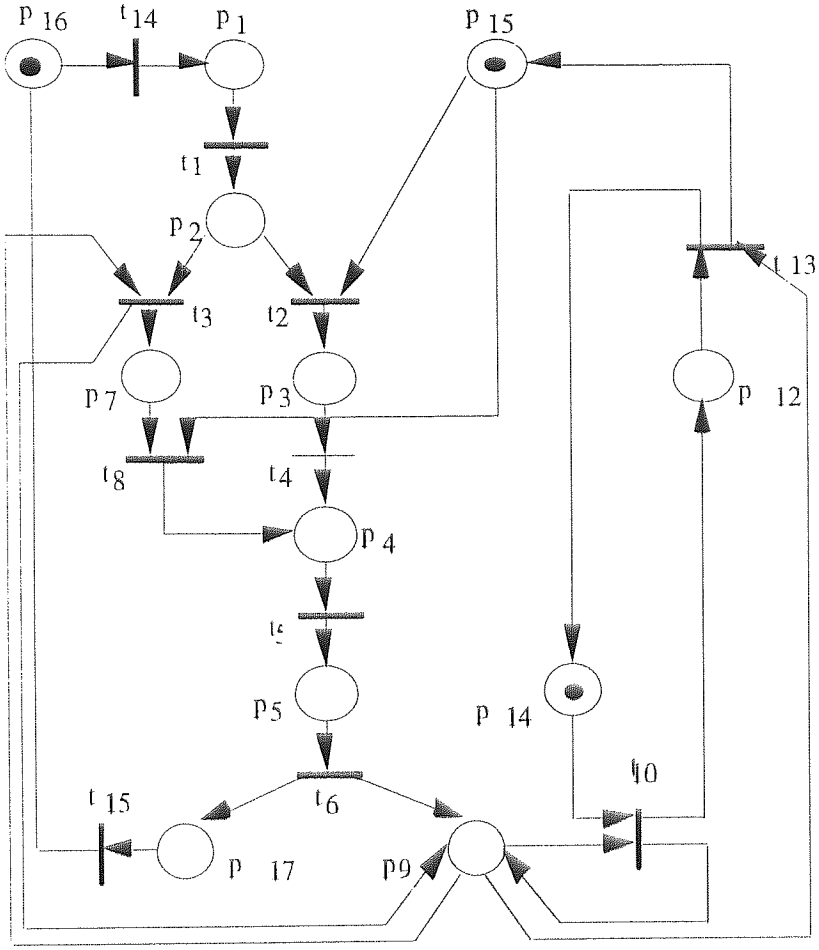


Fig. 3.25 Petri net representation for drum-slider problem [Sagoo 92]

Places	Transitions
P_1 - Slider accelerating towards drum P_2 - Slider at decision point P_3 - Slider inserting into drum P_4 - Slider fully inserted P_5 - Slider withdrawing P_7 - Slider at rest P_9 - Drum rotating status P_{12} - Drum rotating at constant velocity P_{14} - Drum stationary P_{15} - Drum stationary status P_{16} - Slider at rest P_{17} - Slider's motion to rest	T_1 - Slider enters decision point T_2 - Slider proceeds with insertion T_3 - Slider starts to abort motion T_4 - Slider makes insertion into drum T_5 - Slider starts to withdraw from drum T_6 - Slider clears drum T_8 - Slider starts its motion T_{10} - Drum starts to rotate T_{13} - Drum stops rotating T_{14} - Slider commences its motion T_{15} - Slider reaches zero velocity

Table 3.2 - Meanings of places and transitions of Fig. 3.25

3.10 Conclusion

This chapter has reviewed currently available Petri net synthesis techniques for modelling large systems. It was noted that these techniques are based on functional abstraction and that the experience of researchers has shown that functional abstraction is not practical for large industrial applications. This chapter therefore proposed using an object oriented methodology to facilitate the design of complex systems, produce more understandable designs and specifications, facilitate the transition between design and implementation and enable re-use of designs. The methodology was based on the well known OMT methodology and was applied to two well documented problems. The following chapter describes the next stage of the design process, in which the controller for a DEDS is designed.

Chapter 4

Optimal control of DEDSs

4.1 Introduction

Following the specification and analysis of a DEDS as described in the previous chapters, the next stage in the design process is that in which the controller is designed. Closed loop control of DEDSs is based on the supervisory control theory developed by Ramadge and Wonham [87], who used state automata to model the plant and its supervisory controller. However, as described in [Chapter 2] there are several advantages of using Petri nets over state automata to model DEDSs. Hence, this chapter reviews research work that adopts Petri net theory for supervisory control including that of Valette *et al.* [85], Holloway and Krogh [90] and Giua and DiCesare [94a].

Having designed the plant and controller, the next step is to find a way to control it in the best possible way. Exclusive-use shared resources (such as production units in batch process plants, machines in FMSs, processors in computer systems, cash machines in a bank) are components present in most DEDSs. Their common feature is that they are shared by different users or processes, but can only be used by one user or process at a time. The problems raised due to the presence of shared resources is that the overall performance of the system depends on the order in which they are allocated to the users or processes. Since a large capital is involved in designing and operating DEDSs it is important to ensure optimal operation to make these systems commercially viable. Therefore this chapter reviews Petri net based scheduling algorithms and develops an efficient scheduling algorithm to allocate the plant's shared resources in such a way so as to guarantee optimal performance of the plant.

Another consideration is that if at some point there is a physical change in the plant, say, because of a partial failure of a resource, there will be a mismatch between the plant and the supervisory controller. Therefore any changes in the plant set-up will invalidate the control system. This is unacceptable in real-world industrial systems where the controller should at least be able to shut down the plant safely in the case of a partial failure. A novel design for a DEDS controller, first presented by the author of this Thesis in [Azzopardi and Holding 95], is described in this chapter. It is based on the OMT model of the plant and is re-configurable to accommodate for unexpected changes in the plant.

4.2 Supervisory control of DEDSs

Following the standard practice of control theory, Ramadge and Wonham [87] distinguish between the "plant" (object to be controlled) and the "controller" (the agent doing the controlling). This distinction tends to simplify the problem of defining exactly what controlled behaviour is required, as well as what constraints on behaviour are imposed by the underlying physical system.

As described in Section 1.5, uncontrollable events occur spontaneously, however, the occurrence of controllable events depends on the enabling/disabling action of a controller defined by the function $\gamma(w)$, where w is the sequence of events that have been observed up to the current state. A controller of this type is known as a supervisory controller, the fundamental purpose of which, is to provide closed loop control to force the system to behave as specified under a variety of operating conditions. In the control theory of Ramadge and Wonham, the plant and controller are both modelled as finite-state automata with complementary input-output behaviour: The plant automaton generates events from spontaneous state transitions which can he enabled or disabled by the input signal from the controller; whereas the controller automaton accepts events from the plant which force state transitions in the controller and change control inputs to the plant (Fig. 4.1).

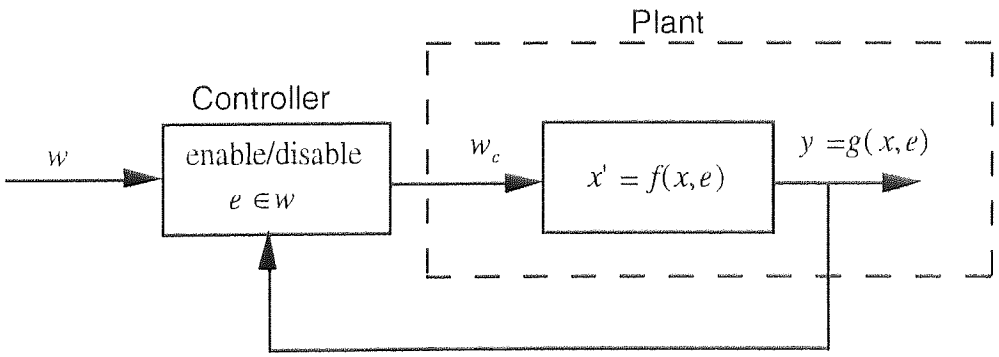


Fig. 4.1 Supervisory Control of DEDSs [Cassandras 93]

4.2.1 A logical DEDS model

This section introduces the notation used in the theory of supervisory control of Ramadge and Wonham [87] in which a DEDS model is used to study the sequences of events that the process can generate. Let Σ denote the finite set of event labels and Σ^* denote the set of all finite strings of elements of the set Σ, including the empty set ε, then, an element of Σ is referred to as an event.

A string, $u = \sigma_1\sigma_2...\sigma_k \in \Sigma^*$, where $\sigma_1, \sigma_2, ..., \sigma_k$ are events, represents a partial event sample path. Partial because there may be more events after σ_k. The set of all physically possible sample paths is a subset L of Σ^* and L is called a language over the alphabet Σ.

A string u is a prefix of a string $v \in \Sigma^*$ if for some $w \in \Sigma^*$, v=uw. If v is an admissible sample path, then clearly so are all the prefixes of v. If the prefix closure of $L \subseteq \Sigma^*$ is defined to be the language $\overline{L} = \{u: uv \in L \text{ for some } v \in \Sigma^*\}$ then it is required that $\overline{L} = L$. In this case it is said that L is prefix closed.

Ramadge and Wonham model the behaviour of a DEDS as a prefix closed language L over the event alphabet Σ. Each $u \in L$ represents a possible event sample path of the DEDS. For example, the trivial DEDS with two events, which operates so that the events α and β always occur alternately, with α or β occurring first, has the behaviour: $L = \{\varepsilon, \alpha, \beta, \alpha\beta, \beta\alpha, \alpha\beta\alpha, ...\}$. The behaviour of the DEDS can be constrained if $|w|_\alpha$ is made to denote the number of occurrences of the event α in the string w, and $L = \{w \in \Sigma^*: \text{for each prefix u of w, } |u|_\alpha \leq |u|_\beta\}$, then L represents a DEDS in which the number of occurrences of the event α is always less than or equal to the number of occurrences of β.

To construct more elaborate examples it is convenient to have a means of language representation. Ramadge and Wonham [89] represent a DEDS by a finite state automaton. Hence, to represent a behaviour L, a generator G is an automaton (defined in [Hopcroft and Ullman 79]) consisting of a state set Q, initial state q_0, and a transition function $\delta: \Sigma \times Q \rightarrow Q$. The set of events possible at state q is the set $\Sigma(q) \subseteq \Sigma$ such that for each $\sigma \in \Sigma(q), \delta(\sigma, q)$ is defined.

G can be represented as a directed graph with a node set Q and an edge $q \rightarrow q'$ labelled σ for each triple (σ, q, q') such that $q' = \delta(\sigma, q)$. G is interpreted as a device that starts in its initial state q_0 and executes state transitions by following its graph. State transitions are considered to occur spontaneously, asynchronously and instantaneously.

The transition function δ of G is extended to a function on $\Sigma^* \times Q$ by defining $\delta(\varepsilon, q) = q$ and $\delta(w\sigma, q) = \delta(\sigma, \delta(w, q))$ whenever $q' = \delta(w, q)$ and $\delta(\sigma, q')$ are defined. The abbreviation $\delta(w,q)!$ means "$\delta(w,q)$ is defined". In terms of the graph of G, $\delta(w,q)!$ means that there is a path in the graph starting from q that is labelled by the consecutive elements of the string w. The closed behaviour of G is defined to be the prefix closed language $L(G) = \{w : w \in \Sigma^* \text{ and } \delta(w, q_0)!\}$

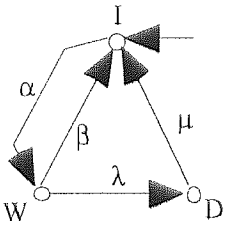


Fig. 4.2 A simple generator [Ramadge and Wonham 87]

Consider the simple generator in Fig. 4.2. It models a machine with three states, labelled I (Idle), W (Working) and D (Down); The initial state, I, is marked by an arrow, and there are four possible state transitions, each labelled by the associated observed event from the event set $\Sigma = \{\alpha, \beta, \lambda, \mu\}$. The closed behaviour of G is the set of all strings obtained by starting in the state I and following the graph. In the formalism of regular expressions this is written as $L(G) = (\alpha\beta + \alpha\lambda\mu)^*(\varepsilon + \alpha + \alpha\lambda)$.

4.2.2 Controllability and Supervision of DESs

This section introduces supervisory control theory of Ramadge and Wonham [87] who state that to control DES it is assumed that certain events of the system can be disabled when desired. This allows the controller to influence the evolution of the system by prohibiting particular events at certain times. The set of events, Σ, is partitioned into controllable and uncontrollable events. Thus $\Sigma = \Sigma_u \cup \Sigma_c$. The events Σ_c can be disabled at any time (e.g. start motor), while those in Σ_u model events over which the controller has no influence (e.g. machine breaks down).

The control input of G is a subset $\gamma \subseteq \Sigma$ satisfying the condition $\Sigma_u \subseteq \gamma$ (the uncontrollable events are always enabled). If $\sigma \in \gamma$, then σ is enabled by γ (permitted to occur) otherwise σ is disabled by γ (prohibited from occurring).

Let $\Gamma \subseteq 2^{\Sigma}$ denote the set of control inputs. A DES represented by the generator G, equipped with a set of inputs Γ is called a controlled DES (CDES). For convenience, a CDES is referred to by its underlying generator G.

Control of a CDES, G, consists of switching the control input through a sequence of elements $\gamma, \gamma', \gamma'', \ldots$ in Γ, in response to the observed string of previously generated events. This type of controller is called a supervisor. Formally, a supervisor is a map $f: L \rightarrow \Gamma$, which specifies the control input f(w) for each possible string of generated events, w. The objective is to design a supervisor that selects control inputs in such a way that the given CDES, G, behaves in obedience to various constraints. Constraints can be viewed as a requirement that certain undesirable sequences of events are not permitted to occur, whilst at the same time, certain other desirable sequences are allowed.

When a CDES is controlled by a supervisor, f, it operates as before, except that it obeys the additional constraint that, following the generation of a string w, the next event must be an element of $f(w) \cap \Sigma\left(\delta\left(w, q_0\right)\right)$. The closed loop system of G supervised by f is denoted by (G,f). Then, the behaviour of (G,f) is denoted by L(G,f), or simply L_f, formally defined as follows:

i) $\varepsilon \in L_f$; and

ii) $w\sigma \in L_f$ iff $w \in L_f$, $\sigma \in f(w)$, and $w\sigma \in L$

If G has some marker states (i.e. states that, say, mark the completion of a process), then the language controlled by f in G is $L_m(G,f) = L_{mf} = L_m(G) \cap L_f$ which is simply that part of the original marked language that survives under supervision. If L_m represents completed tasks, then this language is important because it indicates those tasks that will be completed under supervision.

In practice one may require an alternative representation for the supervisor, f. For this, Ramadge and Wonham used state realisation in terms of an automaton together with an output map [Ramadge and Wonham 87]. Let $T = \left(\Sigma, X, \xi, x_0\right)$ be an automaton, and $\Phi: X \rightarrow \Gamma$, we say that the pair (T, Φ) realises the supervisor f if for each $w \in L_f$, $\Phi\left(\xi\left(w, x_0\right)\right) = f(w)$. This means that the value of f on the string w can be found by first applying w to T causing T to be driven from its initial state to some state x and then computing $\Phi(x)$. Thus T is a standard automaton whose state transitions are driven by the events in Σ.

In standard control terminology G plays the role of the "plant" (object to be controlled), T functions as the "observer" or "dynamic compensator", and Φ is the "feedback". It is possible to realise a supervisor as another DES S. In this case, the control action of S on G is implicit in the transition structure of S as shown in Fig. 4.3 below.

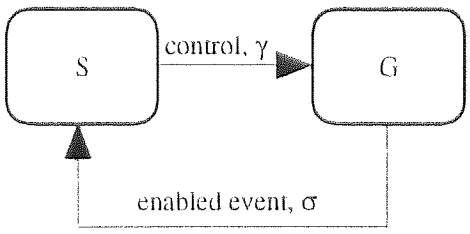


Fig. 4.3 Supervision of a DES

The basic problem in supervisory control is to modify the open loop behaviour of a given DES G so that its behaviour satisfies the specification of the system. Consider the following problem: Given a CDES G with behaviour L, what closed loop behaviours $K \subseteq L$ can be achieved by supervision? This is the concept of controllability where it is said that $K \subseteq \Sigma^*$ is controllable if $\overline{K}\Sigma_u \cap L \subseteq \overline{K}$. This means that for any prefix of a string in K, i.e. any $w \in \overline{K}$, if w followed by an uncontrolled event $\sigma \in \Sigma_u$ is in L, i.e. $w\sigma \in L$, then it must also be a prefix of a string in K, i.e. $w\sigma \in \overline{K}$. In this sense, $\overline{K}$ is conditionally invariant under the action of Σ_u. Generally, the aim of supervision is not to modify L, but instead to achieve a prescribed language for L_{mf}, and to do so while preserving the desired non-blocking property. The conditions under which this is possible are stated in terms of language controllability [Ramadge and Wonham 87].

4.3 Petri net based supervisory control

As described in the previous sections, supervisory control theory of Ramadge and Wonham [87] used a state transition structure based on finite state automata to implement control. However, as discussed in [Chapter 2] Petri nets have several advantages over state automata, namely:

(i) Since the states of a Petri net are represented by the possible markings and not by the places, they allow a compact description; i.e., the structure of the net may be maintained small in size even if the number of markings grows.

(ii) They allow modular synthesis; i.e., the net can be considered as composed of interrelated subnets, in the same way as a complex system can be regarded as composed of interacting subsystems.

(iii) Petri nets can model asynchronous, concurrent processes in a straightforward manner whereas this is more difficult to model with state automata.

Not surprisingly, several researchers have used Petri nets to control DEDSs. Courvoisier *et al.* [83] were the first to describe a system for the design of control logic of Petri net based programmable logic controllers (PLCs). They used the system to control two automatic guided vehicles sharing a common path. This work was extended by Valette *et al.* [85], who show that, by using the Petri net model representing a FMS, it is possible to control the plant by means of a token-player computer program (Fig.4.4). The token player algorithm acts as a supervisory controller, playing tokens on a Petri net model representing the plant. It continuously searches for enabled transitions and fires them. It interacts with the plant by receiving messages which represent the occurrence of an event in the plant, for example, threshold crossings detected in local controllers [Andreu *et al.* 94]. As illustrated in Fig. 4.4, the token-player updates the marking of the Petri net, which represents the state of the plant, when a message is received from a local controller. This is represented by the left part of Fig. 4.4.

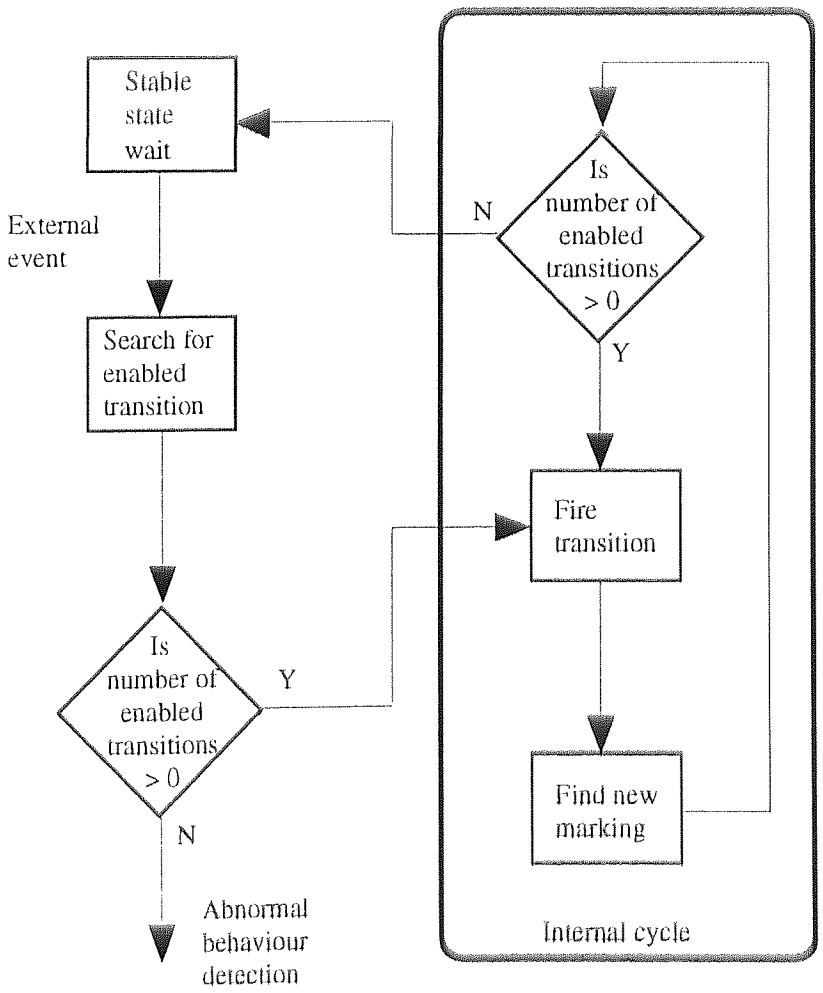


Fig. 4.4 The token-player algorithm [Valette 95]

When the token-player is waiting for a message, then it is in the stable state. When an event occurs, the token-player searches the Petri net data structure to locate the transition representing this event. The current marking of the net must be such that this transition is enabled, otherwise, the state of the supervisor is said to be inconsistent with the actual state. This means that there is a fault in the plant, or that the control system is faulty or that there was a design error at the specification phase of the controller [Valette 95]. The fact that it is impossible to fire a transition which is not enabled ensures that any failure will be detected. The right hand side of Fig. 4.4 shows the internal cycle of the token-player. Once the marking has been updated, then the transitions that are enabled and do not depend on receiving a message are fired. If any of these transitions have messages associated with them, they are executed. The internal cycle ends when all the transitions that are enabled by the current marking depend on receiving a message from the plant, which is the stable state.

Krogh *et al.* [88] describe a set of software tools for the automatic generation of control programs for discrete manufacturing processes. A Petri net model of the control and process logic is extracted from a relational database containing the system specifications. The controller is of the token-player type as in [Valette *et al.* 85], where the control computer maintains a Petri net model of the system to update the marking of the model. Holloway and Krogh [90] used controlled Petri nets (CPNs) (described in Section 2.3.6.2) for the efficient solution of a class of control problems. The main characteristic of the CPN approach to supervisory control is that no transition structure for a controller is given. The control, u, is a function of the actual marking of the net and needs to be computed at each step.

Giua and DiCesare [94a] also proved that Petri nets can be used as language generators in the framework of supervisory control of Ramadge and Wonham [87]. Given a marked Petri net Z=(P,T,I,O,m) with initial marking m_0, the alphabet Σ is represented by the set of transitions T. The closed behaviour is given by the language $L(Z,m_0)=\{\sigma \in T^* | (\exists m)[m_0[\sigma > m]\}$. Given a set of final markings m_f, the marked behaviour is defined as $L_m(Z,m_0)=\{\sigma | (\exists m \in m_f)[m_0[\sigma > m]\}$. A DES is said to be non-blocking when $\overline{L}_m = L$, (where $\overline{L}_m$ is the set of all prefixes of strings in L_m) i.e., any string $\sigma \in L$ can be completed into a string $\sigma\tau \in L_m$.

As in [Holloway and Krogh 90] the transitions in T are partitioned into two disjoint subsets: the set of controllable transitions T_c (that can be disabled if desired), and the set of uncontrollable transitions T_u (that cannot be disabled by an external agent). The supervisor, S, is a Petri net that directly models the system.

The supervisor will run in parallel with the plant, i.e., whenever an event in the system occurs, then the transition in S, representing this event, fires. Furthermore, the transitions that are enabled in S, represent the events that are allowed to occur in the plant, while all other events are disabled. This is very similar to the concept of a token-player [Valette *et al.* 85], the main difference being that it is defined within the framework of supervisory control [Ramadge and Wonham 87]. The net S is called a proper supervisor if the following two properties are ensured:

Trimness

- The net S does not allow blocking markings, i.e., reachable markings from which the final marking cannot be reached.

Controllability

- It is not possible to reach a marking from which an uncontrollable event in the plant is enabled but its corresponding transition in S is not.

In the examples of Ramadge and Wonham [89] it was observed that in general the number of states grows exponentially as the number of systems increases. Thus for realistically sized systems it is unfeasible to generate and investigate the state space manually. Giua and DiCesare [94a] showed how integer programming techniques may be used to validate supervisors for control of DEDSs. These techniques are restricted to supervisors that can be modelled on a class of Place/Transition (P/T) net called elementary composed state machine (ECSM) nets. Their approach is restricted in the sense that, there exist supervisors that cannot be modelled as ECSM nets. Also, although Integer Programming problems are more manageable than methods based on brute force state space search, they cannot always be solved in polynomial time [Giua and DiCesare 94a].

4.4 Scheduling exclusive-use resources

The previous sections have shown how Petri nets can be used for supervisory control of DEDSs. The next stage is to formulate a strategy for the optimal control of the plant. This is done by allocating exclusive-use resources in such a way, so as to optimise the performance of the plant, which is similar to the operations research scheduling problem [Baker 74].

Definition 4.1

The general scheduling problem is defined by MacCarthy and Liu [93] as : n jobs $\{J_1, J_2, ..., J_n\}$ have to be processed and m resources $\{R_1, R_2, ..., R_m\}$ are available. A subset of these resources is required to complete the processing of each job. The processing of job J_j on resource R_i is called an operation and is denoted by the element, O_{ij} of matrix **O**. For each operation O_{ij}, there is an associated processing time denoted by the element, t_{ij}, of matrix **t**. Efficient algorithms for obtaining an optimum solution to the scheduling problem exist only for very simple flow shops: "There is no single algorithm that can solve a general scheduling problem" [Ku *et al.* 87]. The range of methods that have been developed can be grouped into three types [MacCarthy and Liu 93]: efficient optimal methods, enumerative optimal methods, and heuristic methods.

Efficient optimal methods

These methods generate an optimal schedule, with respect to some criterion, in polynomial time. They solve the problem optimally and efficiently even for a large number of jobs, however methods of this type can only be applied to specific problems and are limited to systems with only one (two in some cases) shared resource. For a larger number of resources, it is necessary to apply either enumerative or heuristic methods.

Enumerative optimal methods

Enumerative optimal methods involve a partial enumeration of the set of all possible schedules. Generally they involve mathematical programming formulations, followed by branch and bound methods and elimination methods. The time required for these methods to obtain a schedule grows exponentially as the number of resources that can operate concurrently increases. This is due to the well known state-explosion phenomenon, which makes these methods unsuitable for scheduling systems with a large number of jobs and shared resources.

Heuristic methods

A good heuristic strategy attempts to approximate an optimal solution in polynomial time. There are three main types of heuristics [MacCarthy and Liu 93]:

1. Decisions are made each time a resource is released, or when a job arrives in a queue. Priority rules are examples of this type of heuristic.
2. A neighbourhood structure is defined and the solution found must be optimal within this neighbourhood structure.
3. The order of jobs is determined on one resource after another. For example the shifting bottleneck procedure for job shop problems [Adams *et al.* 88]

Most heuristic algorithms incorporate branch and bound procedures in which the most promising part of the state space is searched exhaustively. The main disadvantage of heuristic methods is that they may be questionable in optimality and stability [Chang and Liao 94].

4.4.1 Petri nets for scheduling DEDSs

The ability of Petri nets to model the dynamic characteristics of DEDSs, has led some researchers to investigate the suitability of using Petri nets for modelling the scheduling problem. This section reviews the work that has been done in this area of research. Carlier *et al.* [85] showed how to model scheduling problems on a timed Petri net (defined in Section 2.3.6.1). They cited the fact that one could model time constraints as well as resource constraints as an advantage over using classical Petri net. This early work was however limited to the state machine class of Petri nets, where every transition has only one input place and one output place.

Viswanadham *et al.* [90] developed a way to prevent deadlock in FMSs using Petri net models. They showed how scheduling rules for ensuring deadlock prevention can be devised by carrying out an exhaustive path analysis of the reachability graph of the Petri net model. This is however impractical in real situations due to the combinatorial state explosion phenomenon so they developed a deadlock-avoidance algorithm that would 'look ahead' into the evolution of the system for a number of steps. Hatono *et al.* [91] proposed scheduling for conflict resolution by using priority rules depending on the scheduling objectives. The work of Viswanadham *et al.* [90] and Hatono *et al.* [91] concentrated on scheduling for deadlock avoidance rather than performance optimisation. However, in Chapter 3 of this Thesis it was shown that deadlock avoidance can be obtained by using a suitable Petri net synthesis method.

Researchers have recently begun to investigate the application of well-known operations research (OR) scheduling algorithms and artificial intelligence (AI) optimisation algorithms to Petri net models of DEDSs. Martinez *et al.* [88] illustrated a method for control of complex production systems, modelled by coloured Petri nets [Jensen 81] and scheduled by a real-time decision module based on knowledge-based scheduling using AI techniques. The necessity of including a real-time decision module was due to the fact that the systems they modelled were not completely deterministic.

Shen *et al.* [92] were the first to suggest scheduling a DEDS by applying the branch and bound search algorithm to the Petri net model of the system. They modified the branch

and bound algorithm by including tests to reduce the search space and hence improve the convergence of the algorithm. These results were extended by the author in [Azzopardi and Lloyd 94a].

Zhang [92] pointed out that most current production planning systems are planned on inadequate models that ignore the temporal costs of actions. He proposed a timed Predicate/Transition (TPr/T) net model for planning and used the well-known A^* search algorithm [Pearl 84] based on the TPr/T net to schedule the system. At the same time, Lee and DiCesare [92, 94a] presented a method for scheduling FMSs based on a timed-place Petri net combined with a heuristic function to limit state explosion. The performance of the search algorithm depends on the information included within the heuristic function.

4.5 A Petri net based scheduling algorithm

This section describes a Petri net based scheduling algorithm [Shen *et al.* 92, Azzopardi and Lloyd 94a] which uses a branch and bound algorithm applied to the timed Petri net model of the plant. To improve the efficiency of the algorithm and to make it usable for larger systems, heuristics are used to reduce the search space. This Thesis introduces a further improvement over the previously published scheduling algorithms by reducing the timed Petri net, effectively removing all the transitions that represent uncontrollable events from the model. This results in a further reduction in the search space of the scheduling algorithm and an increase in its rate of convergence. The output of the algorithm is a partial sequence of transitions, representing controllable events, that guarantees the minimum cycle time for the plant operation. This sequence is used by the Petri net based controller to ensure optimal control.

To schedule a plant (definition 4.1), it is first modelled on a timed-place Petri net and the m resources $\{R_1, R_2, \ldots, R_m\}$ are modelled as PME or SME places, the processing time t_{ij} associated with operation O_{ij} is modelled as a time associated with the Petri net place representing operation O_{ij}. An algorithm to automatically generate the timed-place Petri net model from the scheduling information has been published in [Azzopardi and Lloyd 94b].

To illustrate the effect of the heuristics and reduction technique to the efficiency of the scheduling algorithm, consider a plant with two shared resources used by two jobs as shown in Fig. 4.5.

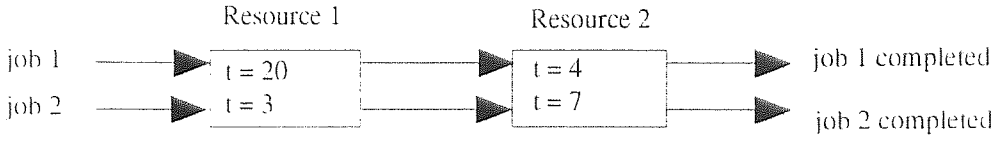


Fig. 4.5 A two resource plant

The matrices representing the scheduling problem are:

$$\mathbf{O} = \begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix} \text{ and } \mathbf{t} = \begin{bmatrix} 20 & 4 \\ 3 & 7 \end{bmatrix}$$

The plant is modelled on a timed place Petri net as shown in Fig. 4.6.

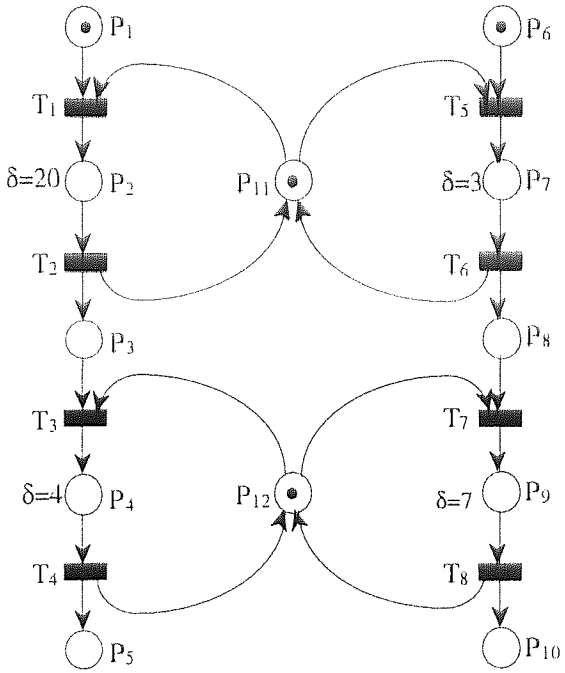


Fig. 4.6 A timed place Petri net model

where;

Place		Transition	
1	job 1 waiting for resource 1	1	Start processing job 1 in resource 1
2	job 1 is being processed by resource 1	2	Stop processing job 1 in resource 1
3	job 1 waiting for resource 2	3	Start processing job 1 in resource 2
4	job 1 is being processed by resource 2	4	Stop processing job 1 in resource 2
5	job 1 is Ready	5	Start processing job 2 in resource 1
6	job 2 waiting for resource 1	6	Stop processing job 2 in resource 1
7	job 2 is being processed by resource 1	7	Start processing job 2 in resource 2
8	job 2 waiting for resource 2	8	Stop processing job 2 in resource 2
9	job 2 is being processed by resource 2		
10	job 2 is Ready		
11	resource 1 available		
12	resource 2 available		

4.5.1 The branch and bound algorithm

In a Petri net based Branch and Bound algorithm [Shen *et al.* 92], the timed place Petri net model, Z_{timed} (defined in section 2.3.6.1), is executed by firing one enabled transition at a time. At every marking, m, where the set of enabled transitions, $ET \subseteq T$, contains more than one transition (i.e. #(ET)>1), only one is fired, whilst the others are stored as the set of alternative enabled transitions, $AT \subset ET$, at that level, i. This procedure is repeated until the marking, m, is equal to the required marking, m_r.

At the level where the required marking is reached, the cycle time for this sequence of events is recorded as the upper bound of the search and the sequence of transitions, S, is stored as the best firing sequence. Backtracking is then initiated up to the level, i, where alternative enabled transitions exist (i.e. $\#(AT_i)>0$). A new branch is formed by firing one of the alternative transitions. The Petri net is then executed in the manner described above until the required marking is reached again. If the cycle time is less than the upper bound, then this sequence of transitions, S, is stored as the best firing sequence and the upper bound is made equal to the new cycle time. Backtracking is initiated again and the process is repeated again, until all possible paths have been explored.

If at any time, whilst exploring a new path, the cycle time is larger than the upper bound, then this branch is abandoned and backtracking is initiated again. When all possible paths have been explored, the upper bound of the search is the minimum possible cycle time and the sequence of transitions that results in the minimum cycle time is the one that is stored as the best firing sequence. The Petri net based branch and bound algorithm is summarised below:

```
1   // Branch and Bound algorithm
2   WHILE (level≥0)
3
4      WHILE (#(ET)=0)
5
6            IF clock < upper bound
7                  find set of enabled transitions, ET_level
8                  increment clock
9
10           ELSE  back track
11                 // if current time ≥ upper bound this is not a good solution
12
13     select one enabled transition, t ∈ ET_level
14     add t to the sequence of transitions, S
15     save the set of alternative transitions AT_level
16     find marking, m, obtained by firing transition, t
17     store time at which transitions were enabled, time_level = clock
18
19     IF the marking, m, is equal to the required marking, m_r
20
21           IF clock < upper bound
22                 upper bound =clock;
23                 best firing sequence = S;
24                 back track
25
26
27  // back track algorithm
28  WHILE (#(ET_level) = 0)
29     level = level -1   // go back to level where #(ET_level) > 0
30
31  IF (level≥0)
32     clock=time_level      // set time
```

Applying the Branch and Bound algorithm to the timed Petri net model of the plant shown in Fig. 4.5, with initial marking m=[100001000011] and required marking m_r=[000010000111], results in the reachability tree shown in Fig. 4.8, with 29 possible markings being investigated. The optimal cycle time is 27 minutes which is obtained by using the transition firing sequence, $S = T_5T_6T_1T_7T_8T_2T_3T_4$. The optimal operation of the plant is illustrated in the Gantt chart of Fig. 4.7.

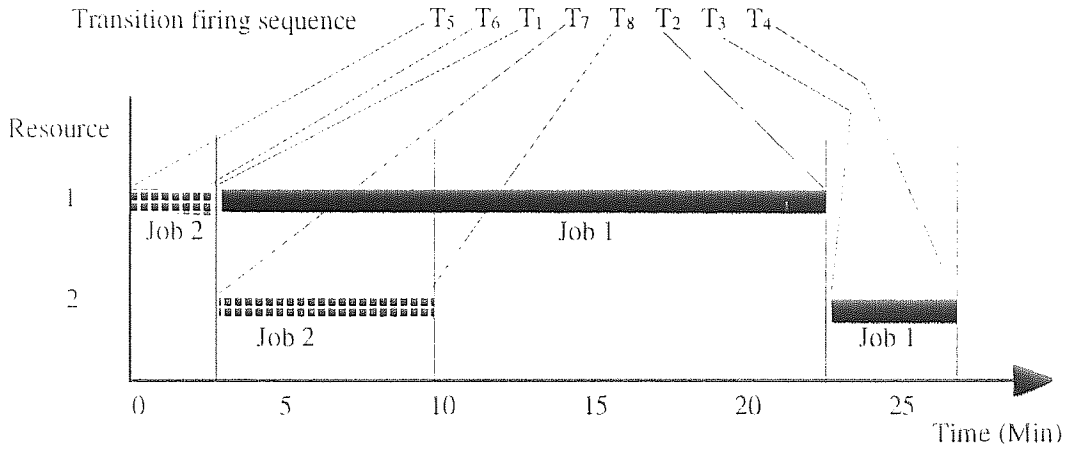


Fig. 4.7 Gantt chart showing optimal operation

Although the Branch and Bound algorithm results in an optimum solution, the processing time required for the algorithm to converge can be very large even for simple problems. In the following sections, methods for reduction of the search space which increase the rate of convergence of the algorithm are described.

4.5.2 Methods for reduction of the search space

In order to reduce the search space of the Branch and Bound algorithm, heuristics may be used to trim the reachable state space [Azzopardi and Lloyd 94a, Lee and DiCesare 94a]. To illustrate the reduction in search space that can be obtained by means of heuristics, consider the following heuristics that were developed by the author of this Thesis in [Azzopardi 93].

Heuristic 1: Detect concurrency

When an enabled transition can be fired concurrently with the fired transition the alternative transition can be removed from the set of alternative transitions at that stage. This reduces the number of branches that the algorithm needs to explore unnecessarily. The algorithm is summarised as:

```
1   IF t ∈ ET_level-1 AND clock = time_level-1
2           remove t from ET_level-1
```

Heuristic 2 : Look-ahead

In the search for the optimum schedule, at every new marking of the Petri net, the current time is compared to the upper bound of the search. If it is less, then the search along that branch continues, otherwise it is abandoned since it will surely not result in the optimum solution. To reduce the search space of the algorithm, a look-ahead heuristic [Shen *et al.* 92] is used. Looking-ahead is done by comparing the upper bound to the sum of the current time plus the remaining processing time of that production unit that still has the longest processing time yet to be performed . The algorithm is summarised below, where $t\_resource_k$ is the remaining processing time on resource k.

```
1       IF (clock + MAX(t_resource_k) > upper bound)
2               back track
```

Heuristic 3: Check for duplicated marking

This test checks for duplicated Petri net marking while exploring a new branch. It checks whether the new marking obtained by firing the current transition is the same as the marking in the next step of the current minimum solution. If this occurs, it means that there are different ways to arrive from to the duplicated state from the initial state of the plant and that working down this branch will result in the same path taken from the duplicated state to the final state as in the current minimum solution [Shen *et al.* 92].

Therefore, if the time required to reach the duplicated state in the current search is greater than or equal to the time required to reach this state in the current minimum solution, it will be useless to examine this route further since it will surely not lead to the optimum solution. The algorithm is illustrated below:

```
1       IF (m = m_r  AND clock ≥ time_level )
2               back track
```

Illustrating the reduction in search space

The effect of these heuristics to the search space of the Petri net model of Fig. 4.6 is illustrated in Fig. 4.8. The shaded areas show those areas that are not explored by the algorithm due to the reduction in search space.

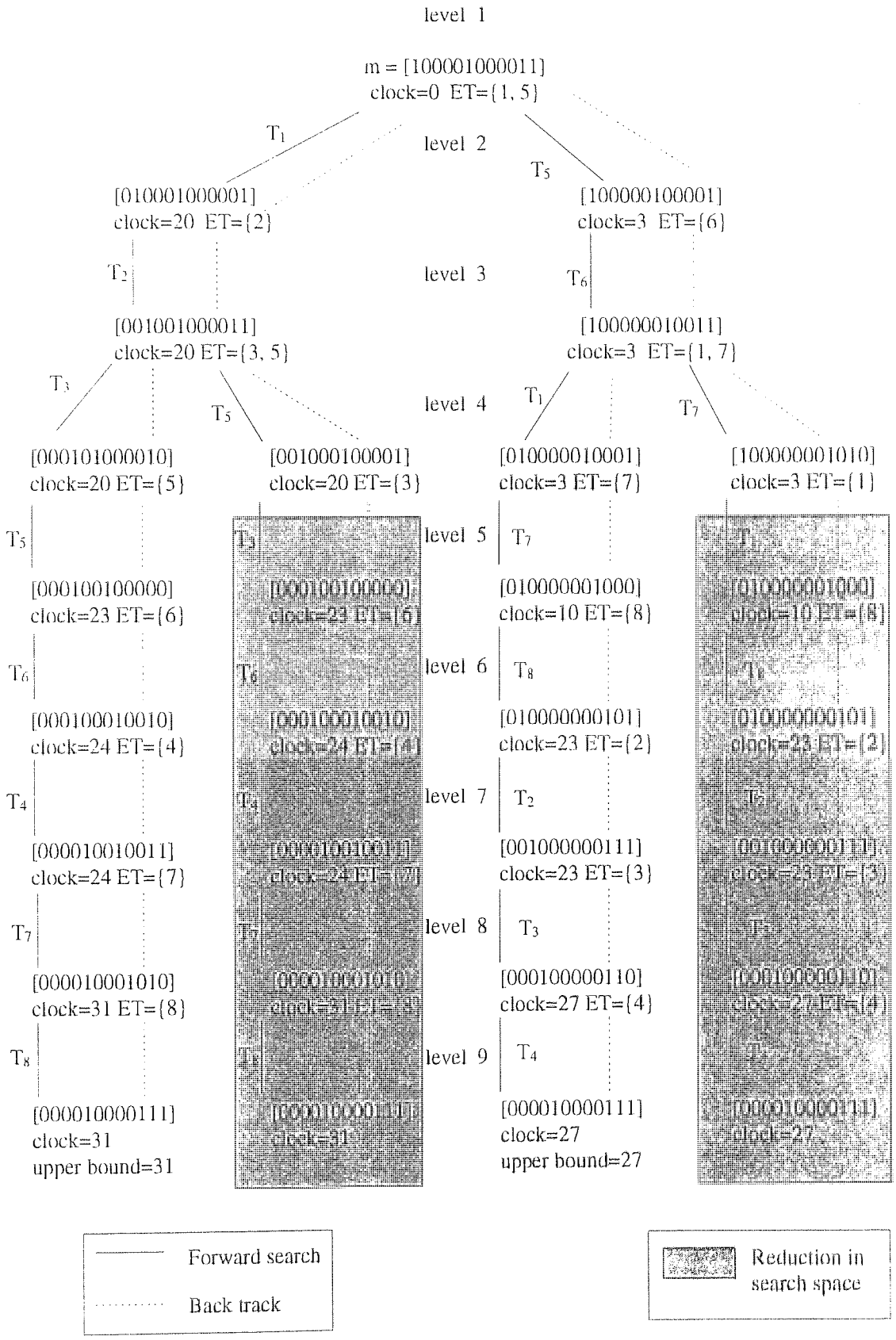


Fig. 4.8 Search space for scheduling algorithm

4.5.3 Petri net reduction

As discussed in Section 4.2.2, in DEDSs events are divided into the set of controllable events and the set of uncontrollable events. For the purpose of scheduling DEDS operation, it is not necessary to consider the uncontrollable events in the system, as, by definition, the controller cannot influence their occurrences. In the Petri net described in Fig. 4.6, the transitions that represent controllable transitions are those that are inputs to places representing processes (i.e. T_1, T_3, T_5 and T_7). The uncontrollable events are represented by the transitions that are outputs from the places representing processes (i.e. T_2, T_4, T_6 and T_8).

In a timed Petri net that uses PME places to represent shared resources, the transition indicating the start of an operation represents a controllable event, whereas the one that represents the end of an operation is not. Therefore, by applying reduction rule R1 (described in Section 2.3.5.1) to the operation places, the transitions representing uncontrollable events are removed from the Petri net model. This is illustrated in Fig. 4.9 where the dotted lines show the places, transitions and arcs that are removed by reducing the net. This reduction can be visualised as the collapse of an input transition, operation place and output transition into one transition.

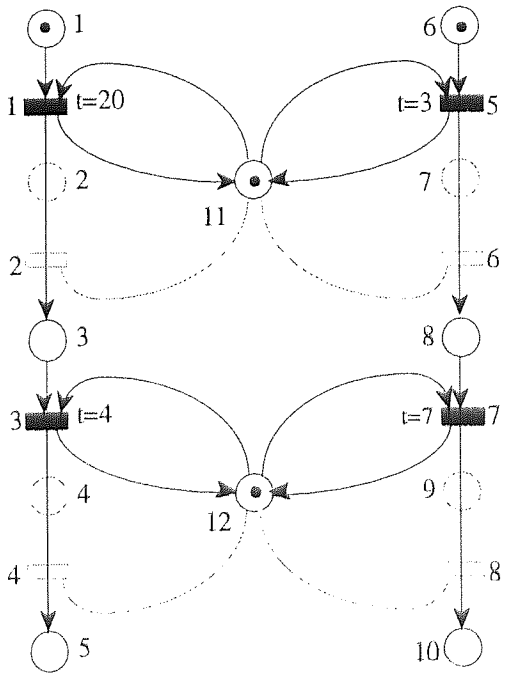


Fig. 4.9 The reduced timed-transition Petri net

Since the operation places, which have been absorbed into a transition, have a time associated with them, the delay is now transferred to the transition. This means that the resulting Petri net is a timed transition Petri net [Sifakis 78].

Fig. 4.10 illustrates the application of the Branch and bound algorithm with heuristics to the reduced Petri net model (i.e. the timed transition Petri net). The shaded areas illustrate the reduction in search space obtained by the heuristics defined in Section 4.5.2.

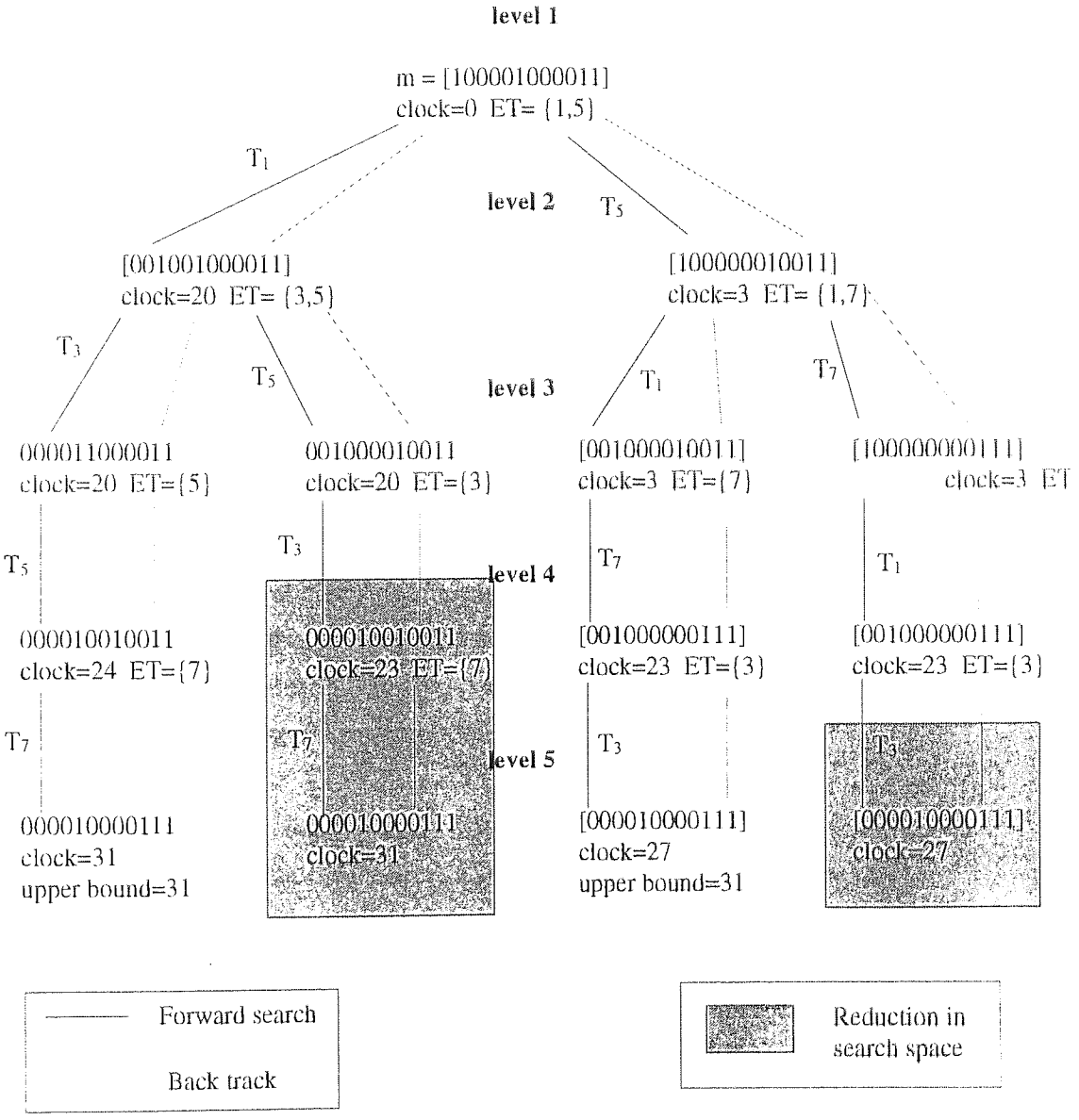


Fig. 4.10 State space of reduced net

By reducing the Petri net and applying the Branch and Bound algorithm with heuristics, the state space has been reduced from the 29 states required by the Branch and Bound algorithm (Fig. 4. 8) to 12 states (fig. 4.10).

The following section reports on a series of experiments that were performed to quantify the reduction in search space and to observe the rate of convergence of the scheduling algorithms on larger plants.

4.5.4 Performance evaluation of the scheduling algorithm

Scheduling problems are difficult combinatorial optimisation problems, therefore the efficiency of a scheduling algorithm is very important for its practical use. Efficiency refers to the computational resources necessary to obtain a solution. For simple algorithms, complexity may be represented using mathematical expressions however, for more complex algorithms like the one developed in this Thesis, computational experimentation is required to quantify complexity [MacCarthy and Liu 93].

In order to evaluate the gain in performance of the algorithm presented in the previous section, compared to the algorithm presented in [Azzopardi and Lloyd 94a] and to the Branch and Bound algorithm, the following experiment was performed:

A computer program was written to generate random scheduling problem (Definition 4.1) in terms of matrices **O** and **t**, which represent the routing information and processing times, for a plant with a given number of jobs and resources. Another program, based on the Petri net modelling algorithm developed by the author in [Azzopardi and Lloyd 94b] then converted this information into a timed place Petri net. The following algorithms were then applied to the Petri net model:

Algorithm A Branch and Bound
Algorithm B Branch and Bound AND 3 heuristics
Algorithm C Petri net reduction AND Branch and Bound AND 3 heuristics

For each algorithm, the number of times a new marking had to be found was recorded. This reflected the number of iterations that the algorithm had to perform in order to converge and is directly proportional to the CPU time that is required. To investigate the improvement of Algorithm C and Algorithm B over the standard Branch and Bound algorithm, the experiment was performed on scheduling problems with different numbers of resources, jobs and processing times.

For a plant with a given number of shared resources, it was observed that number of states that the Branch and Bound algorithm generated increased very rapidly as the number of jobs increased. This occurred to a lesser extent for Algorithm B and to an even lesser extent for algorithm C. The state space that has to be investigated by the three algorithms applied to a plant with 2 shared resources in Fig. 4.11 on a linear scale and in Fig. 4.12 on a logarithmic scale to reveal more information about the whole range of values. For each size of plant an average over 10 randomly generated scenarios were taken.

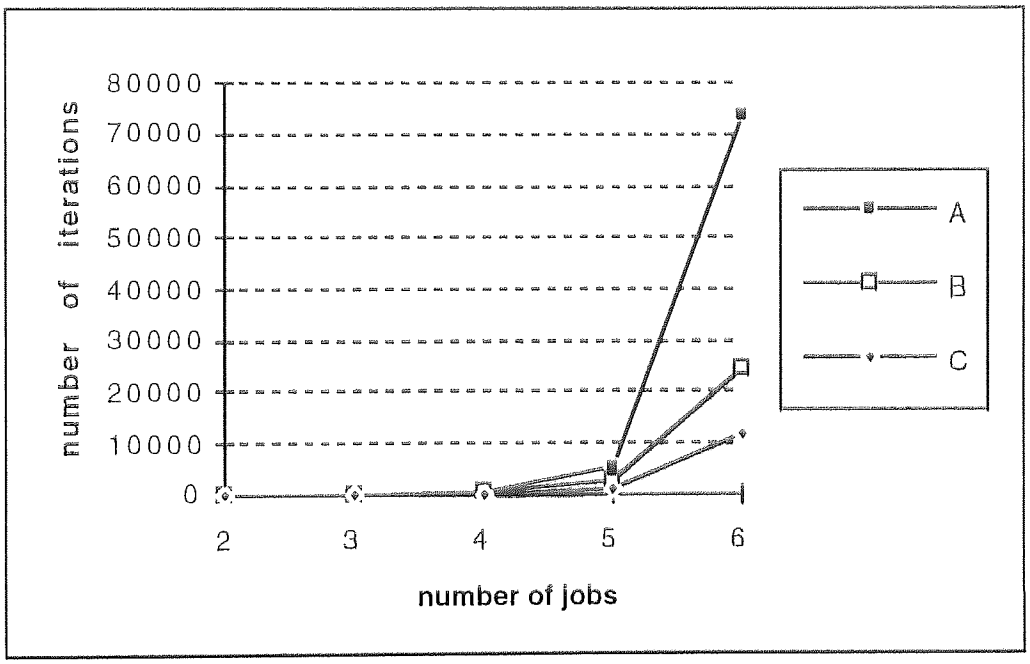


Fig 4.11 Scheduling 2 resources

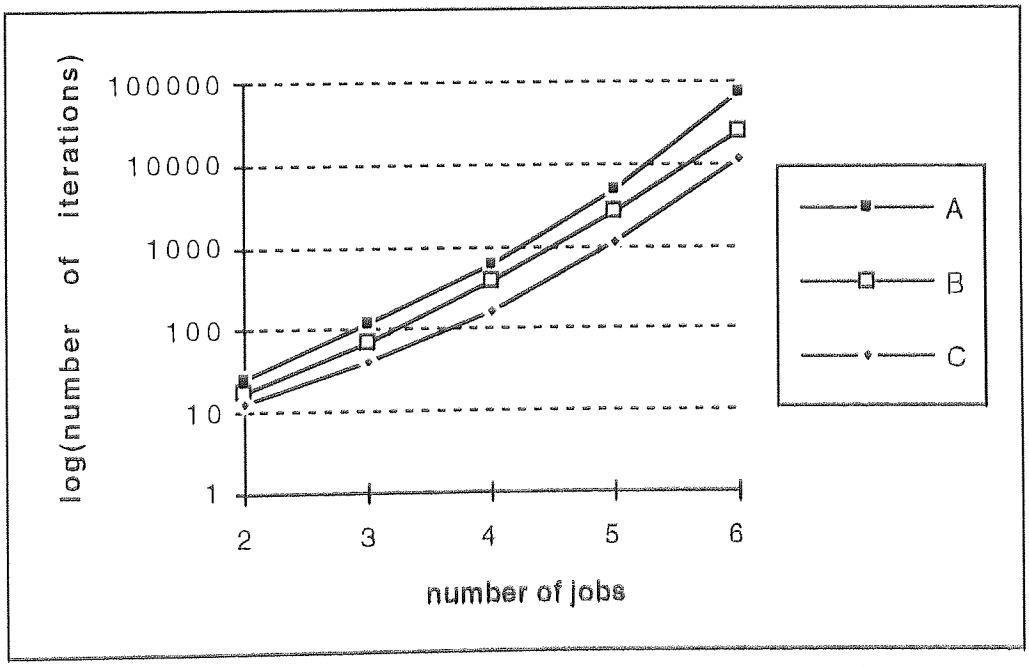


Fig 4.12 Logarithmic scale for the number of iterations

The results showing the number of iterations required by the three algorithms on each of the 220 different scheduling problems are tabulated in Appendix A. The following graphs (Fig. 4.13 to 4.18) show the improvement obtained by Algorithms B and C over Algorithm A when applied to plants with up to 7 shared resources. The x-axis represents the number of jobs and the y-axis represents the improvement factor for Algorithms B and C over Algorithm A. The following legend was used in the graphs:

$$\frac{\text{Number of iterations required by Algorithm A}}{\text{Number of iterations required by Algorithm B}}$$

$$\frac{\text{Number of iterations required by Algorithm A}}{\text{Number of iterations required by Algorithm C}}$$

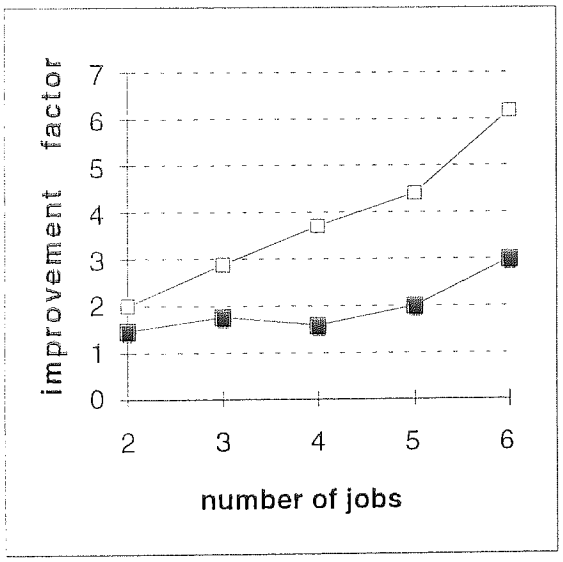


Fig. 4.13 Scheduling 2 resources

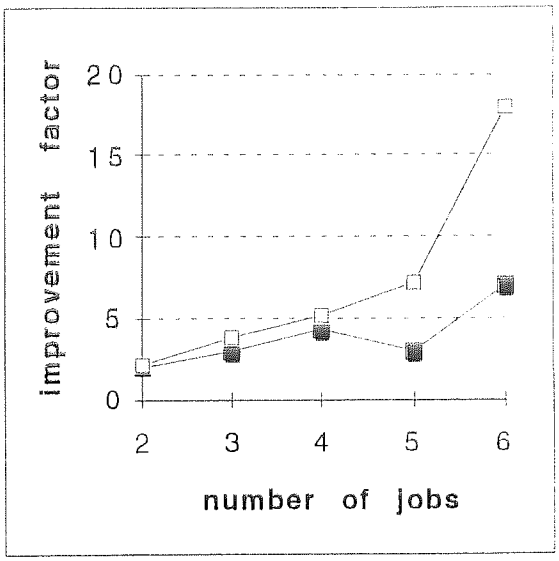


Fig. 4.14 Scheduling 3 resources

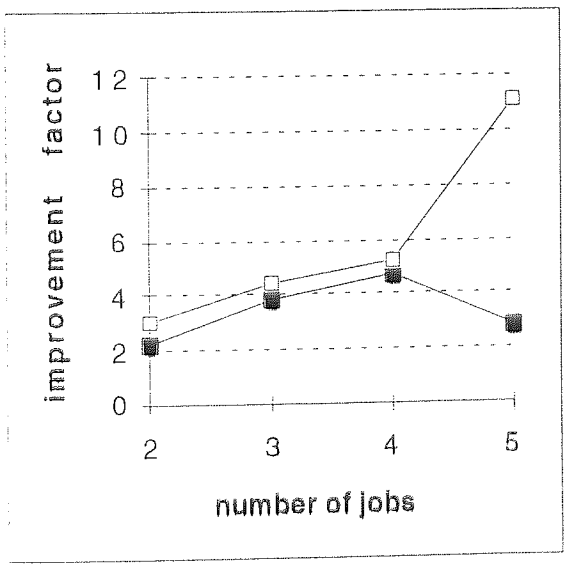


Fig. 4.15 Scheduling 4 resources

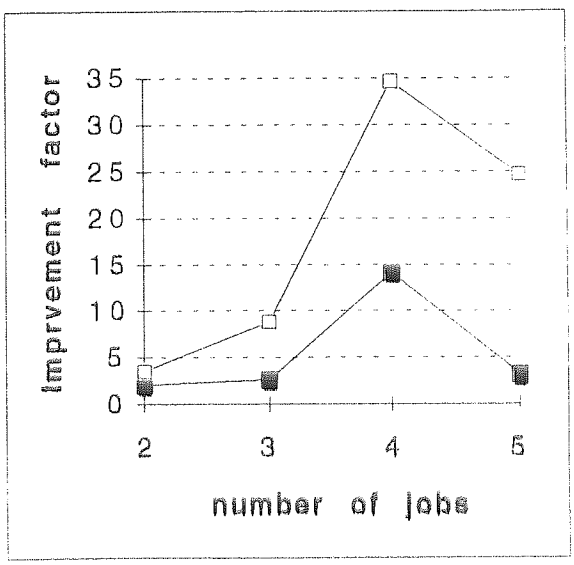


Fig. 4.16 Scheduling 5 resources

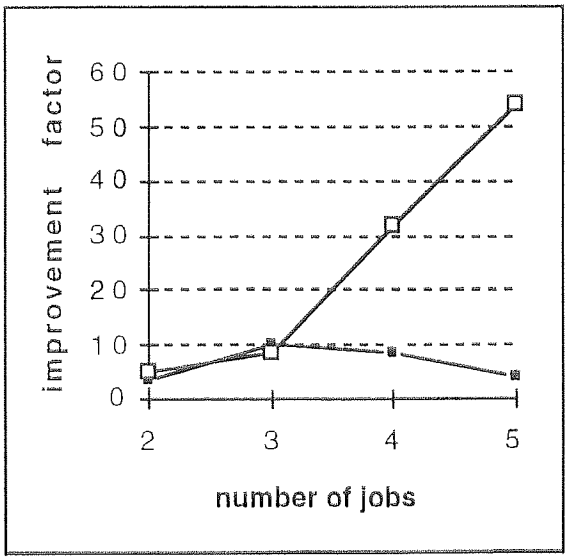


Fig. 4.17 Scheduling 6 resources

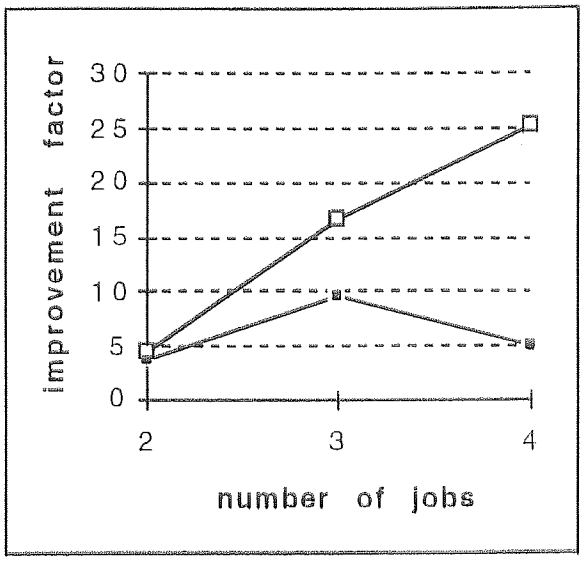


Fig. 4.18 Scheduling 7 resources

From these graphs it can be seen that both Algorithm B and Algorithm C are more efficient than Algorithm A for all the scenarios that were investigated. It can also be seen that the improvement due to Algorithm C increases at a faster rate than that due to Algorithm B, as the complexity of the plant increases. This means that Algorithm C, that was developed in this Thesis, is applicable to larger DEDS and converges several times faster than previously published scheduling algorithms.

4.6 Design of a re-configurable supervisory controller

In industrial DEDSs, shared resources may break down at unknown intervals. In such a situation, jobs must be re-routed to make use of the available shared resources. In the event of a breakdown, the Petri net model within the supervisory controller would not be a correct representation of the plant unless all possible failures are included in the model. If it were possible to model all the possible failures, this would result in a much more complex Petri net because of the additional states that need to be introduced, resulting in high CPU and memory requirements for its analysis.

To be able to respond to both the changes in the set-up and incorrect response of the system, this Thesis proposes a re-configurable Petri net model-based controller first published by the author of this Thesis in [Azzopardi and Holding 95]. The controller uses state feedback (Fig. 4.19), to detect both changes in the plant configuration and incorrect response of the system.

The proposed controller combines the Petri net synthesis technique based on modified OMT (presented in Chapter 3), a Petri net based scheduling algorithm [Section 4.5.3] to ensure optimal operation, a token-player module [Valette *et al.* 85], a manager module and a database containing the OMT specification of the plant, in the form of a hierarchical controller illustrated in Fig. 4.20.

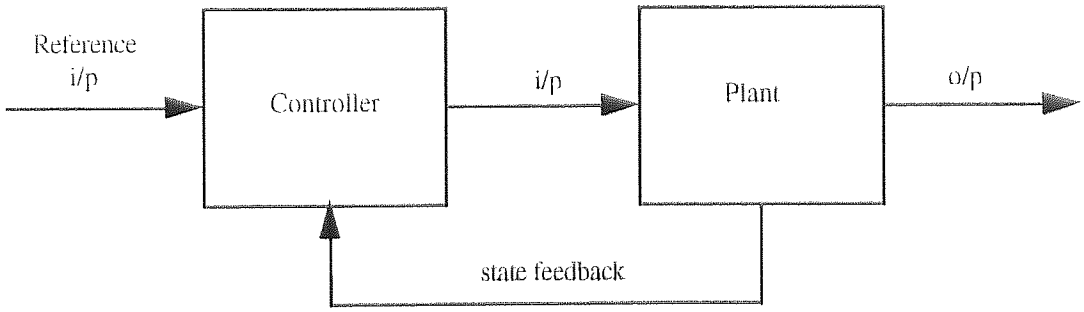


Fig. 4.19 Closed loop control set-up

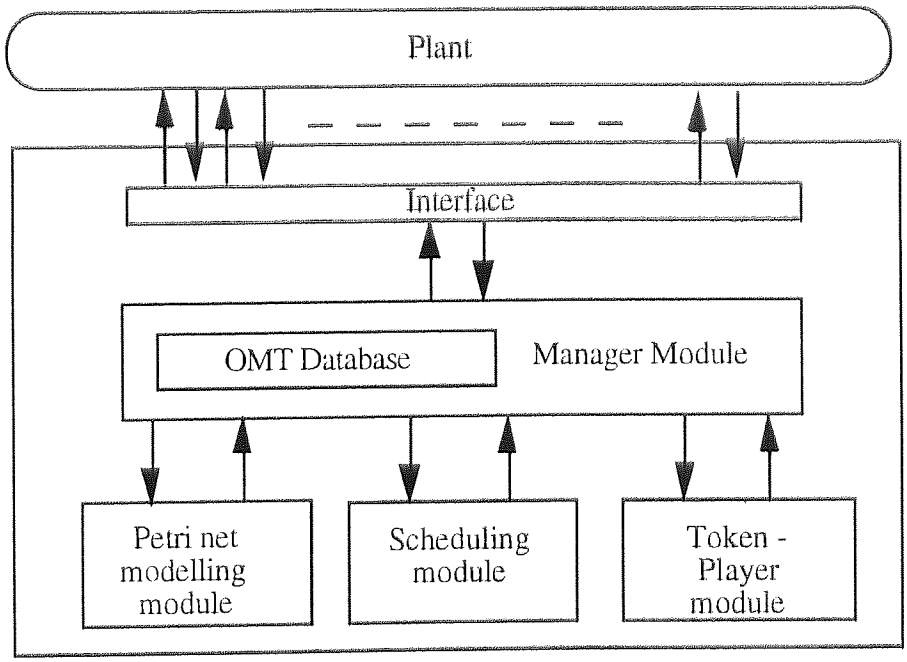


Fig. 4.20 Structure of re-configurable controller

The token-player module controls the plant operation by sending the required signals to the plant to trigger controllable events, in response to state feedback signals received from the plant as shown in Fig. 4.4. If the controller detects a response that is different to the expected one, it is able to re-schedule the plant operation. When the controller detects a change in the plant configuration (via state feedback), it can re-synthesise the Petri net model to reflect the change. The re-scheduling and re-modelling features of the proposed controller are described in the following sections.

4.6.1 Re-scheduling

In practice, there may be situations where actual processing times vary from their estimated value, or where resources are unavailable for a short period of time (down time). In these situations, the response of the plant will differ to that expected by the controller, so the manager module would re-schedule the plant to keep the plant operation optimal. Re-scheduling the operation of a plant was suggested by Chang and Liao [94] (Fig. 4.21) who used a re-scheduling strategy for timely adjusting of the planned schedule of an FMS to cope with disturbances in the plant. In this Thesis the idea of re-scheduling was adopted and included as one of the facilities of the re-configurable controller (Fig. 4.10).

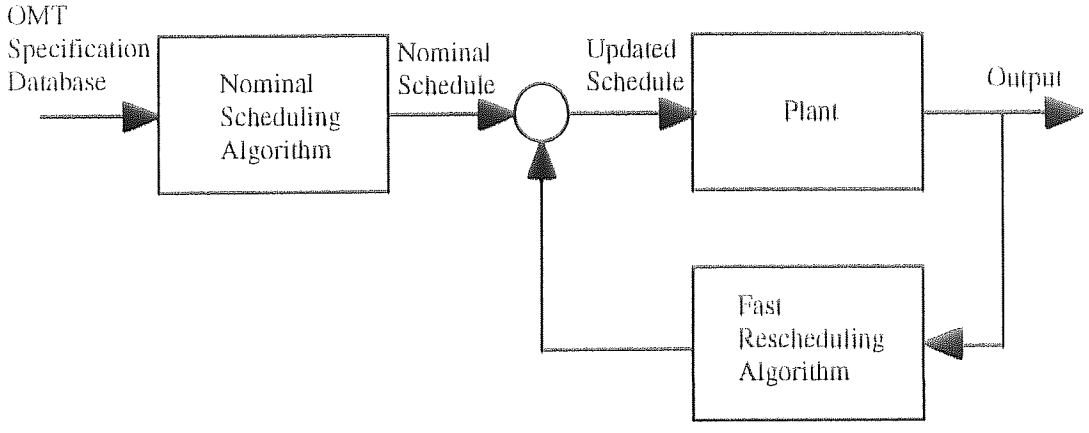


Fig. 4.21 Re-scheduling function of the re configurable controller

Re-scheduling is implemented by the Petri net based scheduling algorithm described in Section 4.5.3 of this Thesis. The manager module passes the current state of the plant, in the form of a marking vector, to the scheduling algorithm, which in turn calculates the optimal schedule for the plant operation. The resulting partial sequence of transitions is used as the control law for the token player algorithm and ensures optimal operation of the plant.

4.6.2 Re-configuration

In a situation where part of the system under control fails unexpectedly, the controller detects this change through the state feedback and re-synthesises the Petri net model and analyses it for the behavioural properties that are specified in the OMT analysis model. Re-configuration is done by using the OMT-based Petri net synthesis technique [Section 3.8], the OMT analysis information stored in the database and the state feedback information.

The new Petri net model has to be analysed and the exclusive-use resources have to be re-scheduled to ensure correct and optimal operation. The flow chart representing this behavioural aspect of the manager module is illustrated in Fig. 4.22.

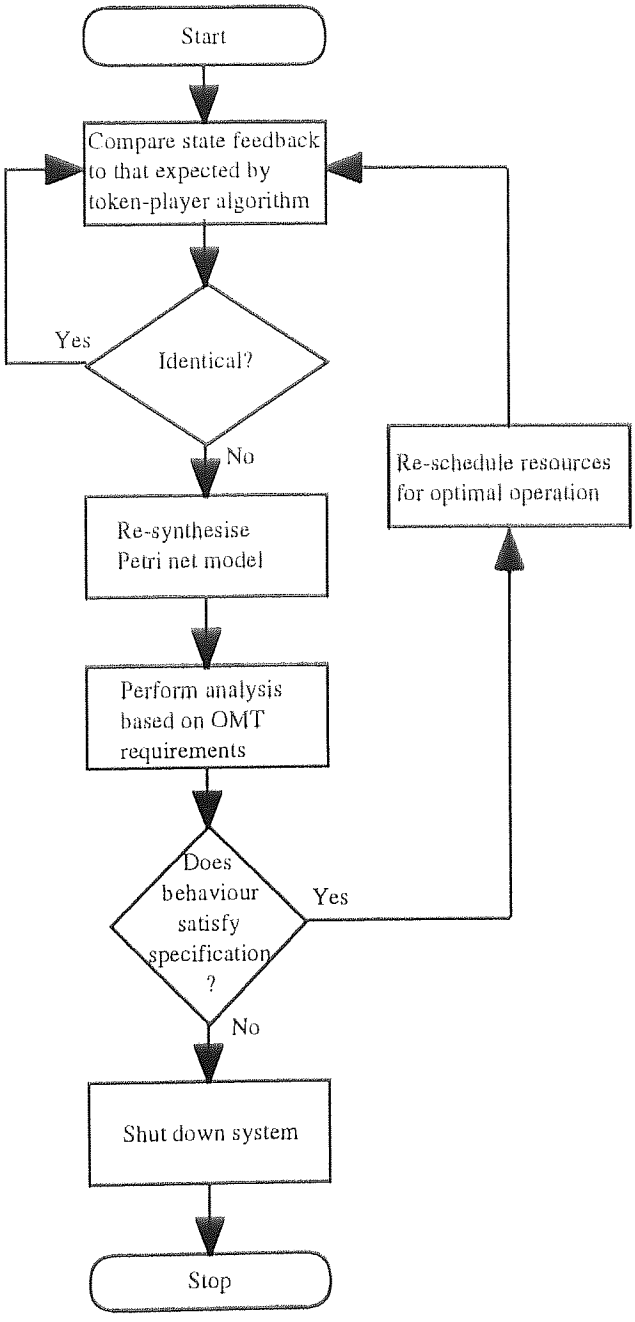


Fig. 4.22 Manager module re-configuration decision

To illustrate the significance of the re-modelling facility of the controller, consider a section of an FMS production path illustrated in Fig. 4.23, with a corresponding Petri net representation as shown in Fig. 4.24, with the significance of the places and transitions listed in Table 4.1. Assume that Unit 1 and Unit 2 are both identical production units.

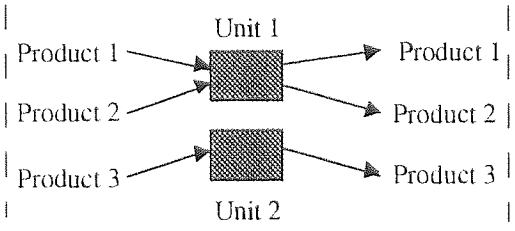


Fig 4.23 Section of a production plant

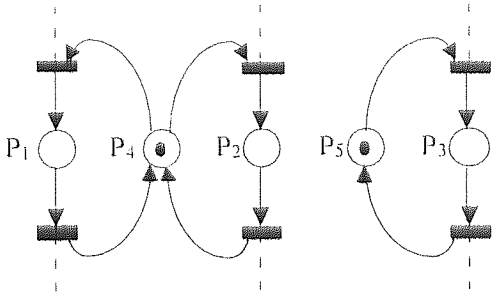


Fig. 4.24 Petri net representing section of production plant

Place		Place	
P1	Product 1 is processed in Unit 1	P4	Unit 1 is available
P2	Product 2 is processed in Unit 1	P5	Unit 2 is available
P3	Product 3 is processed in Unit 2		

Table 4.2 Significance of places and transitions

If, say, Unit 2 were to break down, the production material would have to be re-routed so that the three products are processed by Unit 1, as shown in Fig. 4.25. This would be achieved by modifying the Petri net by removing the token from place P_5 and re-drawing the arcs so that product 3 is processed by Unit 1, as shown in Fig. 4.26. Obviously re-configuration is not always possible; In the case of breakdown of a vital component of the system, where re-configuration is not possible (or not permitted), the controller would have to shut down the plant.

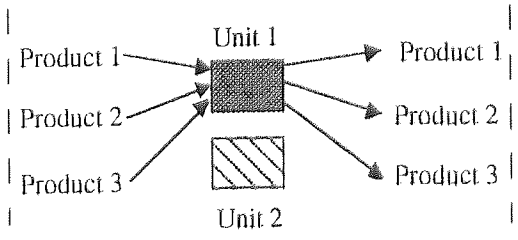


Fig. 4.25 Production Unit 2 breaks down

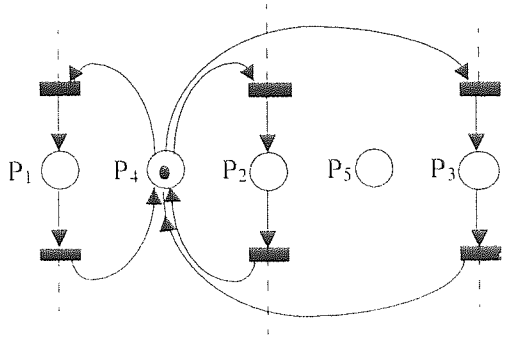


Fig. 4.26 Modified Petri net

4.7 Conclusion

This Chapter has introduced traditional control theory for DEDSs where the supervisory controller is based on a state automaton model of the plant. Extensions to this work were described where the supervisory controller was based on a Petri net and was driven by a token-player algorithm. It was then shown how exclusive use resources can be scheduled by using Petri net based scheduling algorithms in order to ensure optimal control. A novel scheduling algorithm based on Petri net reduction techniques was introduced and experimental evidence shows that it has a rate of convergence that is several times faster than previously published algorithms. Finally, a re-configurable controller was proposed to cater for unexpected changes in the plant. The advantages of the proposed controller over previously published controllers is that;

- Based on the OMT model, it can automatically re-configure the Petri net supervisor to accommodate changes in the plant.
- It can automatically re-schedule the plant operation in the case of a fluctuation in process times or resource breakdown.
- The fast rate of convergence of a Petri net based scheduling algorithm makes it feasible to perform on-line re-scheduling where this is allowed by the physical construction of the plant.

In the following Chapter, an object oriented C++ Petri net class is developed to enable analysis of large Petri nets and to enable the implementation of the scheduling algorithm and token-player algorithm that were described in this Chapter.

Chapter 5

Design of a Petri net software tool kit

5.1 Introduction

This chapter presents the design of a Petri net software tool kit that was implemented in object oriented C++ [Stroustrup 94] to: (i) analyse the behavioural and structural properties of a DEDS; (ii) to control the DEDS by means of a token-player algorithm and; (iii) to schedule the allocation of exclusive-use resources. The class structure of the software and some of the more important algorithms are described. A complete list of functions that have been implemented in the software tool kit are listed in Appendix B. The application of the software tool kit is illustrated by the implementation of a Petri net analysis tool.

5.2 The Petri net class

A Petri net is a five-tuple, Z = (**P**, **T**, **I**, **O**, **m**) [Definition 2.3] which can be represented by a pre-condition matrix, $\mathbf{H}^-$, a post-condition matrix, $\mathbf{H}^+$ and an initial marking vector, $\mathbf{X}_0$, where;

- The pre-condition matrix represents the input function **I**. $\mathbf{H}^-[i,j]=1$ means that place j is an input place to transition i.
- The post-condition matrix represents the output function **O**. $\mathbf{H}^+[i,j]=1$ means that place j is an output place from transition i
- $\mathbf{X}_0[j]=n$ means that in the initial marking of the net, there are n tokens in place i

Hence, the Petri net data structure was implemented in C++ as:

```
struct pnet{
        matrix   H_pre, H_post;     // pre & post condition matrices
        matrix   C;                 // Incidence matrix
        vector   X0;                // initial marking
        int      p,t;               // number of places & transitions
        int      n;                 // reference count
} *pn;
```

where;

- H_pre and H_post are the pre-condition and post-condition matrices
- C is the incidence matrix [Section 2.3.4]
- X0 is the initial marking vector
- p=#(**P**) and t=#(**T**) are constants representing the number of places and transitions in the net
- n is a variable that records the number of instances of the Petri net class

To implement the data structure shown above, it was necessary to implement a matrix and vector class in C++ to provide the respective data structures and operations associated with vector and matrix algebra, since these are not provided as standard features of C++.

To implement algorithms which generate concurrency sets, find deadlock states and observe transition sequences, it is necessary to generate the coverability graph of the Petri net. In order to handle this information it was necessary to implement a reachability tree class*. This class required the implementation of an integer set class* to provide the data structure for, and allow operations on sets of integers. The classes comprising the Petri net tool kit, together with their associations are illustrated in Fig. 5.1.

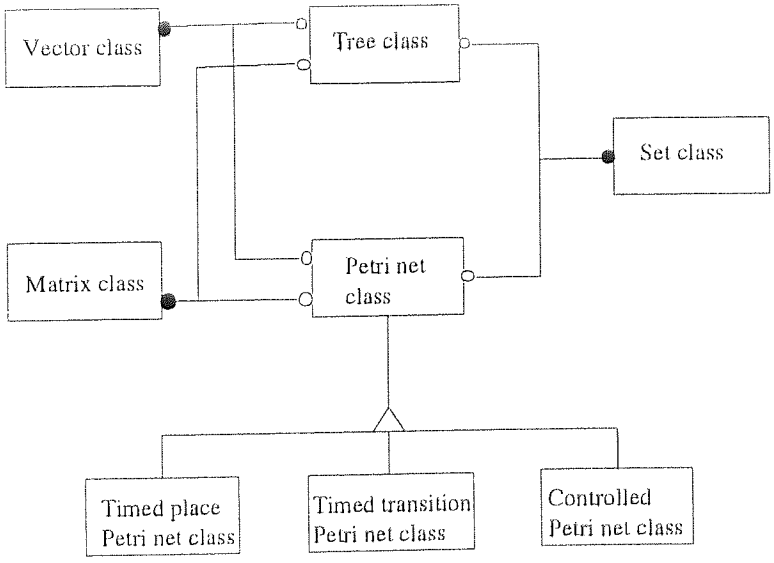


Fig. 5.1 Classes and their associations

The following sections describe the more important functions that are associated with the Petri net class.

* These classes were implemented by Dr. J. Jiang of the IT Research group, Dept. EEAP, Aston University.

5.2.1 Find the set of enabled transitions

A Petri net transition is enabled [Definition 2.4] when its input places are full. Therefore, the following algorithm, **find_enabled_transitions(M)**, was designed to find the set of enabled transitions of a Petri net.

$$\forall t \in \mathbf{T}$$
$$\left(\forall p \in \mathbf{P}: \mathbf{H}^{-}[t,p] = 1, \mathbf{M}[t,p] > 0\right) \rightarrow t \in \mathbf{ET}$$

where, $\mathbf{M}$ is the marking vector, and $\mathbf{ET}$ is the set of enabled transitions

5.2.2 Find the new marking

The new marking, $\mathbf{M}_{new}$ of a Petri net resulting from firing transition, $t \in \mathbf{T}$, at marking $\mathbf{M}$ is obtained by removing a token from each of its input places and adding a token to each of its output places. Hence, the following algorithm, **find_new_marking(t)**, to find the new marking by firing transition t;

$$\forall p \in \mathbf{P}, \mathbf{M}_{new}[p,t] = \mathbf{M}[p,t] + \mathbf{H}^{+}[p,t] - \mathbf{H}^{-}[p,t]$$

5.2.3 Generating the coverability graph

To generate the coverability graph of a 1-bound Petri net, the following algorithm, based on a depth-first strategy [Cormen *et al.* 93] was developed and incorporated into the Petri net C++ class:

1. **find_enabled_transitions(M)**
2. select one enabled transition, $t \in \mathbf{ET}$
3. **find_new_marking(t)**
4. If a place is not 1-bounded, represent this by an 'ω'
5. Marking is a repeated marking $\rightarrow$ go to step 8
6. Save marking and set of enabled but unfired transitions
7. go to step 1
8. Back track to a marking where there are enabled but unfired transitions
9. go to step 2

5.2.4 Calculating the P- and T- invariants

To calculate P and T-invariants, an efficient algorithm, presented and proved correct by Martinez and Silva [82]. This finds all invariants for an ordinary Petri net with n places, m transitions and an incidence matrix **A** [Section 2.3.4], by solving the simultaneous equations defined in [Section 2.3.4.1]. The algorithm was implemented as one of the functions of the C++ Petri net class.

Step. 1
- Let **B** = identity matrix of dimension n x n, **A** = the incidence matrix. Construct the matrix [**B**|**A**]

Step 2

For each index j of transition T_j
- Add to matrix [**B**|**A**] as many rows as there are linear combinations of two lines, in which the element in column j is greater than zero.
- Eliminate from matrix [**B**|**A**] the rows whose element in column j is not zero.

Step 3
- The P-invariants correspond to the non-zero rows of B

To illustrate the algorithm, consider its application to the example shown in Fig. 5.2:

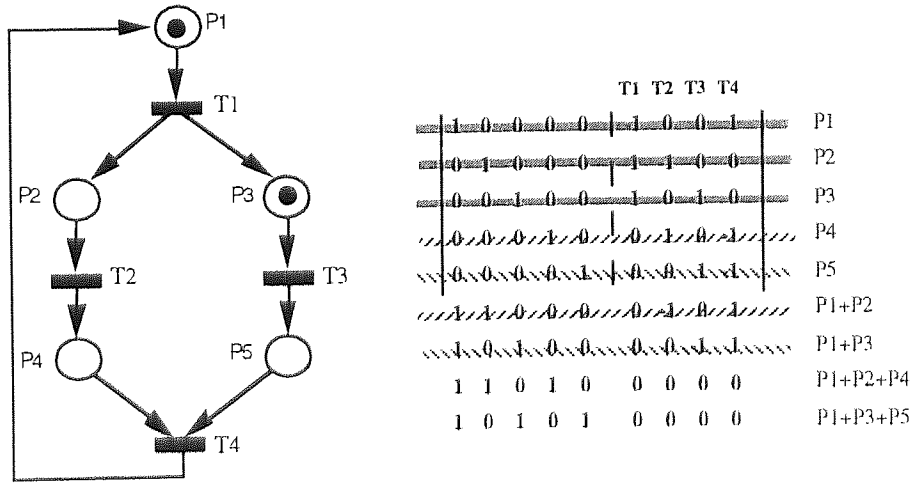


Fig. 5.2 Illustrating the algorithm to find P-Invariants

The matrix $[\mathbf{B}|\mathbf{A}] = \left[\begin{array}{ccccc|cccc} 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -1 \end{array}\right]$

To eliminate the elements in the column representing T_1 we add rows 1 and 2, and rows 1 and 3 to form two new bottom rows and delete the 1st three rows to get:

$$\left[\begin{array}{ccccc|cccc} 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -1 \\ 1 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 1 \end{array}\right]$$

To eliminate the elements in the column representing T_2: we add rows 1 and 3 to form a new bottom row and eliminate rows 1 and 3 to get:

$$\left[\begin{array}{ccccc|cccc} 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

Finally, to eliminate the elements in the columns representing T_3 and T_4 we add rows 1 and 2 to form a new bottom row and delete rows 1 and 2 to get:

$$\left[\begin{array}{ccccc|cccc} 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}\right]$$

Using this result and the initial marking we conclude that the P-invariants are $P_1 + P_2 + P_4 = 1$ and $P_1 + P_3 + P_5 = 2$. This matches the results obtained in [Section 2.3.4.3] where reduction methods were used to obtain the P-invariants of the Petri net illustrated in Fig. 5.1. The algorithm to calculate the T-invariants operates in a similar fashion to the one that calculates the P-invariants except that **B** = identity matrix of dimension m x m, **A** = the transpose of the incidence matrix. The algorithm operates to eliminate the elements in the columns representing the places of the Petri net.

5.3 Petri net extensions using inheritance

When using object oriented techniques, one can define a new class, starting from another class, by means of inheritance [Booch 94]. The inherited class contains all the attributes and operations of the super class but has some additional attributes and operations. Therefore inheritance makes it possible to re-use previously designed classes.

By using inheritance, it is possible to re-use the Petri net class [Section 5.2] to implement classes of Petri net extensions (e.g. timed Petri nets). This is illustrated in Fig. 5.1 and demonstrated in [Section 5.3.1] and [Section 5.3.2], where two new classes representing timed-place Petri nets [Sifakis 78] and controlled Petri nets [Krogh 87] are designed.

5.3.1 Timed-place Petri net

As defined in Section 2.2.2, a timed-place Petri net is a six-tuple, Z = (P, T, I, O, θ, m) where P, T, I, O, m have the same meaning as in the definition of an ordinary Petri net and θ: P $\rightarrow$ $\Re^+$ is the delay vector, whose *i*th component represents the time associated with the *i*th place.

From this definition, it can be seen that a timed place Petri net has the same data structure as an ordinary Petri net but has an additional vector, θ. Therefore, by using the inheritance facility of object oriented C++, the timed place Petri net (TPPN) class was derived from the Petri net class. Hence, the TPPN data structure is:

```
class timed_place_petrinet:public petrinet{
        struct tppnet{
                vector tp;          //delay vector
                int n;              //reference count
        } *tppn;
}
```

Some of the functions of the TPPN class, such as those to find the P- and T- invariants are identical to those of the Petri net class. An advantage of using inheritance is that these functions can be used by the inherited class and therefore it is not necessary to re-define them. These functions are called "virtual functions" [Stroustrup 94]. On the other hand, some functions, such as those to find the list of enabled transitions and generate the reachability graph. need to be re-defined because the condition for a transition to be enabled in a TPPN is different to that in an ordinary Petri net.

In a TPPN, a transition t $\in$ T is enabled iff $\forall$p $\in$ I(t), m(p) > 0 for a time, $\delta > \theta(p)$., therefore this algorithm was implemented as one of the functions of the TPPN class.

5.3.2 Controlled Petri net

As defined in Section 4.3, a controlled Petri net (CPN) is a five-tuple $Z_C = \{P, T, \xi, \chi, \beta\}$, where P is the set of places, T is the set of transitions, $\xi = (P \times T) \cup (T \times P)$ is the set of directed arcs connecting places and transitions, χ is the finite set of control places, represented by squares and β is the set of arcs associating control places with transitions. Therefore the data structure of a CPN is equivalent to that of an ordinary Petri net with an additional set of control places and set of arcs that link the control places to the controlled

transitions. This information can be represented in the form of a vector, $\mathbf{B}$, where $\mathbf{B}_i$ contains the transition number to which the ith control place is linked to. Hence the CPN data structure has been implemented in C++, as:

```
class controlled_petrinet:public petrinet{
        struct cpnet{
                vector B;          // control place connection
                int n;             // reference count
        } *cpn;
}
```

5.4 The reachability tree class

To analyse the dynamic behaviour of the Petri net using reachability tree or coverability graph analysis [Section 2.3.3], it is necessary to provide a data structure that can handle this information and provide the analysis facilities. This was done by implementing a reachability tree class which consists of two data structures. One that records the reachability tree information, including the number of nodes and their interconnections, whilst the other data structure records the node information, including the marking, a list of enabled transitions, preceding and succeeding nodes.

The tree data structure and the node data structure, implemented in C++ are shown below:

```
class tree      {
Tree_node       *rt;
int             counter;        // reachability tree node counter
int             size;           // keep the size of allocated memory
...
```

```
struct  rtree_node
{
        vector  marking;
        Int_Set enabled_trans;  // set of enabled trans
        int     pre, post;      // no of pre & post nodes
        vector  pre_nodes, pre_trans;
        vector  post_nodes, post_trans;
        int     n;              // reference index
}       *node;
```

5.4.1 Deadlock states

The algorithm to find the set of states where deadlock occurs, investigates the nodes of the reachability tree and checks whether any transitions are enabled at that node. If no transitions are enabled, then that node represents a deadlock state. Hence, the algorithm:

```
void    reach_tree :: dead_lock_markings() const
{       for (int i=1; i<=counter; i++)
                if (rt[i].enabled_trans().is_empty()==TRUE)
                {       cout << "\n\tMarking" << rt[i].n() <<"=";
                        (rt[i].marking().vec2set()).set_print();
                }
}
```

5.4.2 Concurrency sets

Concurrency sets [Skeen and Stonebraker 83] were introduced to verify the correctness of commit protocols in a distributed database system. Hill and Holding [90] extended the application of concurrency sets to protocols modelled using Petri nets and were used by Sagoo [92] to analyse Petri net models of hard real-time systems and by Jiang *et al.* [96] to analyse the safety properties of a multi-axis high-speed industrial machine.

The concurrency set of a place $p_i \in P$ is the set of places that can be marked concurrently with p_i. The algorithm to find the concurrency set of a Petri net place involves investigation of the states in the reachability tree, hence the algorithm:

```
//      calculate the concurrent set for the given place
Int_Set reach_tree :: concurrency_set(const int place_index) const
{
        Int_Set  conc_set;
        for (int i=1; i<=counter; i++)
                if ((rt[i].marking().val(place_index-1))!=0)
                        conc_set=conc_set + (rt[i].marking()).vec2set();
        conc_set.element_delete(place_index);
        return conc_set;
}
```

5.5 Using the Petri net tool kit facilities

The C++ Petri net tool kit is a C++ library that allows handling of Petri net, Petri net extensions and reachability tree objects. This section describes the principle functions that have been implemented and included in the Petri net tool kit. It is not feasible to describe all the functions concisely in this Thesis, therefore, a list of all the functions provided by the Petri net tool kit are listed in Appendix B.

5.5.1 Petri net functions

This section lists the functions provided by the Petri net class. These functions operate on the data structure described in Section 5.2. To illustrate the way in which functions can be used, in the following text they are applied to a Petri net object called PN.

5.5.1.1 Constructors

In C++, a constructor [Stroustrup 94] is a function that initialises and allocates new objects. In the Petri net class, the following types of constructor were designed:

1. Create a Petri net object from given pre-condition matrix, post-condition matrix, initial marking vector, number of places and number of transitions. This constructor is defined as:

petrinet(matrix, matrix, vector, int, int);

To illustrate the usage of this constructor, assume that we have defined the following:

```
matrix      pre_condition, post_condition;
vector      initial_marking;
int         number_of_places, number_of_transitions;
```

Then to instantiate a Petri net object, PN, using the information defined above, one would use the constructor:

```
petrinet    PN(pre_condition, post_condition, initial_marking,
            number_of_places, number_of_transitions);
```

2. Create a Petri net object from a net list file . This constructor is defined as:

*petrinet(const char * netlistfile);*

Then, to instantiate a Petri net object, PN, from a net list file called netlist.dat, one would use the constructor:

```
petrinet    PN("netlist.dat");
```

where "netlist.dat" is a text file, containing information regarding the size of the Petri net and the way in which the places and transitions are connected, with the following format:

```
"
# Netlist written in standard format
# Wed Oct 18 15:05:56 1995
#
places=40   transitions=27
:::
transition 1: input places { 1 12 }: output places { 2 13 };
transition 2: input places { 2 15 }: output places { 3 14 };
transition 3: input places { 3 14 37 }: output places { 4 15 };
transition 4: input places { 4 }: output places { 5 };
transition 5: input places { 5 13 }: output places { 6 12 };
... etc.
"
```

3. Create a Petri net object from a graphics file. This constructor is defined as:
   *petrinet(char * graphicsfile);*

Then, to instantiate a Petri net object, PN, from a graphics file called graphics.dat, one would use the constructor:

petrinet PN("graphics.dat");

where "graphics.dat" is a text file of a similar format to "netlist.dat" shown above, but also includes information regarding the location of each place and transition in terms of Cartesian co-ordinates.

5.5.1.2 File output

To save the attributes of a Petri net object in the form of a net list file, the following function was defined:

*void write_netlist(char *filename, char *msg = "");*

To save the attributes of the Petri net object, PN, on a file called "netlist.dat" with a message in the file header, the function would be used as shown below:

PN.write_netlist("netlist.dat","This is a demo Petri net");

5.5.1.3 Accessing the attributes of Petri net objects

The following virtual functions were implemented to access the data structure of the Petri net objects:

PN.Max_place_number() and *PN.Max_transition_number()* return the largest place and transition number respectively.

PN.Initial_marking() returns the initial marking vector of the Petri net, PN.

PN.Input_places(transition) and *PN.Output_places(transition)* return the input places and output places of a transition respectively.
PN.Input_transitions(place) and *PN.Output_transitions(place)* return the input transitions and output transitions of a place respectively.

5.5.1.4 Petri net evolution

Three functions that are related to the Petri net evolution have been implemented according to Definition 2.4 in Section 2.2.2. These are:

PN.find_enabled_transitions(marking_vector) returns a list of transitions that are enabled for a particular marking vector called "marking_vector" [Section 5.2.1].

PN.find_new_marking(marking, transition) returns the new marking of the Petri net, obtained by firing a particular transition at a particular marking [Section 5.2.2].

PN.find_marking(marking, transition_sequence) returns a marking obtained by firing the transition sequence starting from the marking vector that is passed to the function.

5.5.1.5 Petri net analysis

The following functions were implemented to enable the automation of coverability graph, invariant and reduction rules analysis of Petri nets.

PN.coverability graph() generates the coverability graph [Section 2.3.3] of the Petri net, PN. This is an implementation of the algorithm described in Section 5.2.3. Analysis functions to find liveness, deadlock states, unbounded places and concurrency sets are functions associated with reachability tree objects. They have also been implemented and are listed in Appendix B.

The virtual functions *PN.p_invariants()* and *PN.t_invariants()* calculate the P- and T-invariants [Section 2.3.4.1] respectively, using the algorithm described in [Section 5.2.4].

PN.reductionR1() returns the index of place that can be substituted [Section 2.3.5.1], *PN.reductionR2()* returns the index of neutral transition [Section 2.3.5.2] and *PN.reductionR3()* returns the index of an identical transition [Section 2.3.5.3].

Then, *PN.sub_place(place_number)* performs a place substitution on the place with index place_number and *PN.rem_trans(transition_number)*, removes the transition with index transition_number.

5.6 Development of a Petri net analysis tool

To be able to analyse large Petri net models generated during this research work, it was necessary to have access to a Petri net analysis tool. The requirements for this tool were that it was to provide graphical and text input, analysis of ordinary and timed Petri nets, reachability tree analysis and invariant analysis. Therefore, a Petri net analysis tool was developed by using the C++ Petri net tool kit developed in this Chapter, a graphic user interface (GUI) library[1] and a public domain Petri net graphical simulator[2]. The functionality of the Petri net analysis tool is described briefly in this section.

The main window consists of a grid where Petri net graphics are displayed and four drop-down menus (Fig. 5.3) that contain the main functions of the analysis tool.

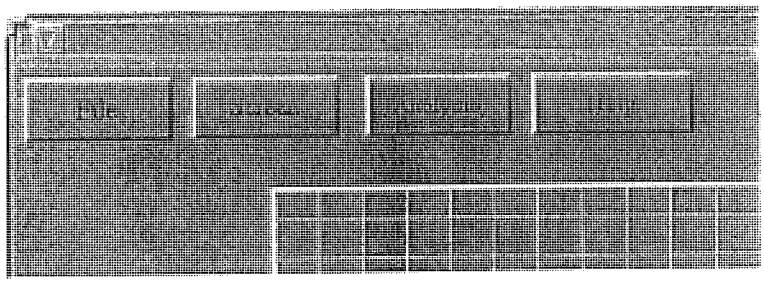


Fig. 5.3 The analysis tool menus

The "File" menu (Fig. 5.4) provides file handling facilities. The Petri net information can be loaded or saved in the form of a graphics file or in the form of a net list file (without graphics).

[1] SUIT, a GUI library developed by the Computer Science Department, University of Virginia, USA
[2] Petri, a public domain Petri net simulator written by Sunil Gupta, Brunel University, UK

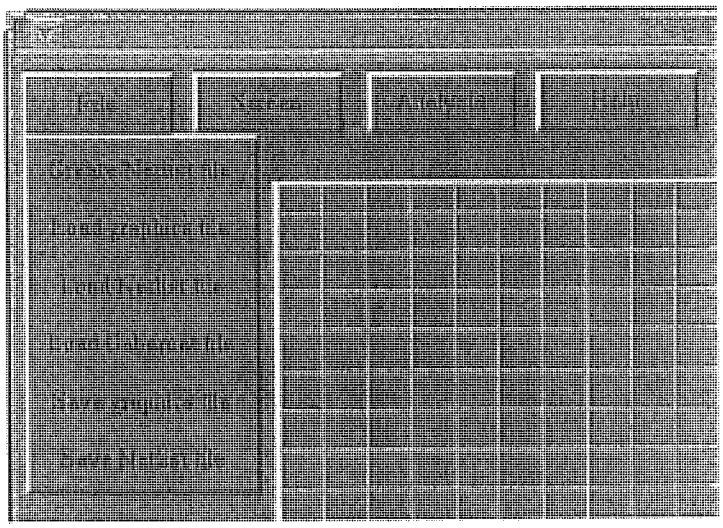


Fig. 5.4 The File drop-down menu

The Petri net can also be input by interactively using the mouse and graphical input buttons (Fig. 5.5) to instantiate places, transitions, tokens and arcs. The Petri net components are positioned on a graphics panel using the mouse. The Petri net components can be labelled (Fig. 5.6) to make the Petri net graphics more understandable.

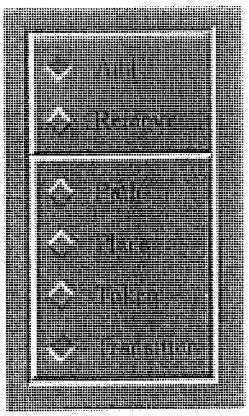


Fig. 5.5 Graphical input buttons

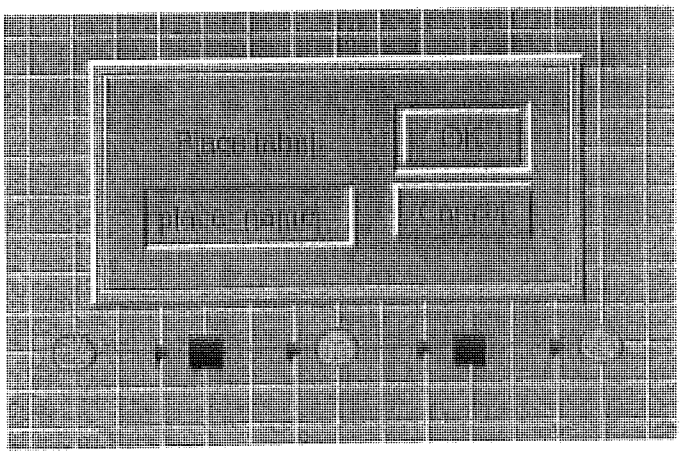


Fig. 5.6 Labelling places and transitions

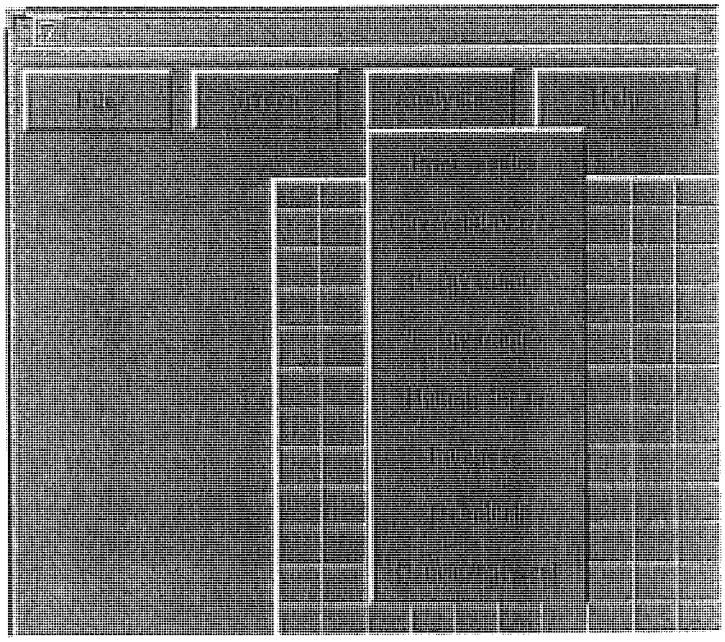


Fig. 5.7 Analysis drop-down menu

When the Petri net is loaded into the analysis tool, then, analysis may be performed by choosing the appropriate analysis function from the "Analysis" menu (Fig. 5.7).

The analysis tool described in this section has been successfully utilised in the research activities of the IT Research Group at Aston University and has been used to analyse Petri nets developed in this Thesis and in [Holding *et al.* 95, 96, Jiang *et al.* 96, Jiang and Holding 96]. These applications include Petri nets of over 100 components and reachable state spaces in excess of 1500 states.

5.7 Conclusion

This Chapter has presented a Petri net tool kit which provides an extension to object oriented C++ that allows operations on Petri nets, Petri net extensions and reachability tree objects. Within the design process of a control system for DEDSs, it can be utilised to analyse the behavioural and structural properties of the system [Section 2.3]; to implement the token-player algorithm [Section 4.3] and to implement algorithms to schedule exclusive-use resources [Section 4.5]

In the following Chapter, the object oriented methodology, scheduling algorithms and software tool kit that were developed in this Thesis have been applied to design and analyse two complex industrial plants.

Chapter 6

Analysis and control of industrial DEDSs

6.1 Introduction

This chapter illustrates how the object oriented methodology that has been developed in this Thesis can be used to model, analyse and optimally control industrial DEDSs. The methodology is applied to two industrial DEDSs. The first is a prototype can sorting machine, developed by Eurotherm Controls Ltd., described in [Jiang 95], and the second example is the semiconductor testing facility of SGS-Thomson Microelectronics (Malta) Ltd. described in [Azzopardi *et al.* 96].

6.2 A high-speed can sorting machine

The can sorting machine illustrated in Fig. 6.1 was documented in [Jiang 95, Holding *et al.* 95] where a System Behaviour Driven Method (SBDM) was used to obtain an Extended Timed Place Transition Net (XTPTN) model of the machine and its behavioural and safety properties were verified using temporal logic. In this section, the modified OMT methodology [Chapter 3] and Petri net tool kit [Chapter 5] are applied to the high-speed can sorting machine to design the synchronisation logic for the independently driven axis of the machine, and illustrate the modularity and re-usability aspects of the methodology. Fig. 6.2 and Fig. 6.3 are a side elevation and a front view of the can sorting machine.

6.2.1 Problem statement

The prototype can sorting machine is illustrated in Figs. 6.1, 6.2 and 6.3. Cans are transferred from the feeder to `Drum1` by `Piston1`, from `Drum1` to `Drum2` by `Piston2` and from `Drum2` to the conveyor by `Piston3`. The two drums, three pistons, feeder and conveyor are driven by independently controlled motors and need to be synchronised so that cans can be transferred safely between the different components of the machine. The drums rotate by a fixed angle of rotation upon reception of a `start rotation` signal.

A piston inserts and withdraws as a continuous motion on reception of a `start cycle` signal.

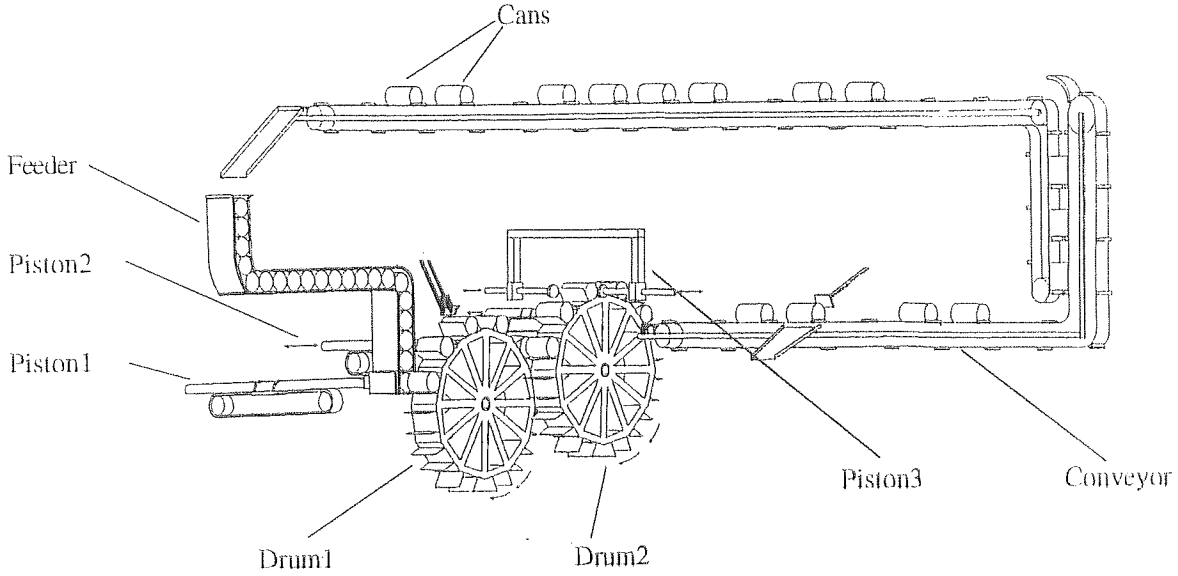


Fig. 6.1 Eurotherm controls prototype can sorting machine

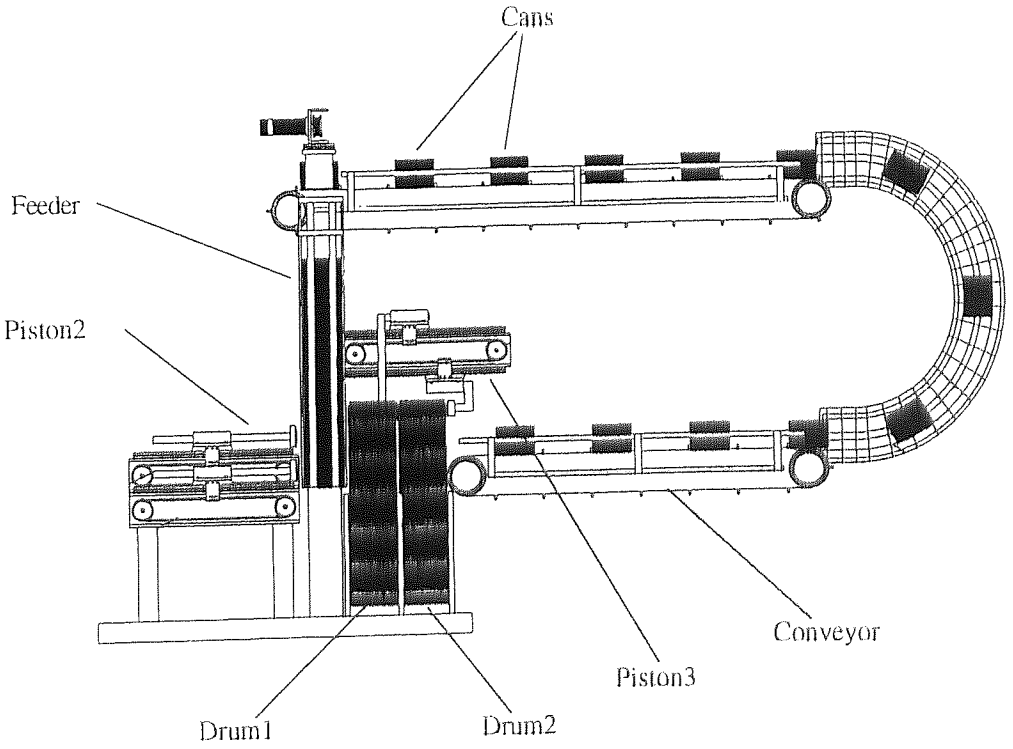


Fig. 6.2 Side elevation

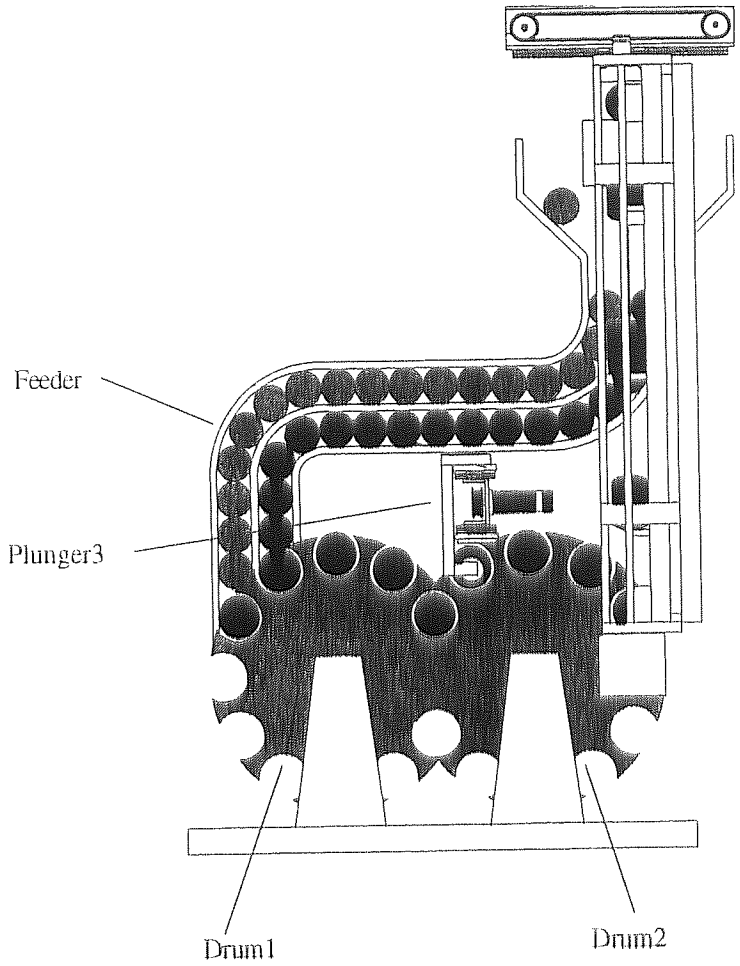


Fig. 6.3 Front view

A controller is required to provide the synchronisation logic to satisfy the following operational requirements [Jiang 95] in order to ensure the safe and correct operation of the machine :

1. The drum should not rotate whilst a piston is inserting into it
2. The piston should not insert while the drum is rotating
3. For a can to be transferred between the two drums, both must be stationary
4. The independent motion of the drums is permitted.

6.2.2 Object model

In this problem we can identify four classes of objects: drum, piston, conveyor and feeder. For simplicity, we will assume that the drums can be stationary in one of three positions or rotating from one position to the next. The drum is controlled by a `start rotation` signal, whilst the event that stops the drum rotating is an automatic event. The drum is initially `stationary` and at `Position_1`.

A piston inserts and withdraws as a continuous motion on reception of a `start cycle` signal, therefore a piston may be either `stationary` or performing an `insert cycle`. The piston is initially `stationary`. The conveyor can be `running` or `stationary` and is initially `running`. It can be started or stopped manually. The can feeder is always available for the piston to insert. Thus, the attributes and functions of the drum, piston, feeder and conveyor are shown in Fig. 6.4.

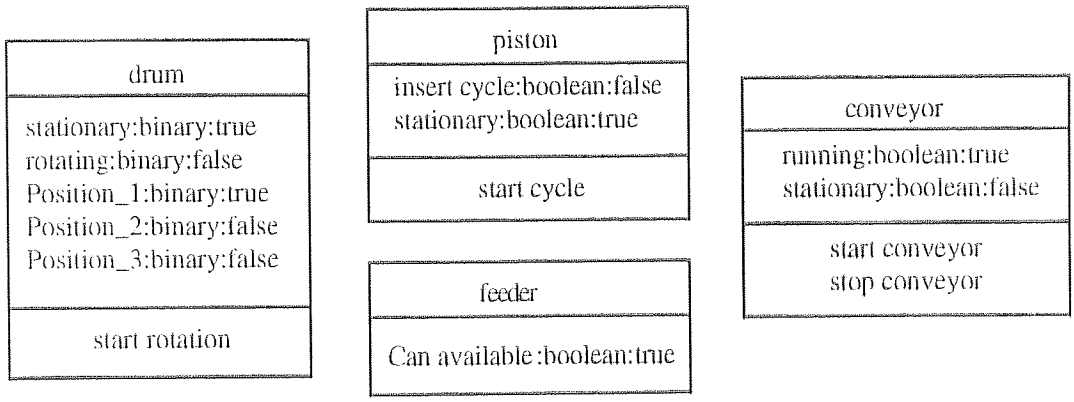


Fig. 6.4 Definition of classes

Two pistons are associated with each drum and there may be either one or two drums associated with a piston. There is one piston associated with a conveyor and a piston may or may not be associated with a conveyor. There is one piston associated with a feeder but a piston may or may not be associated with a feeder. Therefore, the class associations are illustrated in the object model diagram of Fig. 6.5 using standard OMT notation [Rumbaugh *et al.* 91].

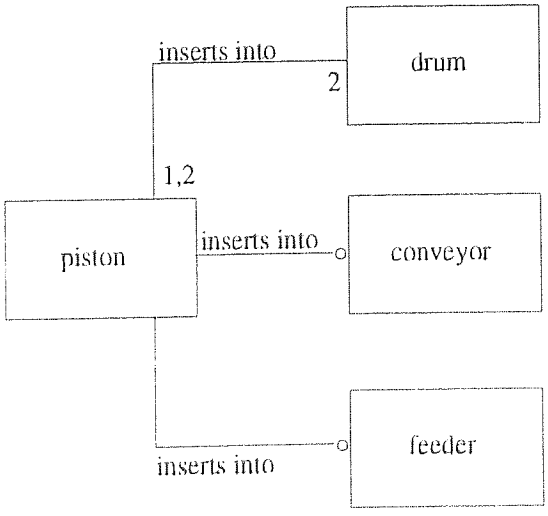


Fig. 6.5 Can sorting machine object diagram

The object model of the can sorting machine consists of the class definitions of Fig. 6.4 and the object model diagram of Fig. 6.5.

6.2.3 Dynamic model

The first step in designing the dynamic model of the machine is to identify the communication events between the objects. In this problem, the drum, conveyor and feeder objects communicate with piston objects when a piston inserts into them. On the other hand the piston communicates with the other objects by inserting into them. Therefore, the OCTs are as illustrated in Fig. 6.6.

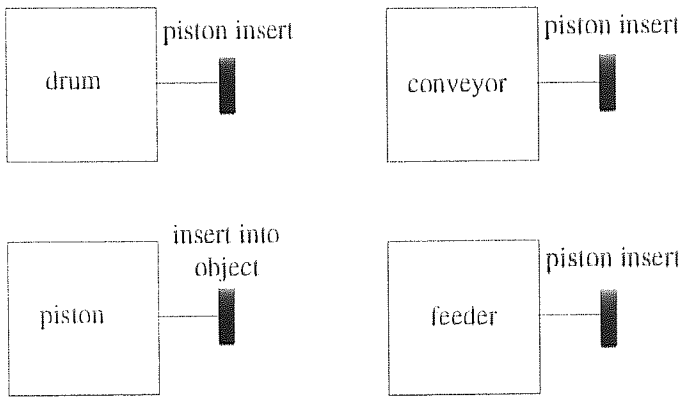


Fig. 6.6 The object communication transitions (OCTs)

The next stage involves developing a Petri net model for each class to describe its important dynamic behaviour. The objects' binary attributes, defined in the object model, are represented by Petri net places in the dynamic model and events are represented by transitions. The controllable events are represented by controlled transitions [Krogh 87]. The places and transitions are then linked by directed arcs to represent the dynamic behaviour of the objects as described in the problem statement. The Petri net models of the piston, conveyor, drum and feeder class are illustrated in Fig. 6.7.

To obtain the complete Petri net model that represents the dynamic behaviour of the machine, it is necessary to initialise the required objects and "hook" them together my merging the relevant OCTs. As described in the problem statement, the can sorting machine consists of two drums, three pistons, a feeder and a conveyor. Therefore two drum objects: `Drum1` and `Drum2`; three piston objects: `Piston1`, `Piston2`, `Piston3`; a feeder object; and conveyor object are initialised and a Petri net is formed by merging the relevant OCTs as shown in Fig. 6.8.

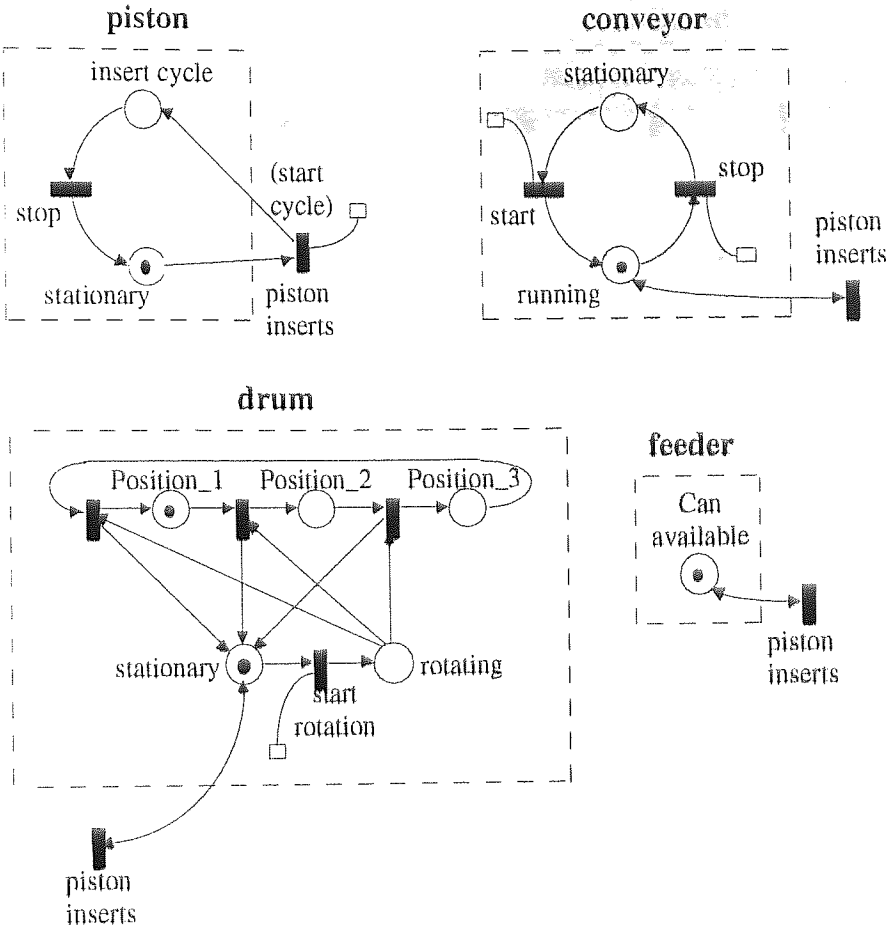


Fig. 6.7 The class dynamic models

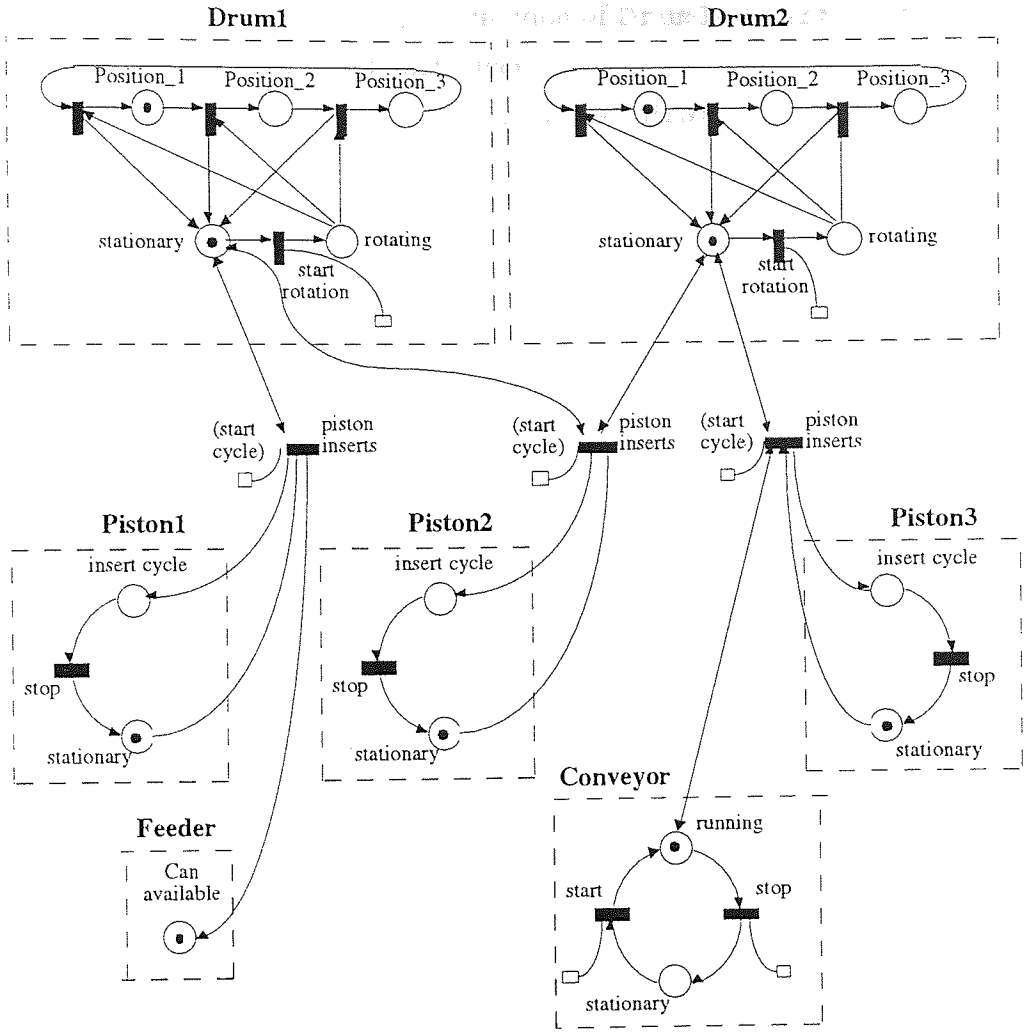


Fig. 6.8 The Petri net model representing the can sorting machine

6.2.4 Functional model

The functional model defines the constraints of the system operation. The output of the functions change the state of the control places in the Petri net model. Therefore in this section, functions are specified to restrict the dynamic behaviour of the system to that described in the problem statement. (Note: The "." notation normally used in OO terminology is adopted. i.e. `Piston2.stationary = true` means that the attribute, stationary, that belongs to `Piston2` is true, i.e. "Piston 2 is stationary". `Drum2.control = 1` means that the control place of`Drum2` is equal to one, etc.).

Function 1: `Drum1.control`
`Drum1.stationary ∧ Piston1.stationary ∧ Piston2.stationary →`
`Drum1.start_rotation.`

Since Drum1.stationary is a pre-condition of Drum1.start_rotation that is already defined by the Petri net structure, then;

Drum1.control = Piston1.stationary ∧ Piston2.stationary ... (6.1)

Function 2: Drum2.control

Drum2.stationary ∧ Piston2.stationary ∧ Piston3.stationary → Drum2.start_rotation.

Since Drum2.stationary is a pre-condition of Drum2.start_rotation that is already defined by the Petri net structure, then;

Drum2.control = Piston2.stationary ∧ Piston3.stationary ... (6.2)

Function 3: Piston1.control

Feeder.can_available ∧ Piston1.stationary ∧ Drum1.stationary → Piston1.start_cycle.

Since Feeder.can_available, Piston1.stationary and Drum1.stationary are pre-conditions of Piston1.start_cycle that are already defined by the Petri net structure, then Piston1.control = 1 ... (6.3)

Function 4: Piston2.control

Drum2.stationary ∧ Piston2.stationary ∧ Drum1.stationary → Piston2.start_cycle.

Since Drum2.stationary, Piston2.stationary and Drum1.stationary are pre-conditions of Piston2.start_cycle that are already defined by the Petri net structure, then;

Piston2.control = 1 ... (6.4)

Function 5: Piston3.control

Conveyor.running ∧ Piston3.stationary ∧ Drum2.stationary → Piston3.start_cycle.

Since Conveyor.running, Piston3.stationary and Drum2.stationary are pre-conditions of Piston3.start_cycle that are already defined by the Petri net structure, then;

Piston3.control = 1 ... (6.5)

Function 5: conveyor.control

Conveyor.start and Conveyor.stop are controlled manually by an operator. Therefore, since the initial condition of the conveyor is that it is running,

conveyor.control_1 = conveyor.control_2 = 0. ... (6.6)

6.2.5 Behavioural Analysis

The Petri net, with the appropriate functions defining the states of the control places, was analysed using the Petri net analysis tool [Section 5.6]. In this particular case, since the functions are based on states that are modelled as Petri net places, the controlled Petri net was converted into an ordinary Petri net by adding arcs to implement the functions as additional pre-conditions for the controllable events. The Petri net model was verified to be live, deadlock free and 1-bounded. Table 6.1 shows the complete concurrency sets of the Petri net illustrated in Fig. 6.8. Concurrency set analysis [Jiang *et al.* 96] was used in order to ensure that the functional and safety requirements, specified in the problem statement, were satisfied.

State	Place	Concurrency Set
Feeder.can_available	1	{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18}
Conveyor.running	2	{1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18}
Piston1.stationary	3	{1, 2, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18}
Piston2.stationary	4	{1, 2, 3, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18}
Piston3.stationary	5	{1, 2, 3, 4, 6, 7, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18}
Piston1.inserting	6	{1, 2, 4, 5, 7, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18}
Piston2.inserting	7	{1, 2, 3, 5, 6, 8, 9, 11, 13, 14, 15, 16, 17, 18}
Piston3.inserting	8	{1, 2, 3, 4, 6, 7, 9, 10, 11, 13, 14, 15, 16, 17, 18}
Drum_1.stationary	9	{1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 13, 14, 15, 16, 17, 18}
Drum_1.rotating	10	{1, 2, 3, 4, 5, 8, 11, 12, 13, 14, 15, 16, 17, 18}
Drum_2.stationary	11	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 13, 14, 15, 16, 17, 18}
Drum_2.rotating	12	{1, 2, 3, 4, 5, 6, 9, 10, 13, 14, 15, 16, 17, 18}
Drum_1.position1	13	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 16, 17, 18}
Drum_1.position2	14	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 16, 17, 18}
Drum_1.position3	15	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 16, 17, 18}
Drum_2.position1	16	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
Drum_2.position2	17	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}
Drum_2.position3	18	{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}

Table 6.1 Concurrency sets

Property 1. The drum should not rotate whilst a piston is inserting into it.
This property can be verified by analysis of the concurrency sets from which it must be shown that `Drum1.rotating` is not present in the concurrency set of `Piston1.inserting` and `Piston2.inserting`, and `Drum2.rotating` is not present in the concurrency sets of `Piston2.inserting` and `Piston3.inserting`. This can be verified by inspection of Table 6.1.

Property 2. The piston should not insert while the drum is rotating.

To verify this property, `Piston1.inserting` and `Piston2.inserting` must not be present in the concurrency set of `Drum1.rotating` and, `Piston2.inserting`, and `Piston3.inserting` must not be present in the concurrency set of `Drum2.rotating`. This can be verified by inspection of Table 6.1.

Property 3. For a can to be transferred between the two drums, both drums must be stationary.

Therefore the places representing `Drum_1.rotating` and `Drum2.rotating` must not be present in the concurrency set for `Piston2.inserting`. From Table 6.1, can be seen that places 10 and 12 do not belong to the concurrency set of place 7, therefore this requirement is satisfied.

Property 4. The independent motion of the drums is permitted.

To maximise performance, the independent motion of the drums is permitted and required. Therefore the places representing `Drum_1.rotating` and `Drum2.rotating` must be present in the same concurrency set. It can be seen from Table 6.1 that place 10 is present in the concurrency set of place 12, hence this requirement is satisfied.

6.2.6 The controller

The can sorting machine can be controlled by a PLC or microprocessor based controller as shown in Fig. 6.9. The controller executes a token player algorithm that is driven by the state feedback signals from the machine. The controller responds by sending the control signals to trigger the controllable events. The set of control signals corresponds to the set of control places in the OMT dynamic model. Therefore, in this application, the set of control signals are: `Drum1.contol`, `Drum2.control`, `Piston1.control`, `Piston2.control`, and `Piston3.control`.

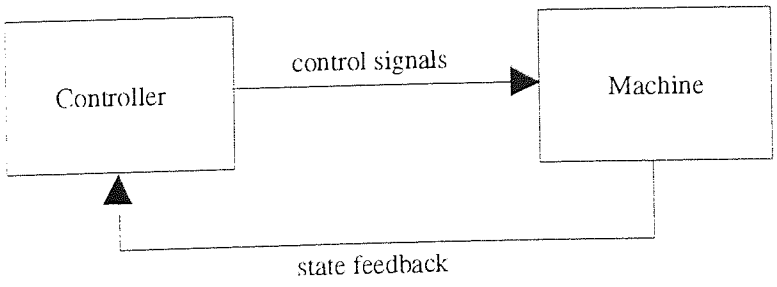


Fig. 6.9 Control of the can-sorting machine

6.2.7 Comparison with a previous design

The can sorting machine described above was first documented in [Jiang 95, Holding *et al.* 95] where a System Behaviour Driven Method (SBDM) was used to obtain the rule-based functional requirements of the system and the behaviour of the machine was modelled on an XTPTN model in [Jiang 95]. The methodology (SBDM) developed by Jiang [95] and the resulting model are discussed briefly and compared with the OO methodology and resulting CPN developed in this Thesis.

SDBM is based on Structured Common Sense [Goldsack and Finkelstein 91] that was developed to support the construction of a formal requirements specification for real time systems. The steps defined in SBDM are:

- Identify each autonomous component within the system
- Determine the actions that each component performs
- Determine the states that each component can be in.
- Identify the events that trigger state transitions
- Determine the links and interactions between the states and construct a rule based representation of the behaviour of the system

Jiang [95] also developed a method for translating the rule based representation into an XTPTN to enable behavioural and timing analysis. This was extended by Holding et al. [95] to translate the rule based representation into a CPN. The application of SBDM to the can sorting machine described above, resulted in the CPN illustrated in Fig. 6.10 where the semantics of the places and transitions are as defined in Table 6.2.

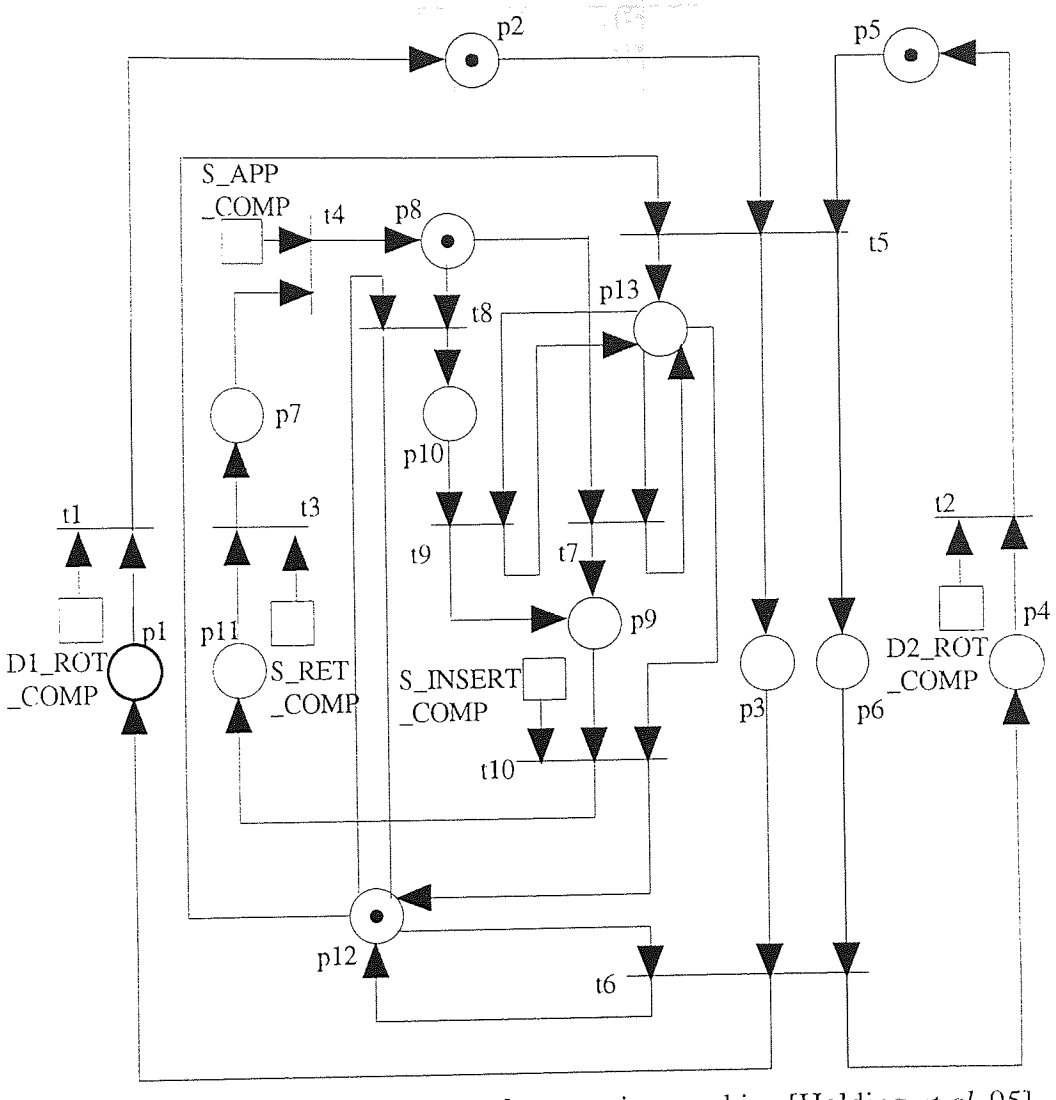


Fig. 6.10 CPN representation of can-sorting machine [Holding *et al.* 95]

Place	Entity	Action / interpretation
1	Drum1	Rotate
2	Drum1	Wait_to_transfer
3	Drum1	Committed_to_transfer
4	Drum2	Rotate
5	Drum2	Wait_to_transfer
6	Drum2	Committed_to_transfer
7	Transfer Slider	Approach_motion
8	Transfer Slider	Decision_position
9	Transfer Slider	Insert_motion
10	Transfer Slider	Abort_motion
11	Transfer Slider	Return_motion
12	Synchronisation logic	Slider_insert_inhibit
13	Synchronisation logic	Drum_rotate_inhibit

Table 6.2 Semantics of SFC model of multi-axis system

Transition	Associated Condition	Mnemonic	Comment
t1	Drum1_rotate_complete	D1_ROT_COMP	
t2	Drum2_rotate_complete	D2_ROT_COMP	
t3	Slider_return_complete	S_RET_COMP	
t4	Slider_approach_ complete	S_APP_COMP	
t5	c1		c1 and c2 ensure mutual exclusion between transitions on selection construct at Step 12
t6	c2		
t8	$\neg c1 \wedge \neg c2$		
t10	S_insert_complete	S_INSERT_ COMP	

Table 6.3 Conditions of SFC model of multi-axis system

Jiang [95] proved that the Petri net in Fig. 6.10 satisfies the safety and liveness criteria that were specified in the problem statement however, the CPNs of Fig. 6.8 and 6.10 are different in various respects. The CPN that was designed using the modified OMT (Fig. 6.8) is clearly modular and shows the separate objects and their synchronisation by means of the OCTs, whereas this is not the case in the CPN that was obtained using SBDM (Fig. 6.10). This makes the CPN of Fig. 6.8 more clear and easier to understand than the CPN of Fig. 6.10. Furthermore, the modularity of the CPN of Fig. 6.8 makes it easier to re-use modules and to make modifications whereas any modifications to the CPN of Fig. 6.10 would have large ripple effects. Another major difference between the two CPNs is the interpretation of the control places. In the CPN that was designed in this Thesis, the control places have the same interpretation as defined by Holloway and Krogh [90], and therefore make use of functions (defined in the functional model) to define their states. However, Holding *et al.* [95] used the control places to represent feedback from the plant. This makes it very difficult to analyse the CPN of Fig. 6.10 unless additional work is done to define functions which simulate feedback from the machine.

Although there are similarities between OMT and SBDM in that the first stages involve identifying the autonomous components (objects) in the system, there are major advantages that the modified OMT presented in this Thesis has over SBDM. First of all, SBDM, unlike OMT, does not distinguish between static (object), dynamic and functional models. Secondly, OMT is more suitable for the analysis of larger systems because it allows re-usability by organising objects into classes and by using inheritance whereas SBDM requires the identification of actions, states and events for each object. Also, OMT designs are modular and therefore they are easier to understand and modify and object oriented designs can be easily implemented in OOP languages such as C++. Finally OMT uses graphical representation for the object model, dynamic model and functional model, making designs easier to understand.

6.3 A semiconductor testing plant

A semiconductor testing plant is a complex batch process plant with large number of processing units and products. A large capital is involved in operating these plants, so it is important to model and analyse these systems to ensure optimal operation. This section illustrates the application of the methodology developed in this Thesis, to the semiconductor testing plant of SGS Thomson Microelectronics Ltd. in Malta which has resulted in the publication of a conference paper [Azzopardi *et al.* 96].

6.3.1 Description of semiconductor testing plants

A semiconductor testing plant[1] consists of a number of processing units such as testers, ovens, printers, lead-straighteners, scanners and packing machines (Fig. 6.11). Semiconductor devices of a specific type are tested in approximately constant size batches.

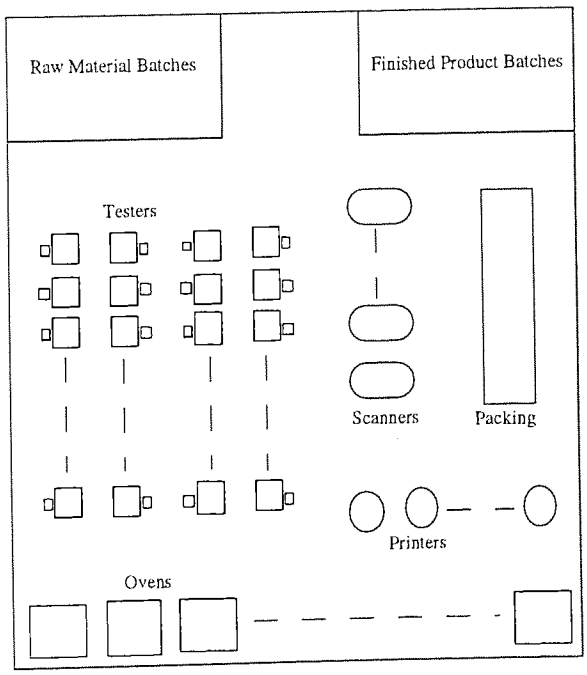


Fig. 6.11 Layout of a Semiconductor testing plant

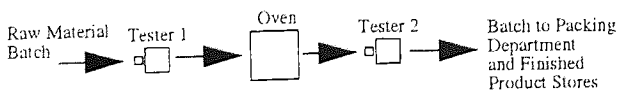


Fig. 6.12 A typical product flow

[1] Based on the testing facility of SGS-Thomson Microelectronics plant in Malta.

Typically a batch must go through a number of tests using various testers, a burn-in or bake period, printing, lead straightening, scanning and packing (Fig. 6.12). Batches of different types of devices must undergo different production paths and the duration of each process differs depending on the type of device. Each tester is capable of testing different devices by using different device handlers and test software.

The number of batches of the required types of device are calculated typically on a monthly basis depending on customer orders. The main problem for production control personnel is to allocate batches of different products to the testers and other machinery in order to avoid bottlenecks, maximise throughput and machine utilisation.

Semiconductor plants are fast moving batch process plants in which time for analysis and scheduling is very limited. Since optimal scheduling of a large number of machines to maximise throughput is a very difficult and time consuming task, rule-of-thumb scheduling methods are often used to arrive at workable but sub-optimal schedules [MacCarthy and Liu 93]. The SGS-Thomson semiconductor testing facility in Malta is a batch process plant which currently tests 700 different products using 100 testers of 21 different types, 26 bake ovens and 29 burn-in ovens. If one were to mathematically model such a plant, it would be possible to apply scheduling algorithms to the model, ensuring optimal control and maximising throughput of the plant. Furthermore, by utilising the controller designed in Chapter 4, the plant would be able to be operated optimally, even in the event of a production unit breaking down.

In the following sections we show how OMT, as modified in this Thesis, and Petri net theory were applied to model a production run of the micros production line of the semiconductor testing plant at SGS-Thomson Microelectronics, Malta.

6.3.2 Problem statement

The micros production line consisted of 10 Type-A testers, 3 Type-B testers and 4 Type-C testers. The production requirement consists of 51 products. The products are tested using one of the two test flows: Test or Test-Bake-Test. The test flow and test time for each batch of material of the products that have a Test-Bake-Test flow are specified in Table 6.4. The test times are measured in hours. It is required to model the plant and to schedule it for its optimal operation.

Prod. No.	Tester No.	1st test time	bake time	Tester No.	2nd test time
1	1	4.5	26.5	8	8.5
2	1	28.5	38.5	8	68.6
3	1	129	38.5	7	379.8
4	1	31.5	38.5	9	77
5	2	117.5	38.5	8	282.7
6	3	117.5	38.5	9	282.7
7	3	117.5	38.5	10	282.7
8	4	105	38.5	11	268.3
9	2	45	38.5	12	108.3
10	4	111	38.5	12	234.33
11	12	19	24	10	19
12	2	4.5	38.5	13	8.8
13	11	38.7	26.5	11	67.6
14	5	40.3	26.5	10	68.1
15	13	60.6	24	13	60.6
16	14	43.5	26.5	14	43.5
17	6	11.7	26.5	14	28.3
18	6	4.8	26.5	14	12.4
19	6	8.1	26.5	14	15
20	6	8.1	26.5	14	20.6
21	5	124	26.5	13	241.3
22	6	124	26.5	14	241.3
23	6	8.1	26.5	8	14.3
24	6	11.7	26.5	9	44.3
25	6	13.9	26.5	12	21.3
26	11	9	24	13	9
Totals		1337	801.5		2908.33

Table 6.4 Test flows for micros products with two tests

6.3.3 Object model

In this problem, there are 3 classes of objects: products, testers, and ovens. There are two classes of products; Product-T, whose test flow only consists of one test and Product-TBT, whose test flow is Test-Bake-Test.

Since both types of product have attributes in common, such as a technical code, package type (i.e. DIL, PQFP, PLCC, etc.), the Product-TBT class and the Product-T class are derived from the same product class, with additional information pertaining to the specific class. For example, the Product-T class has attributes indicating; the test time required to test a batch of devices, the type of tester required, whether the product is being tested or whether the raw material is available. The Tester class has attributes that indicate the type of tester and whether it is available or not and the Oven class has

attributes indicating whether the oven is available or not. The class definitions are illustrated in Fig. 6.13, below.

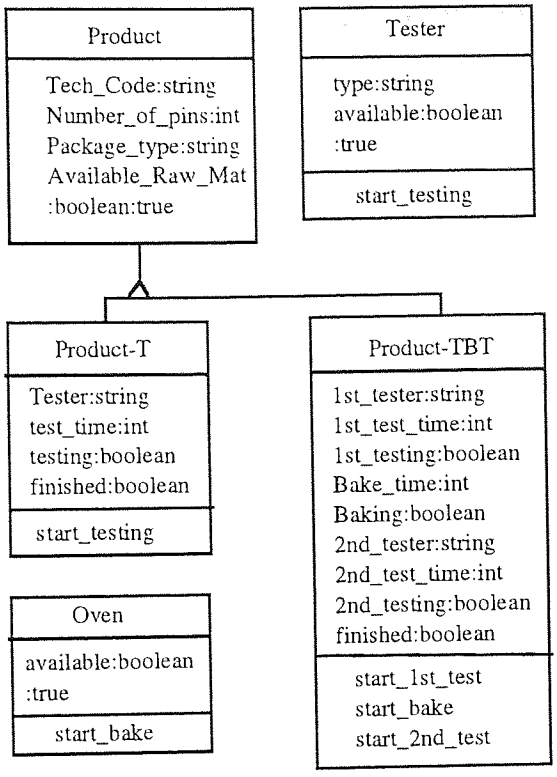


Fig. 6.13 Class Definitions

The next stage in the design of the object model involves adding the associations between the classes. For example, each object belonging to the derived class Product-TBT is associated with 2 testers and an oven, whilst each tester may test many products. hence, the associations between classes are as illustrated in Fig. 6.14.

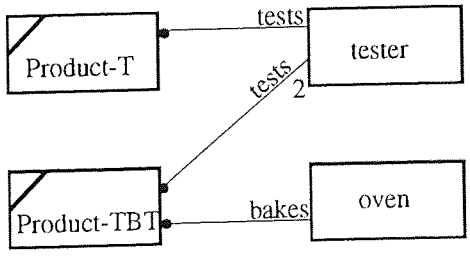


Fig. 6.14 Object Model

6.3.4 Dynamic model

The dynamic model of each class of objects was constructed by taking the following steps:

The first step in designing the dynamic model of the machine is to identify the communication events between the objects and represent them by Petri net transitions. In this problem, Tester objects communicate with product objects by starting and ending the testing process and conversely, the products communicate with the testers by starting and stopping the testing process. Thus the OCTs are as shown in Fig. 6.15.

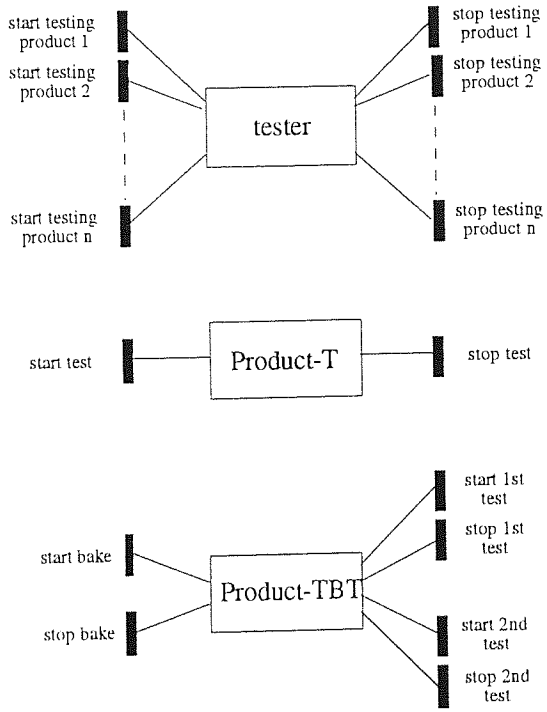


Fig. 6.15 Object Communication Transitions

The next stage involves developing a Petri net model for each class to describe its important dynamic behaviour. The objects' binary attributes in the object model are represented by Petri net places in the dynamic model and events are represented by transitions. The controllable events are represented by controlled transitions [Krogh 87]. The places and transitions are then linked by directed arcs to represent the dynamic behaviour of the objects as described in the problem statement. Therefore the dynamic model (excluding control places for clarity) is as shown in Fig. 6.16. The dynamic model of the ovens is identical to that of the testers and is not shown due to space considerations.

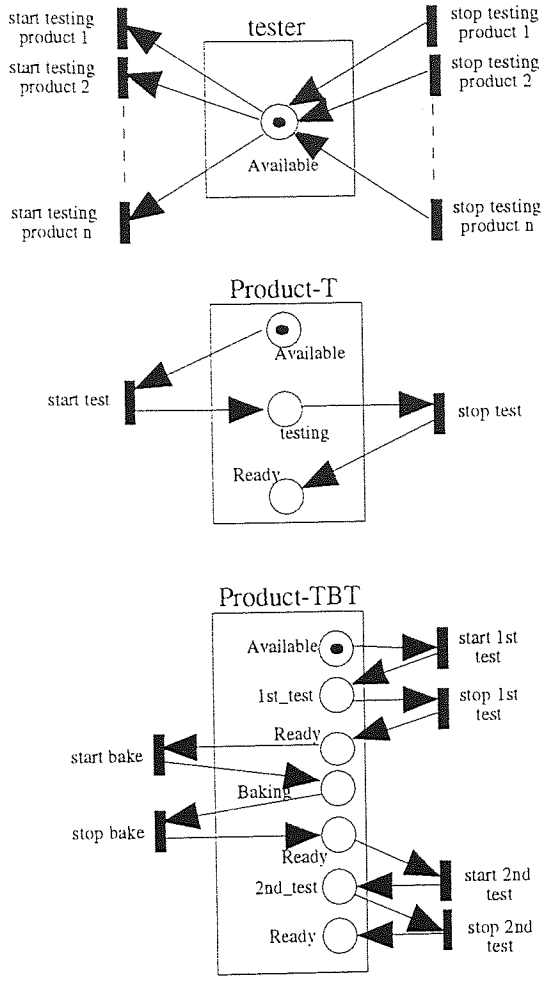


Fig. 6.16 The dynamic model

The complete dynamic model is obtained by initialising the appropriate number of objects, representing the products, testers and ovens. These are then hooked together by merging the OCTs. The production run modelled in this section consisted of 26 products of the TBT type, 26 products of the T type and 10 testers which resulted in a Petri net with 270 places and 208 transitions (which is too large to illustrate graphically in this Thesis). The size of this Petri net makes it very time consuming to analyse, even when using software tools, however, since the Petri net was built using well formed blocks [Valette 79] and PME places [Zhou & DiCesare 93] it is guaranteed to be live and 1-bounded.

6.3.5 Functional model

The state of the control places (that represent the enabling conditions of controllable events) must be represented as functions in the functional model. The functional model is an important part of the OMT model because it defines the control strategy of the production run. For this problem, the control strategy was to maximise throughput and therefore the functional model takes the form of the scheduling algorithm described in Section 4.5.3.

6.3.6 Controlling the plant

The algorithm described in Chapter 4 has been used to generate a sequence of transitions, S_{opt}, that minimises the cycle time. The Gantt chart showing the optimal schedule that is obtained by firing S_{opt} is illustrated in Figure 6.17.

The controller described in Chapter 4 of this Thesis can be used to control the plant by operating the token-player algorithm on the dynamic model of the plant. Using the efficient scheduling algorithm (described in Section 4.6) ensures optimal operation. Since the controller is re-configurable and has a re-scheduling facility, it can give production personnel an alternative schedule in the situation where a production unit breaks down. The alternative schedule is optimal for the degraded plant.

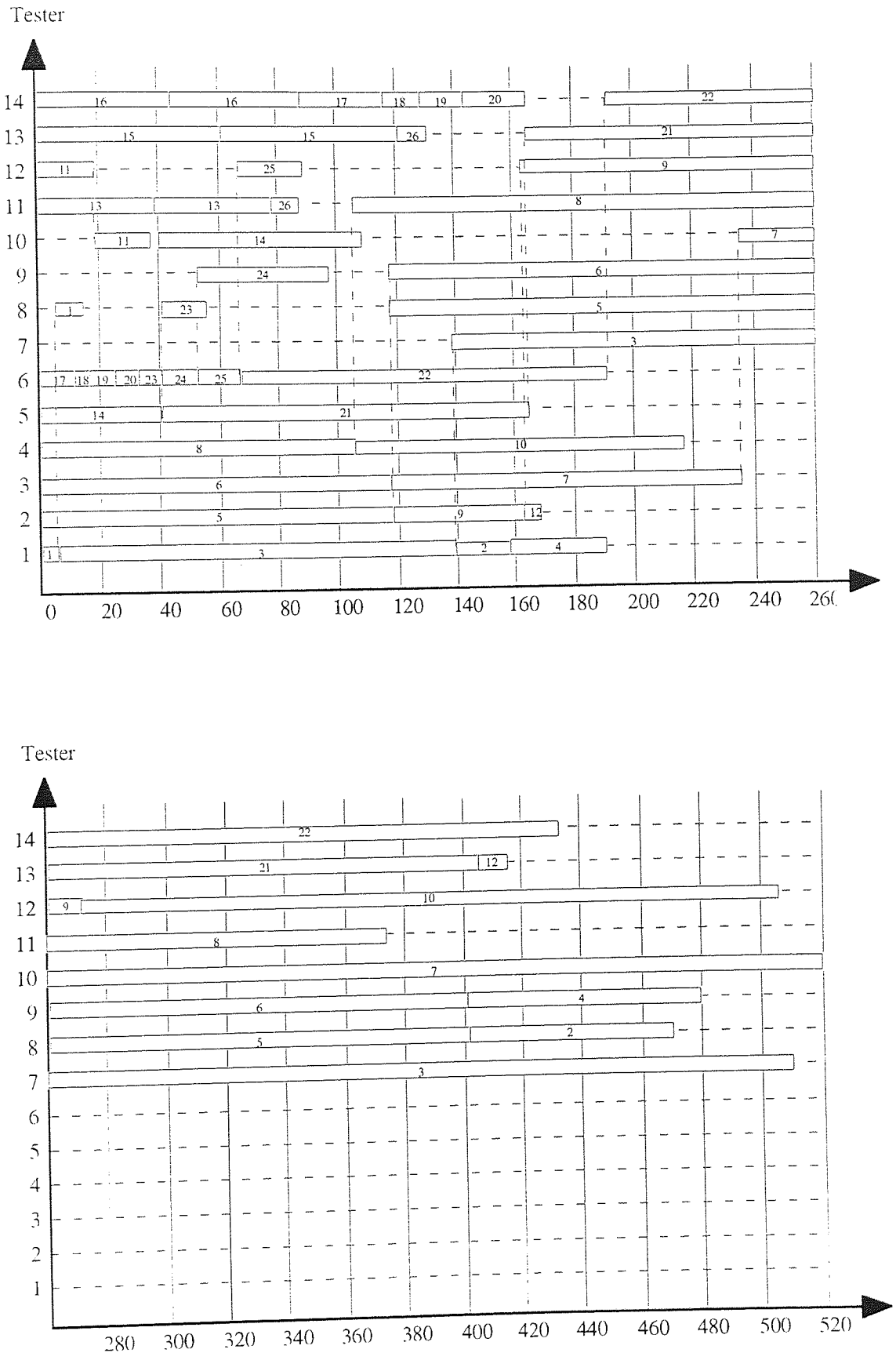


Fig. 6. 17 Gantt chart showing optimal schedule

6.4 Conclusion

This chapter has applied the OO methodology developed in Chapter 3, scheduling algorithm developed in Chapter 4 and the software tool kit developed in Chapter 5 to analyse and control two industrial DEDSs. The first example was based on a can sorting machine developed by Eurotherm Controls Ltd., described in [Jiang 95] and the second example was based on a semiconductor testing facility belonging to SGS-Thomson Microelectronics (Malta) Ltd. described in [Azzopardi *et al.* 96].

Both applications illustrate how the object oriented methodology that was developed in this Thesis allows re-use by grouping similar objects into classes (and by using inheritance in the 2nd example) and produces modular and understandable designs. By applying the methodology to two very different industrial DEDS it has been shown that the methodology is indeed practical and applicable to a wide range of DEDS.

Chapter 7

Conclusion

7.1 Introduction

This Thesis discussed techniques for modelling DEDSs and selected Petri net theory as the modelling technique that was most suited to the research. It described Petri net theory for analysis of DEDSs and discussed three main techniques including: coverability graph analysis, incidence matrix analysis and analysis by means of reduction rules. Since industrial DEDSs are very complex and difficult to model, this Thesis has reviewed currently available Petri net synthesis techniques, including top down, bottom up and hybrid synthesis for modelling large systems. It was noted that these techniques are based on functional abstraction and that the experience of researchers has shown that functional abstraction is not practical for large industrial applications. This Thesis therefore proposed using an object oriented methodology to facilitate the design of complex systems, produce more understandable designs and specifications, facilitate the transition between design and implementation and enable re-use of designs. The methodology was based on the well known OMT methodology and was applied to two well documented problems.

Once a model of the system is obtained, the next step in the design process is to design the controller for the system. This Thesis therefore investigated the adaptation of traditional control theory to DEDSs, that was recently pioneered by Ramadge and Wonham [87], and the adaptation of Petri net theory for supervisory control. It was then shown how exclusive use resources can be scheduled by using Petri net based scheduling algorithms in order to ensure optimal control. A novel scheduling algorithm based on Petri net reduction techniques was introduced with experimental evidence to show that it has a rate of convergence that is several times faster than previously published algorithms. Finally, a re-configurable controller was proposed to cater for unexpected changes in the plant.

This Thesis has also presented a Petri net tool kit which provides an extension to object oriented C++ that allows operations on Petri nets, Petri net extensions and reachability trees. Within the design process of a control system for DEDSs, it can be utilised to analyse the behavioural and structural properties of the system; to implement the token-player algorithm and to implement algorithms to schedule exclusive-use resources.

Finally, the OO methodology developed in Chapter 3, the scheduling algorithm developed in Chapter 4 and the software tool kit developed in Chapter 5 were used to analyse and control two industrial DEDSs. The first example was based on a can sorting machine developed by Eurotherm Controls Ltd., described in [Jiang 95] and the second example was based on a semiconductor testing facility belonging to SGS-Thomson Microelectronics (Malta) Ltd. described in [Azzopardi *et al.* 96]. Both applications illustrated how the object oriented methodology that was developed in this Thesis allows re-use by grouping similar objects into classes and produces modular and understandable designs. By applying the methodology to two very different industrial DEDS it has been shown that the methodology is indeed practical and applicable to a wide range of DEDS.

7.2 Summary of contributions

- **An object oriented methodology for synthesis of DEDS models**

This Thesis presented an object oriented methodology for synthesis of DEDS models. The technique is a modification of OMT [Rumbaugh et. al 91] and consists of three models: the object model, describing the objects in the system and their relationship; the dynamic model, describing the interactions among objects in the system; and the functional model, describing the data transformations of the system. It was shown that the dynamic model can be represented by a controlled Petri net to improve the representation and analysis of the dynamic model of DEDSs. There is a direct link between the three OMT models, thus the construction of the complete model can be obtained by taking a step by step approach. This methodology was applied to two classical problems in Chapter 3 and to two industrial DEDSs in Chapter 6.

- **A fast Petri net based scheduling algorithm**

This contribution is an improvement over previously published Petri net based scheduling algorithms. The algorithm uses a branch and bound algorithm applied to the timed Petri net model of the plant. The efficiency of this algorithm is improved by using heuristics to reduce the search space. The major improvement introduced in this Thesis involved reducing the Petri net, effectively removing all the uncontrollable events from the model to obtain a further reduction in the search space of the scheduling algorithm. The improvement in the rate of convergence of the algorithm is backed up by experimental results presented in Chapter 4 of this Thesis.

- **A re configurable supervisory controller for DEDSs**

Whenever shared resources in industrial DEDSs break down, jobs must be re-routed to make use of the available shared resources. In the event of a breakdown, the state-transition structure on which the supervisory controller is based would not be a correct representation of the plant, unless all possible failures are included in the model. Since it is not practical to model all possible failures in the supervisory controller, in Chapter 4 a Petri net model-based controller is proposed. It uses state feedback to detect changes in the set-up and incorrect response of the system and is re-configurable to accomodate for these changes.

- **A Petri net software tool kit**

To analyse the dynamic model of large DEDSs, to implement scheduling algorithms and control algorithms for realistically sized industrial DEDSs, an object oriented C++ Petri net software library was implemented. It was used for the analysis and implementation of all the examples discussed in this Thesis and for analysis of Petri net models resulting from the work of the IT research group at Aston University

7.3 Suggestions for further work

- **Automation of the modified OMT methodology**

There exist several software tools that allow automation of the OMT methodology. These, however use state diagrams to represent the dynamic model and most packages do not provide analysis facilities for the dynamic model. It would be very useful if further work was done to incorporate the Petri net tool kit that was developed in this Thesis to an OMT software tool. This would provide a software workbench for the design engineer, providing an OMT analysis document supplemented by a Petri net analysis facility.

- **Optimisation of code in tool kit.**

Most of the algorithms that were incorporated within the C++ Petri net class that was developed in this Thesis used brute force techniques to ensure correct operation. Many of the algorithms can be further optimised to reduce their memory and CPU requirements. This would result in the faster response that is required in industrial environments.

- **Use of genetic algorithms for optimisation of DEDSs.**

This Thesis developed a fast scheduling algorithm based on Petri net reduction techniques, the Branch and Bound algorithm and heuristics. Genetic algorithms, developed by Holland [75], are a well known technique for scheduling problems with an irregular and poorly defined search space. It would be interesting to investigate the application of Genetic algorithms to schedule DEDSs that are modelled on Petri nets.

Publications

Azzopardi D., Holding D.J., "Petri nets and OMT for modelling and analysis of DEDSs", submitted to IFAC Control Engineering Practice.

Azzopardi D., Holding D.J. *et al.*, "Object Oriented Petri net sythesis for modelling a semiconductor testing plant", IMACS-IEEE/SMC international conference on Computational Engineering in Systems Applications, pp. 374-378, CESA '96, Lille, France, July 1996.

Jiang J., Azzopardi D., Holding D.J., Carpenter G.F. and Sagoo J.S., "Real-time synchronisation of multi-axis high-speed machines, from SFC specification to Petri net verification.", IEE Proceedings - Control Theory and Applications, Vol. 143, Issue 2, pp. 164-170, March 1996.

Azzopardi D., Holding D.J., "Closed loop control of DEDS using timed Petri net models", Workshop on Analysis and Design of Event Driven Operations in Process Systems, Imperial College, London, UK, 10-11th April 1995.

Azzopardi D., Lloyd S., "Scheduling and Simulation of Batch Process Plant through Petri net modelling", Factory 2000: 4th IEE International conference on Advanced Factory Automation, No. 398, pp. 273-279, York, 3-5th October 1994.

Azzopardi D., Lloyd S., "Reduction of Search Space for scheduling of Batch Process Plant", 2nd IFAC/IFIP/IFORS International Workshop on Intelligent Manufacturing Systems, pp. 433-438, Vienna, June 10-13th, 1994.

Azzopardi D., "Scheduling and Simulation of Batch Process Plant through Petri net modelling", M.Phil. Thesis, School of Engineering, University of Sussex, Brighton, UK, November 1993.

References

[Adams et. al 88] Adams J., Balas E. *et al.*, "The shifting bottleneck procedure for job shop scheduling", Management Science, Iss. 34, pp. 391-401, 1988.

[Agerwala and Choed-Amphai 78] Agerwala T., Choed-Amphai Y., "A synthesis rule for concurrent systems", Proceedings of the Design Automation Conference, pp. 305-311, 1978.

[Alla 95] Alla H., "Modelling and Simulation of Event-Driven Systems by Petri Nets", Workshop on Analysis and Design of Event-Driven Operations in Process Systems, 10-11 April, Imperial College, Centre for Process Systems Engineering, London, 1995.

[Alpern 85] Alpern B., Schneider F.B., "Defining Liveness", Information Processing Letters, Vol. 21, No.4, pp. 181-185, 1985.

[Andreu *et al.* 94] Andreu D., Pascal J.C. *et al.*, "Batch process modelling using Petri nets", IEEE International conference on Systems, Man and Cybernetics, October 2-5, San Antonio, USA, pp. 314-319, 1994.

[Azzopardi *et al.* 96] Azzopardi D., Holding D.J. *et al.*, "Object Oriented Petri net sythesis for modelling a semiconductor testing plant", IMACS-IEEE/SMC international conference on Computational Engineering in Systems Applications, CESA '96, pp. 374-378, Lille, France, July 1996.

[Azzopardi and Holding 95] Azzopardi D., Holding D.J., "Closed loop control of DEDS using timed Petri Net models", Workshop on Analysis and Design of Event-Driven operations in Process Systems, Centre for Process Systems Engineering, Imperial College, UK, 10-11 April 1995.

[Azzopardi and Holding 94a] Azzopardi D., Lloyd S., "Reduction of Search Space for Scheduling of Multi-Product Batch Process Plant", 2nd IFAC/IFIP/IFORS Workshop on Intelligent Manufacturing Systems, pp. 433-438, 13-15 June 1994, Vienna, Austria, 1994a.

[Azzopardi and Lloyd 94b] Azzopardi D., Lloyd S., "Scheduling and Simulation of Multi-Product Batch Process Plant Through Petri net Modelling", Factory 2000: Fourth International Conference on Advanced Factory Automation, No. 398, pp. 273-249, 3-5th October 1994, University of York, UK, 1994b.

[Azzopardi 93] Azzopardi D., "Scheduling and Simulation of Multi-Product Batch Process Plant Through Petri net Modelling", M.Phil. Thesis, University of Sussex, Brighton, UK. 1993.

[Baker 74] Baker K.R., Introduction to Sequencing and Scheduling, Wiley, 0-471-04555-1, 1974.

[Barad and Sipper 88] Barad M., Sipper D., "Flexibility in Manufacturing Systems: Definitions and Petri net modelling", International Journal on Production Research, Vol.26, No.2, pp. 237-48, 1988.

[Berge 62] Berge C., The theory of graphs, NY, John Wiley & Sons, 1962.

[Bertholet 86] Bertholet G., "Checking properties of nets using transformations", Advances in Petri nets 1985, Vol. 222, pp. 19-40, 1986.

[Booch 86] Booch G., "Object-Oriented Development", IEEE Transactions in Software Engineering, Vol 12, Iss. 2, pp 211-221, 1986.

[Booch 91] Booch G., Object-Oriented Design With Applications, Benjamin Cummings, 1991.

[Booch 94] Booch G., Object-oriented analysis and design, Benjamin Cummings, 0-8053-5340-2, 1994.

[Carlier *et al.* 85] Carlier J., Chretienne Ph. *et al.*, "Modelling Scheduling problems with Timed Petri Nets", Lecture Notes in Computer Science, Vol.188, pp.62-82, 1985.

[Cassandras 93] Cassandras C.C., Discrete Event Systems - Modelling and Performance Analysis, Asken Associates, 0-256-11212-6, 1993.

[Chang and Liao 94] Chang S.-C., Liao D.-Y., "Scheduling Flexible Flow Ships with No Setup Effects", IEEE Transactions on Robotics and Automation, Vol. 10, No.2 , pp. 112-122, April 1994.

[Chen 76] Chen P.P.S., "The Entity-Relationship model - towards a unified view of data", ACM Transactions on Database systems 1, March 1976.

[Coad and Yourdon 90] Coad P., Yourdon E., Object-Oriented Analysis, 2nd ed. Englewood Cliffs, NJ. Prentice Hall, 1990.

[Cooling 91] Cooling J.E., Software Design for Real-Time Systems, Chapman and Hall, 0-412-34180-8, 1991.

[Cormen *et al.* 93] Cormen T.C., Leiserson C.E., Rivest R.L., Introduction to Algorithms, MIT Press, 0-262-03141-8, 1993.

[Courtoise *et al.* 71] Courtois P., Heymans F. *et al.*, "Concurrent control with 'readers' and 'writers'", Communication of the ACM, Vol. 14, No. 10, pp. 667-668, 1971.

[Courvoisier *et al.* 83] Courvoisier M., Valette R. *et al.*, "A Programmable Logic Controller Based on a High Level Specification Tool", Proceedings of IECON, San Fransisco, CA, pp. 174-179, 1983.

[David and Alla 92] David R., Alla H., "Petri nets and Grafcet: Tools for modelling Discrete Event Systems", Prentice-Hall, 1992.

[Davis 88] Davis A.M., "A Comparison of Techniques for the Specification of External System Behavior", Communications of the ACM, Vol. 31, No.9, pp.1098-1115, 1988.

[DeMarco 78] DeMarco T., Structured Analysis and System Specification, Englewood Cliffs, NJ Yourdon Press/Prentice-Hall, 1978.

[DerBeek 94] DerBeeck M., "A comparison of statechart variants", LNCS vol 863, Formal Techniques in Real-Time and Fault-Tolerant Systems, pp.128-149, 1994.

[deRoever 94] deRoever W.P., Foundations of Computer Science: Leaving the Ivory Tower, EATCS Bull, 1991.

[Desrochers and Al-Jaar 95] Desrochers A.A., Al-Jaar R.Y., Applications of Petri Nets in Manufacturing Systems, New York, IEEE, 0-87942-295-5, 1995.

[Dubois and Stecke 83] Dubois D., Stecke K.E., "Using Petri Nets to represent Production Processes", Proceedings of the 22nd IEEE Conference on Decision and Control, Vol. 3, pp. 1062-1067, 1983.

[Firesmith 93] Firesmith D.G., Object Oriented Requirement Analysis and Logical design: A Software Engineering Approach, Wiley, 1993.

[Fraser *et al.* 91] Fraser M.D., Kumar K. *et al.*, "Informal and Formal Requirements Specification Languages: Bridging the Gap", IEEE Transactions on Software Engineering, Vol.17, No.5, 1991.

[Galton 87] Galton A., Temporal logics and their applications, Academic Press, 0-12-274060-2, 1987

[Genrich and Lautenbach 81] Genrich H.J., Lautenbach K., "System modelling with high-level Petri nets", Theoretical Computer Science, Vol. 13, pp. 109-136, 1981.

[Giua and DiCesare 94b] Giua A., DiCesare F., "Blocking and Controllability of Petri nets in Supervisory control", IEEE Transactions on Automatic Control, Vol. 39, No. 4, pp. 818-823, 1994b.

[Giua and Dicesare 94a] Giua A., DiCesare F., "Stuctural Analysis for Supervisory Control", IEEE Transactions on Robotics and Automation, Vol.10, No.2, pp.185-195, 1994a.

[Goldsack and Finklestein 91] Goldsack S.J., Finkestein A.C.W., "Requirements engineering for real-time systems", Software Engineering Journal, Vol.6, No.3, pp. 101-115, May 1991.

[Hack 72] Hack, Analysis of Production Schemata by Petri nets, TR-94, MIT, Boston, 1972.

[Hall 90] Hall A., "Seven Myths of Formal Methods", IEEE Software, Vol. 7, No. 5, pp. 11-19, September 1990.

[Hamilton 78] Hamilton A.G., "Logic for Mathematicians", Cambridge University Press, 0-521-21838-1, 1978.

[Harel *et al.* 90] Harel D., Lachover H. *et al.*, "Statemate: a working environment for the development of complex reactive systems", IEEE Transactions on Software Engineering, 16, pp.403-414, 1990.

[Harel 87] Harel D., "Statecharts: A formalism for complex systems", Sci. Comp. Progr., 8, pp. 231-274, 1987.

[Hatono *et al.* 91] Hatono I., Yamagata K. *et al.*, "Modelling and on-line scheduling of Flexible manufacturing systems using stochastic Petri nets", IEEE Transactions on Software Engineering, Vol.17, No.2 , pp. 126-32, 1991.

[Healey 75] Healey M., "Principles of Automatic Control", Hodder and Stoughton, 0-340-17671-7, 1975.

[Hill and Holding 90] Hill, M. R., Holding D.J., "The modelling, simulation, and analysis of commit protocols in distributed computing systems", Proc. UKSC Conference on Computer Simulation, Brighton, UK, pp. 207-212, 1990.

[Hoare 85] Hoare C.A.R., Communicating Sequential Processes, New Jersey, Prentice-Hall, 1985.

[Hoare 78] Hoare C.A.R., "Communicating Sequential Processes", Communications of the ACM, Vol 21, No.8, pp. 666-677, August 1978.

[Holding *et al.* 96] Holding D.J., Jiang J. *et al.*, "A unified approach to the design of synchronisation logic for a hybrid control system", Proc 13th IFAC World Congress, IFAC'96, San Fransisco, 1996.

[Holding *et al.* 95] Holding D.J., Jiang, J. *et al.*, "A rule-based approach to the software synchronisation of intermittently synchronised drives", Proc IFAC Workshop on Motion Control, Munich, Germany, pp. 461-468, 1995.

[Holland 75] Holland J. H., Adaptation in Natural and Artificial Systems. University of Michigan Press (1975). Reprinted by MIT Press, 1992.

[Holloway and Krogh 90] Holloway H.E., Krogh B.H., "Synthesis of Feedback Control Logic for a Class of Controlled Petri nets", IEEE Transactions on Automatic Control, Vol. 35, No. 5, pp. 514-523, 1990.

[Hopcroft and Ullmann 79] Hopcroft J.E., Ullman J.D., Introduction to Automata Theory, Languages and Computation, Reading, Mass, Addison-Wesley, 1979.

[Hughes and Cresswell 68] Hughes G.E., Cresswell M.J., "An Introduction to Modal Logic", Methuen & Co., London, 1968.

[Jackson 83] Jackson M.A., System Development, Englewood Cliffs, New Jersey: Prentice-Hall International, 1983.

[Jacobsen 87] Jacobsen I., "Object Oriented development in an industrial environment", OOPSLA '87 as ACM SIGPLAN 22, Vol. 22, Iss. 12, pp. 183-191, 1987.

[Jeng and DiCesare 93] Jeng M.D., DiCesare F., "A Review of synthesis techniques for Petri nets with applications to automated manufacturing systems", IEEE Transactions on System, man and cybernetics, Vol. 23, No. 1, pp.301-312, 1993.

[Jensen 81] Jensen K., "Colored Petri nets and the Invariant method", Theoretical Computer Science, Vol.14, 1981.

[Jiang *et al.* 96] Jiang J., Azzopardi D. *et al.*, "Real-time synchronisation of multi-axis high-speed machines, from SFC specification to Petri net verification", IEE Proceedings - Control Theory and Applications, Vol. 143, Issue 2, pp. 164-170, March 1996.

[Jiang 95] Jiang J., "Development of Real-Time Process Control Systems Using Formal Techniques", PhD Thesis, Aston University, Birmingham, UK, April 1995.

[Jiang and Holding 96] Jiang J., Holding D.J., "The formalisation and analysis of sequential function charts using a Petri net approach", Proceedings of the 13th IFAC World Congress, IFAC'96, San Francisco, 1996.

[Joseph 89] Joseph M., Goswami A., "Formal Description of Real-Time Systems: A Review", Information and Software Technology, Vol.31, No.2, pp.67-76, 1989.

[Kripke 63] Kripke S., "Semantical considerations on modal logic", Acta Philosophica Fennica, Vol. 16, pp. 88-94, 1963.

[Krogh and Beck 86] Krogh B.H., Beck C.L., "Synthesis of place/transition nets for simulation and control of manufacturing systems", Proceedings of the 4th IFAC/IFORS Symposium on Large Scale Systems, Zurich, 1986.

[Krogh 87] Krogh B.H., "Controlled Petri nets and maximally permissive feedback logic", Proc. 25th Ann. Allerton Conference, Univ. Illinois, Urbana, 1987.

[Krogh *et al.* 88] Krogh BH., Willson R. *et al.*, "Automated generation and evaluation of control programs for discrete manufacturing processes", Renseelaer's 1st Int. Conference on Computer Integrated Manufacturing, Troy NY, pp 92-99, 1988.

[Ku *et al.* 87] Ku H., Rajagoplan D. *et al.*, "Scheduling in batch processes", Chemical Engineering Progress, pp. 35-45, August 1987.

[Lau 91] Lau Y.K.H., "Towards a Unified Methodology for the Design and Development of Distributed Control System Software", D.Phil. Thesis, Oxford University, 1991.

[Le Bail *et al.* 91] Le Bail J., Alla H., David R., "Hybrid Petri nets", 1st European Control Conference, Grenoble, pp. 187-191, 1991

[Lee and DiCesare 94a] Lee D.Y., DiCesare F., "Scheduling of Flexible Manufacturing Systems Using Petri Nets and Heuristic Search", IEEE Transactions on Robotics and Automation, Vol.10, No.2, pp. 123-132, 1994a.

[Lee and DiCesare 92] Lee DY., DiCesare F., "Experimental study of a heuristic function for FMS scheduling", Proc. Japan-USA Symposium on Flexible Automation, San Fransisco, CA, Vol.2, pp. 1171-78, 1992.

[Lee-Kwang 87] Lee-Kwang H., Favrel J. *et al.*, "Generalised Petri net reduction method", IEEE Transactions on Systems, Man and Cybernetics, Vol. SMC-17, pp. 297-303, 1987.

[Leveson 86] Leveson N.G., "Software Safety: Why, What and How", Computing Surveys, Vol. 18, No. 2, pp. 125-163, 1986.

[Maccarthy and Liu 93] Maccarthy B.L., Liu J., "Addressing the gap in scheduling research: a review of optimization and heuristic methods in production scheduling", International Journal of Production Research, Vol. 31, No.1, pp 59-79, 1993.

[Manna and Pnueli 83] Manna Z., Pnueli A., Verification of Concurrent Programs: A Temporal Proof System, Technical Report, Stanford University, 1983.

[Martinez and Silva 88] Martinez J., Muro PR. *et al.*, "Merging artificial intelligence techniques and Petri nets for real time scheduling and control of production systems", Proceedings of the 12th IMACS World Congress on Scientific Computation, Paris, pp. 528-531, 1988.

[Martinez and Silva 82] Martinez J., Silva M., "A simple and fast algorithm to obtain all invariants of a generalized Petri net", Second European Workshop on Application and Theory of Petri nets, Springer-Verlag, pp. 301-310, 1982.

[McConnell 93] McConnell S., Code Complete, Microsoft Press, 1-55615-484-4, 1993.

[Meyer 88] Meyer B., Object-Oriented Software Construction, Prentice Hall International, 1988.

[Murata and Komoda 87] Murata T., Komoda N., "Liveness analysis of sequence control specification described in capacity designated Petri nets using reduction", Proceedings of the IEEE International Conference on Robotics and Automation, Raleigh, North Carolina, pp. 1960-1965, 1987.

[Murata 89] Murata T., "Petri Nets : Properties, Analysis and Applications", Proceedings of the IEEE, vol. 77, no.4, 1989.

[Narahari and Viswanadham 85] Narahari Y., Viswanadham N., "A Petri net approach to the modelling and analysis of flexible manufacturing systems", Annals of Operations Research, Vol. 3 , pp. 449-472, 1985.

[Ogata 90] Ogata K., Modern Control Engineering, Prentice-Hall, 0-13-598731-8, 1990.

[Olderog 86] Olderog E.R., Hoare C.A.R., "Specification-Oriented Semantics for Communicating Processes", ACTA Informatica, Vol. 23, No. 1, pp. 9-66, 1986.

[Ostroff 89] Ostroff J., Temporal Logic for Real-Time Systems, Research Studies Press, 1989.

[Ostroff 92] Ostroff J.S., "Formal Methods for the Specification and Design of Real-Time Safety Critical Systems", Journal of Systems Software, Vol.18, pp.33-60, 1992.

[Pearl 84] Pearl J., Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley, 0-201-05594-5, 1984.

[Peterson 81] Peterson J.L., Petri Net Theory and the Modelling of Systems, Prentice-Hall, 1981.

[Petri 62] Petri C., "Kommunikation min Automaten", Germany, Ph.D. Dissertation, University of Bonn, 1962.

[Ramadge and Wonham 87] Ramadge PJ., Wonham WM., "Supervisory Control of a class of discrete event processes", SIAM Journal of Control Optimisation, 25(1), pp. 206-230, 1987.

[Ramadge and Wonham 89] Ramadge PJ., Wonham WM., "The Control of discrete event systems", Proc. IEEE, Vol. 77, No.1, pp. 81-98, January 1989.

[Reisig 82] Reisig W., Petri Nets, Springer-Verlag, 3-540-13723-8, 1982.

[Rescher and Urquhart 71] Rescher N., Urquhart A., Temporal logic, Springer-Verlag, 1971.

[Rumbaugh *et al.* 91] Rumbaugh J., Blaha M. *et al.*, "Object-Oriented Modelling and Design", USA, Prentice-Hall, 0-13-629841-9, 1991.

[Sagoo and Holding 90] Sagoo J.S., Holding D.J., "The specification and design of hard real-time systems using timed and temporal Petri nets", Microprocessing and Microprogramming, (30), pp. 389-396, 1990.

[Sagoo 92] Sagoo J.S., "The development of hard real-time systems using a formal approach", PhD Thesis, Aston University, Birmingham, UK, 1992.

[Scholefield 90] Scholefield D.J., "The Formal Development of Real-Time Systems: A Review", Technical Report, University of York, UK, 1990.

[Seidewitz 89] Seidewitz E., Object-Oriented Systems Analysis, NASA, 1989.

[Shen *et al.* 92] Shen L., Chen Q. *et al.*, "Truncation of Petri Net models for simplifying computation of optimum scheduling problems", Computers in Industry, Vol.20, No.1, pp25-43, 1992.

[Shlaer and Mellor 92] Shlaer S., Mellor S., Object Lifecycles: Modelling the world in states, Yourdon Press, 0-13-629940-7, 1992.

[Shlaer and Mellor 88] Shlaer S., Mellor S.J., Object-Oriented Systems Analysis: Modeling the World in Data, Yourdon Press, 1988.

[Sifakis 78] Sifakis J., "Performance Evaluation of systems using nets", LNCS Net Theory and Applications, Vol. 84, pp.307-319, 1978.

[Skeen 83] Skeen D., Stonebraker M., "A formal model of crash recovery in a distributed system", IEEE Transactions of Software Engineering, vol. 9, No. 3, pp. 219-228, 1983.

[Sobh *et al.* 94] Sobh M., Owen C.J. *et al.*, "A Subject-Indexed Bibliography of Discrete Event Dynamic Systems", IEEE Robotics and Automation Magazine, Vol.1, No.2, pp. 14-20, June 1994.

[Spiegel 80] Spiegel M.R., Advanced Mathematics for Engineers and Scientists, McGraw-Hill, 0-07-084355-4, 1980.

[Stroustrup 94] Stroustrup B, The C++ Programming Language, Addison Wesley, 0-201-53992-6, 1994.

[Suzuki and Murata 83] Suzuki I., Murata T., "A Method for stepwise refinements and abstractions of Petri Nets", Journal of Computer and Systems Science, No. 27, pp.-51-76, 1983.

[Suzuki and Murata 82] Suzuki I., Murata T., "Stepwise refinements of transitions and places", Application and Theory of Petri nets, Vol. 52, pp. 136-141, 1982.

[Valette 79] Valette R., "Analysis of Petri Nets by stepwise refinements", Journal of Computer and System Science, No.18, pp. 35-46, 1979.

[Valette *et al.* 85] Valette R., Courvoisier M. *et al.*, "Putting Petri nets to work for controlling flexible manufacturing systems", Proceedings of the International Symposium on Circuits & Systems, Kyoto, Japan, pp. 929-932, 1985.

[Valette 95] Valette R., "Petri nets for control and monitoring: specification, verification, implementation", Workshop on Analysis and Design of Event-Driven Operations in Process Systems, 10-11 April, Imperial College, Centre for Process Systems Engineering, London, 1995.

[Viswanadham *et al.* 95] Viswanadham N., Narahari Y. *et al.*, "Deadlock Prevention and Deadlock Avoidance in Flexible Manufacturing Systems using Petri Net Models", IEEE Transactions on Robotics and Automation, vol.6, No.6, pp 713-723, 1990.

[Ward and Mellor 85] Ward P., Mellor S., Structural Development for Real-Time Systems, New York, Yourdon Press, 1985.

[Wirth 71] Wirth N., "Program Development by Stepwise Refinement", Communications of the ACM 14, No. 4, pp. 221-27, 1971.

[Yourdon 79] Yourdon E., Constantine E., Structured Design, Englewood Cliffs: New Jersey: Yourdon Press, 1979.

[Yourdon 89] Yourdon E., Modern Structural Analysis, Englewood-Cliffs, New Jerse: Yourdon Press, 1989.

[Zhang 92] Zhang D., "Planning using timed Pr/T Nets", Proc. Japan USA Symposium on Flexible Automation, Dept. CS, California State University, San Fransisco, CA, Vol.2, pp.1179-1184, 1992.

[Zhou and DiCesare 92] Zhou M.C., Dicesare F. *et al.*, "A Hybrid Methodology for synthesis of Petri Net Models for Manufacturing Systems", IEEE Transactions on Robotics and Automation, Vol.8, No.3, pp 350-61, 1992.

[Zhou and DiCesare 91] Zhou M.C., Dicesare F., "Parallel and Sequential Mutual Exclusions for Petri Net modelling of Manufacturing Systems with shared Resources", IEEE Transactions on Robotics and Automation, Vol.7, No.4, pp 515-27, 1991.

[Zhou and DiCesare 93] Zhou MC., DiCesare F., "Petri net Synthesis fo Discrete Event Control of Manufacturing Systems", Kluwer Academic Publishers, 0-7923-9289-2, 1993.

[Zurawski and Zhou 94] Zurawski R., Zhou MC., "Petri Nets and Industrial Applications: A Tutorial", IEEE Transactions on Industrial Electronics, Vol. 41, No. 6, December 1994.

Appendix A

Experimental results

This appendix contains tables that show the number of iterations required by the following algorithms to schedule plants with n jobs as described in Chapter 4.

Algorithm A Branch and Bound
Algorithm B Branch and Bound AND 3 heuristics
Algorithm C Petri net reduction AND Branch and Bound AND 3 heuristics

Scheduling 2 resources:

n	A	B	C
2	26	22	14
2	22	12	12
2	26	20	14
2	26	20	14
2	21	12	11
2	24	12	11
2	26	20	12
2	24	12	11
2	25	20	14
2	28	20	12
3	111	82	33
3	51	38	16
3	156	100	68
3	104	37	27
3	105	35	37
3	134	84	36
3	153	96	58
3	159	68	68
3	157	109	49
3	66	27	24
4	368	196	45
4	480	276	115
4	507	230	144
4	719	419	223
4	693	448	138
4	699	475	263
4	548	439	75
4	918	666	349
4	626	518	137
4	509	148	138
5	4400	3162	739
5	8219	1356	2710
5	3298	2569	504
5	7321	1300	1833
5	3581	1259	223
5	4422	2482	1201
5	5812	4265	1195
5	4368	2748	1276
5	5987	3567	904
5	3956	3321	953
6	60398	14447	629
6	70807	36008	21909
6	91717	19620	27876
6	99077	30520	5645
6	70523	51676	7985
6	76381	13653	14948
6	45738	5589	4234

Average number of iterations for scheduling a plant with 2 resources

n	A	B	C
2	24	17	12
3	119	67	41
4	606	381	162
5	5136	2602	1153
6	73520	24501	11889

Scheduling 3 resources

n	A	B	C
2	43	28	20
2	34	16	18
2	38	16	18
2	34	16	18
2	55	18	25
2	52	30	25
2	51	30	19
2	34	16	18
2	38	16	15
2	35	16	15
3	279	33	66
3	309	210	130
3	312	51	90
3	200	95	43
3	849	187	116
3	281	173	108
3	161	78	59
3	279	64	51
3	190	49	45
3	461	185	145
4	4799	397	184
4	3184	1047	1038
4	4841	1405	1548
4	2354	236	197
4	4661	1289	1259
4	2038	527	280
4	6356	911	1092
4	3012	485	476
4	5379	1722	882
4	3690	1204	789
5	21913	8719	6274
5	39508	6207	1236
5	20143	11581	5766
5	36603	7437	3980
5	21030	10824	509
5	34247	23046	10646
5	19151	6223	795
5	38147	3548	2253
5	29004	12025	1974
5	18569	4160	5400
6	1408213	297171	41597
6	1187764	103937	71395
6	310876	73025	31484
6	1237750	246213	62856
6	638561	11858	82125
6	463504	24831	7885
6	310141	43733	11997

Average number of iterations for scheduling a plant with 3 shared resources

n	A	B	C
2	41	20	19
3	332	112	85
4	4031	922	774
5	27831	9377	3883
6	793829	114395	44191

Scheduling 4 resources

n	A	B	C
2	32	18	11
2	143	48	33
2	76	22	26
2	50	20	14
2	46	20	15
2	82	54	28
2	76	22	26
2	32	18	15
2	67	48	29
2	31	18	12
3	940	45	80
3	296	39	55
3	491	41	44
3	517	307	160
3	390	63	74
3	967	400	347
3	274	75	82
3	375	182	98
3	333	41	84
3	745	196	173
4	5142	1052	788
4	19513	9494	4461
4	4785	1114	842
4	9108	2969	1961
4	2057	693	227
4	13056	2548	5011
4	9231	137	1395
4	16428	517	194
4	15678	1312	3333
5	130989	7685	23527
5	702922	215549	35701
5	161138	31789	35738
5	275323	115859	63725
5	149991	35602	25124
5	269202	205942	7834
5	785319	270925	32094
5	137206	40908	11783

Average number of iterations for scheduling a plant with 4 shared resources

n	A	B	C
2	63	28	20
3	532	138	119
4	10555	2204	2023
5	326511	115532	29440

Scheduling 5 resources

n	A	B	C
2	65	24	16
2	124	108	51
2	40	22	14
2	85	26	17
2	40	22	13
2	62	24	19
2	225	94	43
2	110	48	33
2	90	26	30
2	39	22	14
3	1510	324	176
3	853	47	87
3	781	58	62
3	1352	120	156
3	2427	2096	320
4	16034	1404	134
4	38820	887	332
4	15731	3768	706
4	39610	588	1645
4	3258	1359	453
5	1555860	740188	32418
5	2082942	502316	60635
5	71340	698	880
5	557353	172296	48343
5	298114	23028	44377

Average number of iterations for scheduling a plant with 5 shared resources

n	A	B	C
2	88	41	25
3	1384	529	160
4	22690.6	1601.2	654
5	913121	287705	37330

Scheduling 6 resources

n	A	B	C
2	151	30	36
2	101	30	21
2	48	26	16
2	282	68	34
2	48	26	15
2	47	26	17
2	128	32	18
2	64	28	19
2	202	32	34
2	72	28	19
3	1896	76	87
3	491	51	57
3	572	88	65
3	611	51	40
3	2148	314	427
4	98742	26584	6944
4	22871	2573	2372
4	25326	1435	386
4	272722	14106	1724
4	35596	9616	2891

Average number of iterations for scheduling a plant with 6 shared resources

n	A	B	C
2	114	32	22
3	1143	116	135
4	91051	10862	2863
5	2470153	612616	45669

Scheduling 7 resources

n	A	B	C
2	242	68	49
2	96	32	21
2	59	32	18
2	74	32	20
2	137	34	25
2	222	36	60
2	90	32	21
2	102	32	40
2	266	36	35
2	94	32	20
3	15643	1495	1675
3	3215	209	142
3	27581	3295	516
3	1915	92	77
3	1029	57	61
3	2051	571	359
3	1798	74	72
3	715	72	102
3	2332	699	693
3	7533	112	155
4	287117	205	427
4	370997	5671	2201
4	133879	64142	4763
4	20405	1712	296
4	197547	118604	9727
4	106185	28166	4138
4	69298	553	393
4	86354	49806	30426
4	13682	166	223
4	57316	600	234

Average number of iterations for scheduling a plant with 7 shared resources

n	A	B	C
2	138	36	30
3	6381	667	385
4	134278	26962	5282

Appendix B

Petri net tool kit facilities

Petrinet.h - header file for petrinet class containing data structure and function declarations

```
class vector;                   // vector & matrix class are friends of
class matrix;                   // the petrinet class
class Int_Set;                  // Integer Set class
class Tree_node;                // Reachability Tree class
class timed_place_petrinet;
class timed_transition_petrinet;


class petrinet{

// a petri net data structure
protected:
        struct pnet{
                matrix H_pre, H_post;       //pre & post conditions
                matrix C;                   //incidence matrix
                vector X0, X;               //initial & current marking
                int p,t;            //number of places & transitions
                int n;              //reference count
                Int_Set non_safe_places; // set of 'not 1-bounded' places
                Int_Set fired_transitions; // set of fired transitions
        } *pn;

        void error(char *mesg1, const char *mesg2 = ""); // private function


public:

// * constructors
        petrinet(petrinet& x);

// * blank constructor for initialisation
        petrinet();

/*      create a petri net structure in memory from given type, pre & post
condition matrices, initial marking vector, current marking,
delay vector, number of places and transitions, set of non-safe places,
set of fired transitions.*/
        petrinet(int, matrix, matrix, vector, vector, vector, int, int, Int_Set, Int_Set);

// create a petri net structure in memory from a netlist file (14/12/94)
        petrinet(const char * netlistfile);
        petrinet read_netlist(char * netlistfile);

// create a petri net structure in memory from a set of files (mtrx)
        petrinet(char * precond, char * postcond, char * initmark, char * delay, int , int);

// create a petri net structure from a cabernet file
        petrinet(char * cabnetfile, char * flag);
```

```
// create a petri net structure from a graphics file
        petrinet(char * graphicsfile, int);

//  destructor
        ~petrinet();

// write a petrinet netlist file
        void write_netlist(char *filename, char *msg = "");

// save a petrinet structure as a set of files
        void write_pn_matrices(char * msg="");

// print pn_matrices on screen (just for testing purposes)
        void print_petrinet();

// equality operator
        virtual petrinet operator=(petrinet & rv);

// generate reachability tree
        virtual reach_tree reachability_tree();

// return the max place index
        virtual int    Max_place_number();

// return the max transition index
        virtual int    Max_transition_number();

// return the initial marking
        virtual vector Initial_marking();

// return the input places of transition
        virtual Int_Set Input_places(int);

//return the output places of transition
        virtual Int_Set Output_places(int);

// return the input transitions of a place
        virtual Int_Set Input_transitions(int);

// * return the output transitions of a place
        virtual Int_Set Output_transitions(int);

// return set of 'not 1-bounded' places **inh**
        virtual Int_Set non_safe_places();

// return set of fired transitions
        Int_Set fired_transitions();

// function to find list of enabled transitions at current marking (& time)
        vector find_enabled_transitions(vector);

// function to find new marking by firing transition
        vector find_new_marking(vector, const int);

// function to find new marking by firing sequence of transitions
        vector find_marking(vector, vector);

// calculates and prints p-invariants
        virtual void p_invariants();

// calculates and saves p-invariants
        virtual void save_p_invariants(char *);
```

```
// calculates and prints t-invariants
        virtual void t_invariants();

// calculates and saves t-invariants
        virtual void save_t_invariants(char *);

// reduction - place substitution
        petrinet sub_place(int);

// reduction - remove transition according to R2 and R3
        petrinet rem_trans(int);

// reduction -  returns index of place that can be substituted
        int reductionR1();

// reduction - returns index of neutral transition
        int reductionR2();

// reduction - returns index of identical transition
        int reductionR3();

// save the list of transitions that do not fire
        virtual void save_unfired_trans(char *);

// save list of places that are not 1-bounded
        virtual void save_unsafe_places(char *);
```

timed_place_petrinet.h - header file for timed place petrinet class containing data structure and function declerations

```
class vector;          // vector class is friend
class timed_place_petrinet:public petrinet{
friend class timed_transition_petrinet;
// the timed place petri net data structure is a petrinet
// data structure plus a delay vector
protected:
        struct tppnet{
                vector tp;      //delay vector
                // time_record, records the time of arrival of a token
                vector time_record;
                int n;          //reference count
        } *tppn;


public:

// * constructors
        timed_place_petrinet(timed_place_petrinet &);
        timed_place_petrinet();
        timed_place_petrinet(matrix, matrix, vector, int, int, vector);
        timed_place_petrinet(char * netlistfile);

// * destructor
        ~timed_place_petrinet();

// * equality operator
        timed_place_petrinet operator=(timed_place_petrinet & rv);
```

```
// * file handling
        timed_place_petrinet read_netlist(char * netlistfile);
        void write_netlist(char *filename, char *msg = "");

// * analysis
        reach_tree timed_reach_tree();
        vector find_enabled_transitions(vector, double);
        vector find_new_marking(vector, int, double);
        void reset_time();

// * scheduling to return a sequence of transitions for optimal operation
        vector schedule(vector              // brute force
        vector schedule_T1(vector           // with Test 1 ... concurrency detect
        vector schedule_T2(vector);         // with Test 2 ... repeated marking
        vector schedule_T3(vector);         // with Test 3 ... look ahead
        vector schedule_rss(vector);        // reduced search space (3 tests)

// * reduce tppn by removing places that represent operations and
// shifting the time delay to transitions => ttpn
        void reduce2ttpn(char *filename, char *msg="reduced from TPPN");
```

timed_transition_petrinet.h - header file for timed place petrinet class containing data structure and function declarations

```
class vector;          // vector class is friend
class timed_transition_petrinet:public petrinet{
friend class timed_place_petrinet;
protected:
// the timed transition petri net data structure is a petrinet
// data structure plus a delay vector
        struct ttpnet{
                vector tp;     //delay vector
                // time_record, records time at which trans was fired
                vector time_record;
                int n;         //reference count
        } *ttpn;


public:

// * constructors
        timed_transition_petrinet(timed_transition_petrinet &);
        timed_transition_petrinet();
        timed_transition_petrinet(char *filename);

// * destructor
        ~timed_transition_petrinet();

// * equality operator
        timed_transition_petrinet operator=(timed_transition_petrinet & rv);

// * file handling
        timed_transition_petrinet read_netlist(char * netlistfile);
        void write_netlist(char *filename, char *msg = "");

// * analysis
        reach_tree timed_reach_tree();
        vector schedule(vector);
        vector schedule_heuristics(vector);
        vector find_enabled_transitions(vector, double);
        vector find_new_marking(vector, int, double);
```

```
        void reset_time();
```

reach_tree.h - header file for timed place petrinet class containing data structure and function declerations

(NOTE: this class was developed by J.Jiang, Senior software engineer, Eurotherm controls Ltd., Worthing, UK)

```
class  Tree_node     {
friend class Int_Set;
friend class vector;
        struct  rtree_node
        {     //    REACHABILITY TREE NODE DATA STRUCTURE
                vector  marking;
                Int_Set enabled_trans;      // set of enabled trans
                Int_Set conflict_trans;     // set of conflicting trans

                // keep track of pre & post nodes
                int    pre, post;           // no of pre & post nodes
                vector  pre_nodes, pre_trans;
                vector  post_nodes, post_trans;
                int    time;                // time at which marking is reached
                int    n;                   // reference index, e.g. marking index
        }     *node;

public:

// constructors
        Tree_node(vector&, Int_Set&, Int_Set&, int, int, vector&, vector&,
          vector&, vector&, int, vector&, int);
        Tree_node();

// destructor
        ~Tree_node() { delete[] node;};

        Tree_node&     operator=(const Tree_node&);   // node assignment
        Tree_node(const Tree_node&);                  // copy constructor for node

        // Following functions are used to read/write each field of a node
        vector&         marking() { return node->marking;}
        Int_Set&        enabled_trans() { return node->enabled_trans;}
        Int_Set&        conflict_trans() { return node->conflict_trans;}
        int&            pre() { return node->pre;}
        int&            post() { return node->post;}
        vector&         pre_nodes() { return node->pre_nodes;}
        vector&         pre_trans() { return node->pre_trans;}
        vector&         post_nodes() { return node->post_nodes;}
        vector&         post_trans() { return node->post_trans;}
        int&            time() { return node->time;}
        int&            n() { return node->n;}
        void            Tree_node_print() const;          // display all fields of node
};


class reach_tree {     // reachability tree data structure
friend class Tree_node;
        Tree_node     *rt;
        int         counter;        // reachability tree node counter
        int         size;           // keep the size of allocated memory

public:                                 // constructor without argument
        reach_tree();                   // constructor with argument
        reach_tree(const int size);     // destructor
        ~reach_tree(){ delete[] rt; };
```

```
        void    reach_tree_append(Tree_node&);      // add a node into the tree
        void    reach_tree_delete(const int);       // delete a node with given code

        int     reach_tree_root() const;            // return the root node code
        void    reach_tree_print() const;           // display all information
        void    reach_tree_save(char *filename) const;          // save reach_tree
        int     number_of_nodes() const {return counter;};      // the number of nodes

        // given a node code, get a node
        Tree_node&      reach_tree_node(const int) const;

// replace a node with the given code in the tree by the given node
        void    node_replace(const int, Tree_node&);

// given a node code, read/write its marking
        vector&         marking(const int);

// given a node code, read/write its enabled transitions
        Int_Set&        enabled_trans(const int);

// given a node code, read/write its conflict transitions
        Int_Set&        conflict_trans(const int);

// given a node code, read/write its reached time
        int&            time(const int);

// given a node code, read/write its RFT
        vector&         RFT(const int);

// given a node code, read/write its pre
        int&            pre(const int);

// given a node code, read/write its post
        int&            post(const int);

// given a node code, read/write its pre_trans
        vector&         pre_trans(const int);

// given a node code, read/write its pre_nodes
        vector&         pre_nodes(const int);

// given a node code, read/write its post_trans
        vector&         post_trans(const int);

// given a node code, read/write its post_nodes
        vector&         post_nodes(const int);

// reach_tree assignment operator
        reach_tree&     operator=(const reach_tree&);
        reach_tree(const reach_tree&);  // copy constructor

        Int_Set concurrency_set(const int) const;

// to find out all the transitions which are enabled at the
// same time as the given transition
        void    dead_lock_markings() const;
        Int_Set concurrency_t_set(const int) const;

        int     tree_size() const { return size;};

// return a node with the given index in the tree
        Tree_node&      node(const int) const;
```

Appendix C

Abbreviations and Acronyms

AI	artificial intelligence
AT	set of alternative transitions
CDES	controlled discrete event system
CSP	communicating sequential processes
CPN	controlled Petri net
CPU	central processing unit
DEDS	discrete event dynamic system
DES	discrete event system
EC	elementary circuit
ECSM	elementary composed state machine
EP	elementary path
ER	entity - relationship
ET	set of enabled transitions
FMS	flexible manufacturing system
GUI	graphic user interface
OCT	object communication transition
OMT	object modelling technique
OO	object oriented
OOA	object oriented analysis
OOD	object oriented design
OOP	object oriented programming
PLC	programmable logic controller
PME	parallel mutual exclusion
P/T net	place/transition net
SBDM	system behaviour driven method
SEC	simple elementary circuit
SME	sequential mutual exclusion
SPP	solitary place path
STP	solitary transition path
TPPN	timed place Petri net
TPr/T net	timed predicate/transition net
XTPTN	extended timed place transition net