Development of Real-Time Process Control Systems Using Formal Techniques

Jingyue Jiang

Submitted for the degree of Doctor of Philosophy

THE UNIVERSITY OF ASTON IN BIRMINGHAM

April 1995

The University of Aston in Birmingham

Development of Real-Time Process Control Systems Using Formal Techniques

By Jingyue Jiang

Submitted for the degree of Doctor of Philosophy

April 1995

SUMMARY

A major application of computers has been to control physical processes in which the computer is embedded within some large physical process and is required to control concurrent physical processes. The main difficulty with these systems is their event-driven characteristics, which complicate their modelling and analysis. Although a number of researchers in the process system community have approached the problems of modelling and analysis of such systems, there is still a lack of standardised software development formalisms for the system (controller) development, particular at early stage of the system design cycle.

This research forms part of a larger research programme which is concerned with the development of real-time process-control systems in which software is used to control concurrent physical processes. The general objective of the research in this thesis is to investigate the use of formal techniques in the analysis of such systems at their early stages of development, with a particular bias towards an application to high speed machinery. Specifically, the research aims to generate a standardised software development formalism for real-time process-control systems, particular for software controller synthesis.

In this research, a graphical modelling formalism called Sequential Function Chart (SFC), a variant of Grafcet, is examined. SFC, which is defined in the international standard IEC1131 as a graphical description language, has been used widely in industry and has achieved an acceptable level of maturity and acceptance. A comparative study between SFC and Petri nets is presented in this thesis. To overcome identified inaccuracies in the SFC, a formal definition of the firing rules for SFC is given. To provide a framework in which SFC models can be analysed formally, an extended time-related Petri net model for SFC is proposed and the transformation method is defined.

The SFC notation lacks a systematic way of synthesising system models from the real world systems. Thus a standardised approach to the development of real-time process control systems is required such that the system (software) functional requirements can be identified, captured, analysed. A rule-based approach and a method called system behaviour driven method (SBDM) are proposed as a development formalism for real-time process-control systems.

Key words: Real-Time Process-Control Systems, SFC — Sequential Function Chart (or Grafcet), Petri Nets, Rule-Based Approach, Temporal Petri nets.

Acknowledgements

I wish to thank my supervisor Dr D. J. Holding for providing his guidance, support, and encouragement in all aspects of my research, particular for his perceptive criticism and excellent suggestions to the drafts of this thesis. I am also deeply indebted to Dr G. F. Carpenter and my colleagues in the IT (Information Technology) Group at Aston. Special thanks are due to my wife Yixian for being very patient and understanding and to my daughter Vicky. Their love and motivation of my family helped me finish my study.

Finally, I would like to take this opportunity to thank the Science and Engineering Research Council (SERC), Department of Trade and Industry (DTI), and Eurotherm Controls (U.K.) for funding me as a research officer on grant GR/R60666.

Contents

List of Figures

List of Tables

List of Acronyms

FSMs	Finite State Machines
SFC	Sequential Function Chart
TPTN	Timed Place Transition Nets
XTPTN	Extended Timed Place Transition Nets
SBDM	System Behaviour Driven Method
PLC	Programmable Logic Controllers
IEC	International Electrotechnical Commission
PrT	Predicate Transition
C/E	Condition Event
LTL	Linear Temporal Logic
BTL	Branching Temporal Logic
TURT	Token Unavailable Remaining Time
LNCS	Lecture Notes in Computer Science

Chapter 1 Introduction

1.1 Real-Time Systems

Developments of modern technology, in the form of computers and communication networks, have increasingly created man-made systems which are not easily described by conventional difference and differential equations [Ho 89]. These systems include computer networks, computer controlled manufacturing systems, power plant control systems, patient monitoring systems, weapon control systems, robotics, traffic and transportation control systems. The behaviour of these systems is characterised by the inclusion of discrete events. Typical examples of discrete events are: turning off a pump or closing a valve when the level in a liquid tank reaches a predetermined value; or switching a motor off in response to the closure of a micro-switch indicating that some desired position has been reached. Such man-made systems are classified by researchers as Discrete Event Systems (DES) or Discrete Event Dynamic Systems (DEDS) [Ostroff 89, Ostroff and Wonham 90, Heymann 90] as opposed to the more familiar Continuous Variable Dynamic Systems (CVDS) [Ho 89, Cassandras and Ramadge 90]. The study of such systems involves the mathematical modelling and analysis of discrete event processes. A large variety of computer and communication systems can be described using discrete mathematics.

Some applications of computers to physical processes involve both continuous variables and discrete events, such systems are commonly called hybrid systems. Such systems are relatively complex and their description involves the mathematics of both continuous and discrete event systems. However, it is common to find that such systems include supervisory functions concerned with the sequencing and co-ordination of activities. By a process of abstraction, attention can be focused on the supervisory function or level, which usually concerns discrete events or decisions and can be modelled using discrete event mathematics.

A major field of application of computers has been to control physical processes, where the computer is interfaced directly to some physical process and is dedicated to controlling the operation of that process [Mok 83]. A key feature of such systems is that the computer is often embedded within some large physical process and is required to perform its function within specified time bounds dictated by the physical process. That is, the correctness of these systems depends not only on the logical results of the computation, but also on the time at which the results are produced. It is for this reason that such applications have become known as real-time systems [Mok 83, Jahanian and Mok 86, Goldsack and Finkestein 91, Kemmerer and Ghezzi 92, Shaw 92] or embedded systems [Zave 82, Burns and Wellings 91] or real-time process-control systems [Bologna and Leveson 86, Jaffe *et al* 91]. This thesis is essentially concerned with such systems. The terms "real-time", "embedded", and "real-time process-control" will be used interchangeably in this thesis.

Real-time systems are increasingly being designed and used in everyday life. Although such systems offer many benefits, there can also be some disadvantages. For example, in applications such as flight control and weapon control systems, malfunctioning could lead to loss of life. The potential consequence of system failure depends on the characteristics of the particular application. To distinguish between various levels of responsiveness demanded of a system, real-time systems are often distinguished as "*soft*" and "*hard*" [Mok 83, Shin 87, Faulk and Parnas 88, Burns and Wellings 91]. *Soft real-time systems* are those systems in which response times are important but the system will still function correctly if deadlines are occasionally missed. That is, a result that is delivered earlier than required is acceptable and a result that comes a little later than required is still usable. *Hard real-time systems* are those systems in which it is absolutely imperative that responses occur within a specified deadline. If the timing constraints are not met, i.e., the result generated by the computation is too early or too late, then the real-time system has failed. It is also necessary to distinguish between the severity of the consequence of a failure leads to a hazardous situation or even a possibly catastrophe, then the system is said to be safety critical [Leverson 86, McDermid and Thewlis 91]. For example, an aircraft control system would be classified as a safety-critical hard real-time system if a missed deadline could lead to a catastrophe. On the other hand, a pedestrian traffic control system would be classified -as far as responsiveness is concerned - as a soft real-time system because the intermittent delays are tolerant. All the systems described in this thesis are considered to be hard real-time systems unless otherwise specified.

1.2 Characteristics of Real-Time Systems

The original distinctions between conventional data processing systems and real-time process control systems have become blurred with the development of modern, responsive, data processing systems such as on-line point of sales and banking systems. However, it is possible to distinguish between such systems and hard real-time systems. The following describes some of characteristics found in hard real-time systems. Although each feature may not be unique to such systems, the complete set of features is typical of hard real-time systems [Ward and Mellor 85, Burns and Wellings 91]. The significance of the list is that any formalism for real-time system development must be powerful enough to describe a system with all of these characteristics.

Timing constraints

By the nature of real-time systems, the computer embedded within the system must respond to real-world events. Since the system will not perform satisfactorily if the time at which the result arrives does not satisfy the timing constraints of the physical processes under control, timing is one of most important features of real-time systems and also a major area of difference between real-time systems and data processing systems. For example, in an emergency case the nuclear power system must be shut down within strict timing constraints, otherwise it will result in the loss of life. This dramatic example illustrates that timing is a very important feature of real-time systems, particular for safety critical real-time systems.

Environment

A real-time system's timing requirements are usually defined by the physical processes under control (i.e., the environment) rather than by the computer. In general, a real-time computer must try to keep up with its host environment [Lin and Burke 92]. The environment under which a computer operates is an active component of the real-time system and the real-time system and its environment can be considered to form a synergistic pair [Shin 87].

Concurrency

The environment of a real-time system tends to consist of several coexisting external physical processes with which the computer software system must simultaneously interact. It is the very nature of these external physical processes that they exist in parallel, i.e., the activity of one physical process may occur simultaneously with other activity on another physical process. For example, a manufacturing system may consist of robots, conveyor belts, sensors, actuators which can have parallel activities. These parallel activities in real-time systems create some difficult problems because they need to interact or cooperate with each other. A major problem associated with real-time systems which exhibit concurrency is how to describe concurrent behaviour using appropriate notations and how to control concurrent behaviour using appropriate synchronisation and control mechanisms.

Size and Complexity

As computers become more powerful, more control functions previously performed only by human operators or hard equipment are assigned to computers. This means that the software system will become more complex. For example, the size of the on-board software on NASA's space shuttle is at least 30 times larger (and considerable more complex) than the software for the previous Saturn V [Leveson 86]. As the timings and concurrency must be considered in designing real-time systems, this leads to the fact that real-time systems are often more complex than data processing systems.

Efficiency

Although progress in hardware technology has made high-performance processors available, high-speed execution may not solve all the problems faced by real-time systems [Stankovic 88, Lin and Burke 92]. Just as "real-time" differs from concurrency, high efficiency is not a necessary feature of real-time system implementation although , generally speaking, efficiency is more important in real-time systems than in other types of systems. Efficiency heavily depends on the timing constraints associated with the physical process under control and begins to be important only when the physical process has a high computation requirement and must satisfy the demanding timing constraints.

Reliability and Safety

Reliability is defined as the probability that a system will perform its intended function for a specified period of time under a set of specified environmental conditions [Leveson 86]. In general, reliability is concerned with making a system *failure-free*. Since most embedded systems are primarily control-oriented, the failure of such a system may be very expensive either in terms of lost production, as in the case of process control, or in terms of human life, as in the extreme case of weapon control systems. Therefore, real-time systems must be extremely reliable.

Safety is a property of the "real world" [McDermid 90] which is defined as freedom from those states that can cause death, injury, occupational illness, damage to (or loss of) equipment (or property), or environmental harm [Leveson 86] . In general, safety is concerned with making a system *accident-free*.

Maintainability

Like any system, real-time systems need to be maintained because of changes occurring in the environment. When such changes occur during or after development, the maintenance of the real-time systems is often very difficult because the need to accommodate the new requirement may lead to changes which have ripple effects upon the system's concurrent behaviour and ability to satisfy time constraints [Faulk and Parnas 88].

1.3 Research Background

Recently, there has been a growing awareness of the role and importance of software in real-time systems [Bologna and Leveson 86, Shin 87, McDermid 90, McDermid and Thewlis 91, Wellings 91, Lin and Burke 92, Kemmerer and Ghezzi 92, Leveson and

Nenmann 93]. This is mainly due to the hazardous consequences which have been brought about by the malfunction of software in safety critical applications. For example, a software error caused a stationary robot to move suddenly with impressive speed to the edge of its operation area, crushing a nearby worker to death [Ostroff 92]. Formal mathematical techniques have long been recognised to be useful in overcoming some of the difficulties in the specification, design and implementation of software for complex real-time systems [Pnueli 86, Jahanian and Mok 86, Manna and Pnueli 88, Ostroff 89, Joseph and Goswami 89, Goldsack and Finkestein 91, Manna and Pnueli 92]. These methods allow precise specification and provide support for formal proofs which allow the correctness of design or implementation to be verified with respect to the specification.

This research forms part of a larger research programme which is concerned with the development of real-time systems in which software is used to control concurrent physical processes. Such software is primarily control-oriented and the controller is often embedded within some systems such as a manufacturing machine or a processing plant. The general objective of this thesis is to investigate the use of formal techniques in the analysis of such systems at their earlier stages of development, with a particular bias towards an application to high speed machinery. This thesis is more concerned with the software development issues rather than the physical processes under control. The physical processes will be considered only from the point of view of their software interface characteristics.

In real-time process-control, perhaps the most widely used methods of providing software control is to use programmable logic controllers (PLC) [Warnock 88, Crispin 90, Michel 90, Swainston 91, Joshi and Supinski 92]. The programmability of a PLC provides the flexibility required in control system design. For example, when a design change is made, it is only necessary to change the software system rather than to disconnect or re-route a signal wire. In the past years, there has been a growing interest in using high-level programming languages, instead of application-oriented low level languages or assembly language, for PLC system developments [Silva 89, Michel, 90, David and Alla 92, Falcione and Krogh 93, IEC 93]. The International standard IEC1131 Part 3: Programming Languages for Programmable Controllers [IEC 93] which is becoming increasingly influential in recent years, defines five different languages which target different levels of abstraction for PLC systems [David 91, Falcione and Krogh 93, IEE 93, Halang and Kramer 92, 94].

(i) Sequential Function Chart (SFC) is a graphical description language which is used to formulate the co-ordination and co-operation of asynchronous sequential processes, i.e. to capture a high-level description of sequential control logic of a system. SFC partitions a process-control software into a set of *steps* (states) and *transitions* (state transitions) interconnected by *directed links.* Each step is associated with a (set of) *action*(s), and each transition is associated with a *condition* which is normally an external event which initiates or governs the response of the system.

(ii) Function Block Diagram (FBD) is a graphical language inspired by diagrams of digital circuits, which is used to represent a certain module of overall functionality pre-defined for designing process-control software. A function block is a program unit (or a computational element) which takes inputs and produce one or more outputs. For example, function blocks may perform binary, numerical, analogue or character string processing. Each function block is depicted by a rectangular box and a function block diagram is represented as a network of function blocks connected by directed links.

(iii) Ladder Diagram (LD) is a graphical programming language which is based on the method used for describing relay logic circuits in the electrical/electronics industry. LD represents a convenient method of expressing logical statements (Boolean expressions).

(iv)Structural Text (ST) is a Pascal-based real-time programming language with certain features to facilitate real-time programming such as the duration literal, time data type, and the ability to deal with time and date. ST uses modules and tasks to describe the structure and to implement the behaviour of process-control software. ST provides rich data types and statements.

(v) Instruction List (IL) is an assembler-like low level programming language which is used to implement the process control software. For example, each instruction shall contain an *operator*, *with* one or more *operands* separated by commons.

These five languages have been used in industry and considerable experience has accumulated in the use of the low level notations. However, associated analysis techniques have not been well developed, and there is a significant lack of formal analysis methods. Recently, formal proof techniques have been applied to the verification of FBD and ST by Halang and Kramer [92, 94]. This work shows how to create function blocks, prove their correctness, and interconnect them so that the safety properties of the entire PLC software can be easily performed. Formal method has also been applied to verification of the safety and reliability of PLC software written in LD by Moon [94]. The method proposed by Moon consists of three parts: a system model, assertions, and a model checker. The *model* is a representation of a PLC's behaviour expressed in LD. *Assertions* are questions about the behaviour of the system, which are expressed in branching time temporal logic (see section 2.2.6 in Chapter 2). The *model checker*, a method developed based on branching time temporal logic by Clarke *et al* [86], determines the consistency of the model and assertions. However, formal techniques have not been used to analyse the Sequential Function Chart (SFC) which is often used to describe the control and synchronisation logic of process-control software at early development stage.

It is well known that the requirements specification is one of the first activities of software system development [Cohen *et al* 86, Sommerville 92]. The set of all requirements specifications forms the basis for subsequent development stages of the systems. The key problem of requirements specification is to formulate an adequate and correct set of requirements specifications. It has been recognised for a long time that errors made during this stage are the most difficult and expensive to correct. For example, software requirement errors can cost up to many times more to correct than errors introduced later in the software development process [Boehm 81]. Moreover requirements specifications may have impact on safety of physical processes under control [Leveson 86, Jaffe *et al* 91]. Therefore, techniques to provide adequate requirements specifications and to verify safety requirements early are of great importance.

This thesis mainly concentrates on the requirement specification stage because of the lack of adequate methods and tools in both requirements specification capturing and requirements specification analysis in process-control system developments, particular for PLC systems. The work presented in this thesis is specifically concerned with

i) investigating the discrete mathematical model recommended for real-time process-control system in the standard IEC1131 Part 3;

ii) developing analysis methods for the recommended model;

iii) developing a method for capturing the requirements specification of real-time control and synchronisation logic for real world systems;

iv) verifying the properties of the captured requirements specification.

1.4 Thesis Organisation

This thesis is roughly divided into three parts. The first part (Chapters 1-2) describes the background of the research and presents the environment and theoretical framework in which the research is embedded. The second part (Chapters 3-5) investigates the discrete mathematical model SFC which has been recommended by the standard IEC1131 Part 3. The third part of the thesis (Chapters 6-7) is concerned with the method for eliciting and representing functional requirements of real-time process-control systems and the application of the proposed method. Finally, conclusions of this research are drawn and recommendations for future research are pointed out. The following is a summary of each chapter.

Chapter two describes the background to the research and presents the mathematical foundation of models and techniques used to establish the basis for the following chapters. Emphasis is placed on the SFC model. Since SFC has evolved from another discrete event model called Grafcet [IEC848 88, David and Alla 92], the Grafcet model is discussed and a survey of past work of Grafcet is provided. To assist the discussion, considerable use is made of Petri net model theory [Peterson 81, Reisig 85]. The reason that Petri nets are used rather than other formal models is that Grafcet was originally inspired by Petri nets and thus contains many similar features. Finally, the chapter surveys the work of rule-based approach in both data processing systems and real-time process-control systems.

The detailed investigation of Grafcet is presented in chapter three. It is shown that Grafcet is a powerful model because of its features such as the simultaneous firing rule and implicit communication (dependency). However, problem also arises due to the implicit communication (dependency). Implicit communication makes it difficult to distinguish (or to trace) whether states occur in a particular intended order. A method is proposed to alleviate this deficiency.

Based on chapter three, a thorough comparative study of SFC and Petri nets is presented in chapter four. In this chapter, the evolution rules defined for SFC are revised to overcome the identified ambiguity. To provide a framework in which SFC description can be analysed formally, an extended time-related Petri net model for SFC and the transformation methods from SFC to the extended model are defined. This allows many of the existing formal analysis techniques developed for Petri nets to be applied to the analysis of transformed SFC.

Chapter five presents techniques for analysing an SFC design by transforming it into the extended Petri net model. Reachability graph analysis technique of Petri net model [Peterson 81, Reisig 85, Murata 89] is used to analyse the behaviour of the SFC system. To analyse the concurrent behaviour and temporal properties of a SFC system, trace theory [Mazurkiewicz 87, 88] and timing analysis techniques of Petri nets [Ramchandani 74, Merlin and Farber 76] are used.

Chapter six concentrates on expressing the functional requirements of real-time process-control systems in terms much closer to the "real world concepts" whilst still presenting them in an analytical framework. For this reason, a rule-based formalism and an elicitation method called the System Behaviour Driven Method (SBDM) are proposed. A rule-based formalism as a means of describing the functional requirements has been presented. Two different rule schemes are presented and the elicitation method for capturing the functional requirements is described in detail. Also, the formal techniques used to verify whether the rule-based descriptions reflects the required system behaviours are defined.

Chapter seven demonstrates the applications of the rule-based formalism and the SBDM method by examples, in which the proposed elicitation method (SBDM) is illustrated in a step by step fashion. The application of the formal techniques presented in Chapter six is also shown via the verification of the required system behaviours (specification) of the examples.

Finally, chapter eight summarises and evaluates the contribution of this research effort and the solutions introduced through rule-based formalism. In addition, areas of further research are recommended.

Chapter 2 Literature Review

2.1 Discrete Event Models and Paradigms

The problem of specifying the behaviour of a real-time system makes considerable demands on the descriptive powers and mathematical properties of any formal or semi-formal notation. This chapter examines various approaches to the specification of real-time systems and evaluates their suitability for describing hard real-time systems comprising sets of concurrent communicating processes.

In the past years, a variety of (discrete) formalisms have been proposed for real-time systems. These include Software Requirements Engineering Methodology (SREM) [Alford 77, 85], Structured Analysis /Real-Time (SA/RT) [Ward and Mellor 85], SCR/A7E specification [Heninger 80], VDM [Jones 86], Statecharts [Harel 87], transition axiom method [Lamport 83, 89], PAISLey [Zave 82], Z [Spivey 92], Temporal Logic [Pnueli 86, Manna and Pnueli 92], Real-time Logic (RTL) [Jahanian and Mok 86], Real-time Temporal Logic (RTTL) [Ostroff 89], Petri Nets [Peterson 81, Reisig 85], Time and Timed Petri nets [Ramchandani 74, Merlin and Farber 76], Grafcet [David and Alla 92], State Machines [Ostroff 89, Shaw 92], Extended State Machines (ESMs) [Ostroff 89] and Communicating Real-Time State Machines (CRSMs) [Shaw 92], and the process algebras CSP [Hoare 85], timed-CSP [Reed and Roscoe 86], and timed-CCS [Moller and Tofts 90].

Many formalisms for specifying the behaviour of systems differ because of their choices in semantic domains [Wing 90]. Some focus on just the states, some on just the events, and some on both. In state-based formalisms, the system is represented by the set of system states and transitions between system states; the behaviour of the system is described by the properties of states and how properties of some states depend on those of some other states. The typical examples on states are the state-based pre/postcondition formalisms VDM [Jones 86] and Z [Spivey 92]. In event-based formalisms, the system is expressed by allowable sequences of atomic events; each time an event occurs it may produce other events at a certain time. Typical examples on events are process algebra formalisms CSP (Communicating Sequential Processes) [Hoare 85] and CCS (Calculus of Communicating Systems) [Milner 89]. Formalisms which focus on both states and events include state machines [Ostroff 89, Shaw 92] and Petri nets [Peterson 81, Reisig 85, Murata 89].

State-based formalisms and event-based formalisms are different [Karam and Buhr 91] and are often considered as dual methods [Valette *et al* 85]. In fact, in state-based formalism the notion of event is derivative although the notion of state is fundamental. By contrast, in event-based formalism the state is derived notion although the notion of event is primary [Inan and Varaiya 88].

Although both state-based and event-based notations have been used in the software development of real-time process-control systems, "there are no models of discrete-event systems that are mathematically as concise or computationally as feasible as are differential equations for continuous variable dynamical systems. There is thus no agreement as to which is the best model, particularly for the purpose of control" [Fleming 88] (quoted from [Cao and Ho 90]). For this reason, only the formalisms within the category which encompass both states and events will be examined in detail in this chapter.

In real-time process-control systems, the behaviour of a system is typically described in terms of changes of the states of the systems, where each state transition results from the occurrence of an event which takes place at a discrete point in time [Ostroff 89, Heymann 90]. Thus, formalisms that focus on both states and events appear to be much more natural and straightforward to system designers and engineers because the behaviour of the system can be expressed in a simple and familiar way. Formalisms which focus on both concepts also offer advantages to the formal method community because relationships such as dependency and independency between events and states can be explicitly described. The reviewed formalisms include finite State Machines [Hopcroft and Ullman 79, Ostroff 89, Shaw 92], Statecharts [Harel 87], Grafcet [David and Alla 92], Petri nets [Peterson 81, Reisig 85], Time(d) Petri nets [Ranchandani 73, Merlin and Farber 76, Sifakis 80], and Temporal Petri nets [Suzuki and Lu 89, He and Lee 90, Sagoo and Holding 90, 91].

2.2 Mathematical Foundations and Review

2.2.1 Finite State Machines (FSMs)

A *finite automaton* (FA) is formally defined as a 5-tuple [Hopcroft and Ullman 79]

$$M = (Q, \Sigma, \delta, q_0, F) \tag{2.1}$$

where

(1) Q is a finite, non empty set of *states*;

(2) Σ is a finite *input alphabet*;

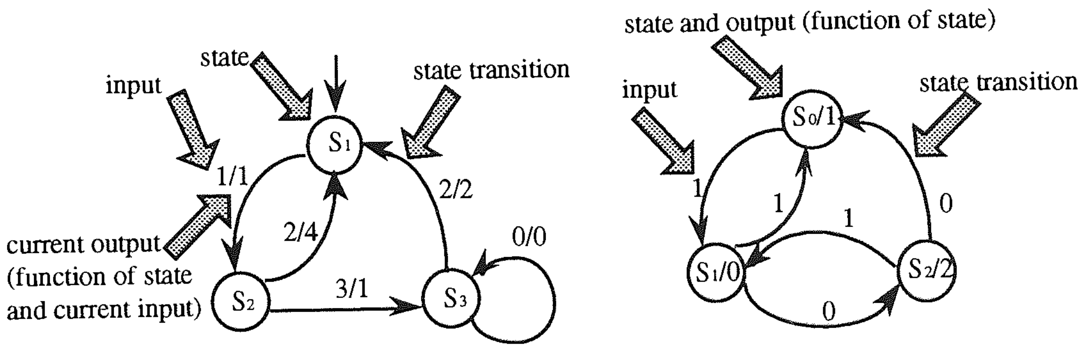
(3) $\delta : Q \times \Sigma \rightarrow Q$ is a mapping from $Q \times \Sigma$ to Q which is called the *state transition function*;

(4) $q_0 \in Q$ is the *initial state*; and

(5) $F \subseteq Q$ is the set of *final states*.

One limitation of the FA as defined above is that it has no output alphabet and its output is limited to a binary signal: "accept"/"don't accept." Models in which the output is chosen from some other alphabet have been considered. There are two distinct approaches; the output may be associated with the state (called a *Moore machine*) or with the transition (called a *Mealy machine*) [Hopcroft and Ullman 79].

A Mealy machine is formally defined as a six-tuple $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where Q, Σ, δ, and q_0 are as in the FA defined above. Δ is the *output alphabet* and $\lambda : Q \times \Sigma \rightarrow \Delta$ is a mapping from $Q \times \Sigma$ to Δ. That is, the output given by λ depends on the input and the current state. A Moore machine is also defined as a six-tuple $M = (Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where all is as in the Mealy machine, except that λ is defined as $\lambda : Q \rightarrow \Delta$. That is, the output given by λ only depends on the current state. Fig.2.1(a) and Fig.2.1(b) are graphical representations of an example of a Mealy machine and an example of a Moore machine, respectively.



a) A Mealy machine model b) A Moore machine model
Fig.2.1 Finite state machine models

The system modelled by a FSM changes from state to state along predefined state transitions as they execute. In FSMs, state transitions are normally considered to be instantaneous. FSMs can be used to describe a process or a complete system at an abstract level. However, viewing a real-time system as a finite state machine is sometimes difficult because the timing constraints do not fit well into the model. FSMs force sequential thinking; concurrent computations are not expressed naturally by an FSM [Chandrasekharan *et al* 85, Ostroff 89]. Also FSMs do not provide decomposition of complexity [Zave 82] (i.e. there is no structuring mechanism in FSMs for grouping sets of related states).

Because of above restrictions, one must also consider a machine that also provides some facility supporting timing constraints and concurrent processing. State machines have been extended in order to model concurrent real-time systems, typical examples are Extended State Machines (ESMs) [Ostroff 89], Communication Real-Time State Machines (CRSMs) [Shaw 92] and Statecharts [Harel 87].

ESM's [Ostroff 89] are finite state machines enhanced with the standard programming notions of data variables, guarded events, concurrency and communication. ESM's provide communication, for synchronised cooperation, as in CSP [Hoare 85] via communication operations (message passing) over channels. For instance, the send operation $c!\alpha$ (or $c!m$) in ESM M_1 which means "send the command α (or message m) over channel c to some other ESM, say M_2", and the corresponding receive operation in M_2 is $c?\alpha$ (or $c?m$) which means "receive the command α (or message m) from M_1 over channel c". Each transition in an ESM is defined as a guarded operation (transformation function) which has lower and upper time bounds which define the time-span within which a transition is allowed to fire. These time bounds are used for calculating the best and worst case execution times by examining the maximum cumulative delays of trajectories which are paths through ESM. When the guard (an enabling condition) evaluates to *true*, then the transition occurs within the duration of a time bound causing an instantaneous change of state determined by the operation.

CRSM [Shaw 92] is a relatively new notation for specifying concurrent real-time systems including the monitored and controlled physical environment. Similar to ESM's, CRSM's are essentially state machines that communicate synchronously over unique unidirectional channels in a manner much like the CSP [Hoare 85]. Compared to ESM's, CRSM's differ in the way in which transitions are defined and time is handled. Each transition in a CRSM is described by a guarded command: <guard> $\rightarrow$ <command>, where the <guard> is a Boolean expression over the local variables of machine M and the <command> can be either an IO, an *input* or *output*, (send or receive a message), or an *internal* command (a computation or a physical activity such as opening a gate or moving a robot arm). A transition can only be executed (fired) if its guard is evaluated to *true*. The execution time for an internal command c is given by a best-case/worst-case pair $[t_{min}(c), t_{max}(c)]$, indicating that the duration d of c is somewhere in the interval $0 \leq t_{min}(c) \leq d \leq t_{max}(c)$. These bounds can be interpreted as a requirement (i.e., all implementation must obey these constraints) or as a given behaviour (i.e., the given c obeys these constraints). IO times are also represented by pairs, only in this case the pair denotes the earliest and latest times that the IO can occur after entering a given state. The intersection of a sender's and a receiver's intervals gives the time of possible communications. For example, suppose

i) machine M_1 enters state U at time t_U;
ii) U has a transition $g_1 \rightarrow E(x)?$ with time pair $[a_1, a_2]$ where a_1, a_2 are times,
iii) machine M_2 enters state V at time t_V;
iv) V has a transition $g_2 \rightarrow E(x)!$ with time pair $[b_1, b_2]$ where b_1, b_2 are times;

then, if g_1 and g_2 evaluate true, the earliest possible time at which IO will occur is:

$$t = \max(t_U + a_1, t_V + b_1) \tag{2.2}$$

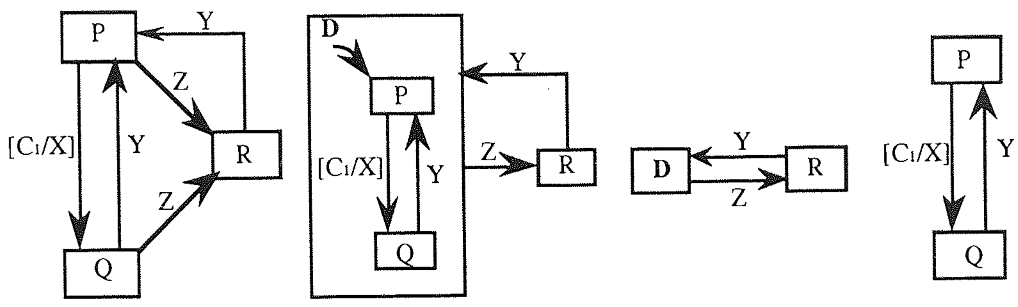
2.2.2 Statecharts

Statecharts [Harel 87] provide an abstraction mechanism based on FSM's. Statecharts denote composition of state machines into super-machines which may execute concurrently. The state machines contain transitions which are marked by enabling conditions and events. It is assumed that events are instantaneous, and a global discrete clock is used to trigger sets of concurrent events. Statecharts are hierarchical, and may be composed into complex charts. Transitions may be linked between charts by chart construction operators. Statecharts is a specification language rather than a programming language, and emphasises execution models and external view rather than internal structure.

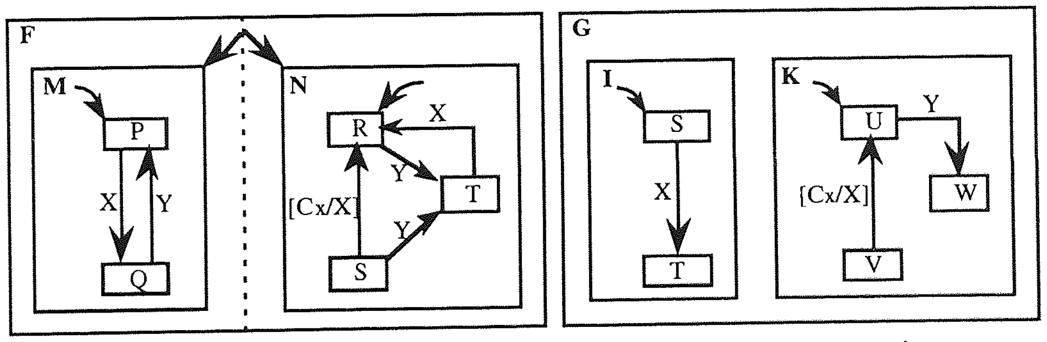
Given a Statechart one may perform a number of operations, including: '*refinement*' (decomposing a state into more lower-level states), '*clustering*' (making a higher level state from more other states), '*OR*' constructs (connecting two charts in a disjunction), '*AND*' constructs (connecting two charts in a conjunction). Each transition in a Statechart is labelled with a pair (C/E) where C is an enabling condition (a Boolean expression on event) and E is an event which triggers the enabled transition.

Let us consider a Statechart example, as described by the graph shown in Fig.2.2(a). The system in this example comprises three states (P, Q and R) and three events (X, Y and Z) that trigger transitions between the states. Note that event X causes state Q only if condition C_1 holds. In this case the condition acts as a guard rather than as a transition trigger. Fig.2.2(b) describes how the Statechart of the Fig.2.2(a) is clustered. In this figure states P and Q interact with state R and can be clustered as a new state **D** which is generated as a higher-level state of P and Q. Since event Z causes both P and Q to change to R, we can denote it by a single transition Z from the higher level state **D** to R. The single transition refers to two possible transitions, each from one of the "substates". Another issue that Fig.2.2(b) emphasises is the default entry point, shown by the small unlabelled arc, which is to state P in **D**. Thus, any transition directed to **D** will obey the default entry and will cause a default transition to P. Therefore event Y has the same effect as in Fig.2.2(a).

The main objective of Statecharts is to solve the problems that characterise FSM's. Two of the severe problems in traditional FSM's are the lack of support for modularity and the exponential state explosion. These problems are elegantly solved by the Statechart's visual concept of *zooming*. Fig.2.2(c) and Fig.2.2(d) demonstrate zooming in both directions based on the example shown in Fig.2.2(b). We can zoom-out and consider only higher-level states as in Fig.2.2(c) or zoom-in and consider only events and substates in the clustered state as shown in Fig.2.2(d). Finally, Fig.2.2(e) illustrates conjunction in which a state **F** is formed by an AND constructor between charts **M** and **N**, and Fig.2.2(f) illustrates disjunction in which a state **G** is formed by an OR constructor between charts **I** and **K**.



(a) Statecharts (b) Clustering (c) Zooming-out (d) Zooming-in



(e) AND construction (f) OR construction

Fig.2.2 Examples of Statecharts

A number of studies have examined theoretical issues concerning Statecharts and have identified problems and ambiguities, such as the transition which may enable itself [Scholefield 90, der Beeck 94]. To avoid or solve this kind of problem the semantics of Statecharts have had to have been altered [der Beeck 94]. Currently, there are over 20 variants of Statecharts and their scope and limitations are summarised in [der Beeck 94].

2.2.3 Grafcet and Sequential Function Chart (SFC)

To describe increasingly complex discrete real-time process-control systems, a commission entitled *Normalisation of the representation of the Specifications of Logic Controllers* was set up in France in 1975. This group developed a discrete mathematical model, Grafcet, for modelling complex control systems and particularly manufacturing systems. Grafcet is defined as a discrete mathematical model for control systems which describes the function to be performed only, i.e. it defines a sequential machine in the mathematical sense, free from all technology and all implementation [David and Alla 92].

Grafcet model is inspired by the Petri net model (see this Chapter) with a few differences [David 91]. Although Grafcet is defined with a view to PLC systems, no field of application is explicitly excluded because Grafcet provided a general process oriented way of describing control functions. Grafcet was adopted by the IEC as an international standard in 1988: "Sequential Function Chart" [IEC848 88]. The Grafcet standard has now been accepted by many countries [David 91] and it is widely taught and used in industry, particularly in France.

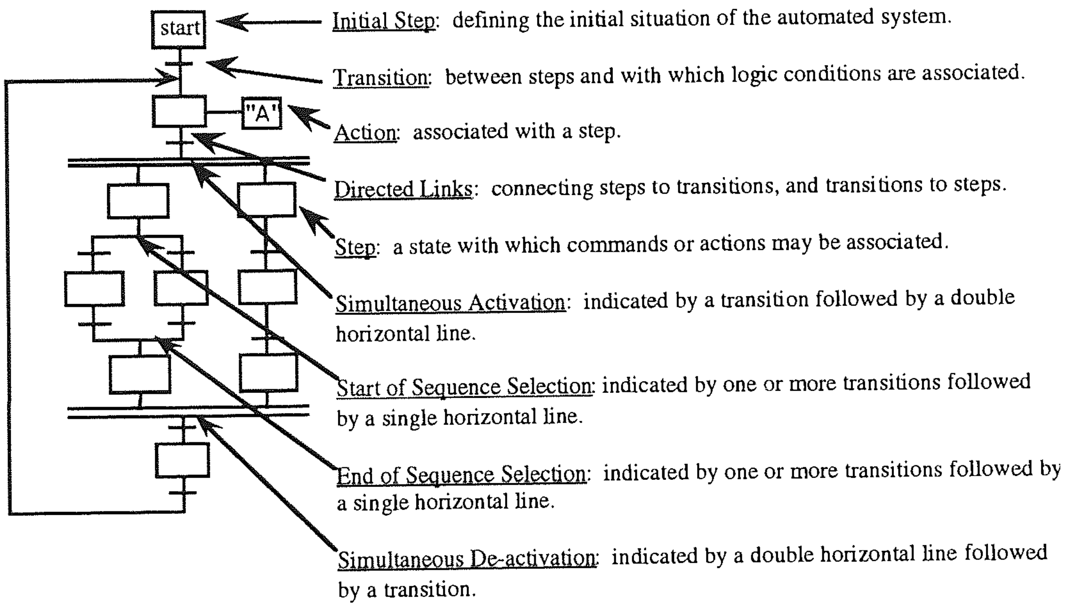

start
"A"
Initial Step: defining the initial situation of the automated system.
Transition: between steps and with which logic conditions are associated.
Action: associated with a step.
Directed Links: connecting steps to transitions, and transitions to steps.
Step: a state with which commands or actions may be associated.
Simultaneous Activation: indicated by a transition followed by a double horizontal line.
Start of Sequence Selection: indicated by one or more transitions followed by a single horizontal line.
End of Sequence Selection: indicated by one or more transitions followed by a single horizontal line.
Simultaneous De-activation: indicated by a double horizontal line followed by a transition.

Fig.2.3 A Grafcet model

The basic concepts of Grafcet are "*step*", "*transition*", "*action*", and "*receptivity*". In Grafcet a system is described as a sequence of interconnected *steps* and *transitions*. Each step represents a partial state of the system which can be associated with an action. Graphically a step is represented by a named box and its associated action is represented by another box which is (horizontally) connected with step. An action represents a processing operation which may generate an output to the environment and is performed when the step is active. At a given instant a step is either *active* or *inactive*. Each step has two external attributes; a *step flag* and an *elapsed time*. The step flag is used to represent the state of a step. The elapsed time represents the duration during which the current step has been active. Each transition acts as a guard passing "control" from one or more predecessor steps to one or more successor steps. A transition is either enabled or disabled. A transition is said to be enabled when all the immediate preceding steps are active, otherwise it is said to be disabled. After a transition is enabled; its firing is governed by the receptivity associated with the transition. Receptivity is a Boolean function of the Grafcet variables or the step's states. If the receptivity associated with an enabled transition is true, then the transition should fire immediately, otherwise the enabled transition must wait until the receptivity becomes true. The entities in a Grafcet model and the graphical notation used to describe the model are shown in Fig.2.3.

There is no discipline enforced on the construction of Grafcet as in Petri nets and unstructured flowcharts. The only structural restriction in Grafcet is that two steps are always separated by a transition and two transitions are always separated by a step. A hierarchical decomposition capability, called *macro*, is provided which may be adopted by the user to impose a design discipline. A macro is a Grafcet having only one input step and one output step, and is similar to a block in a structured programming language.

Grafcet not only represents the *static* structure of a control system, but also allows one to determine its *dynamic* behaviours as well. By introducing evolution rules similar to the firing rule of Petri nets, a Grafcet representation can reflect the evolution of a control system. The Grafcet model evolves as steps complete their processing operations and the successor transition fires, activating the next steps. The following are the evolution rules defined for Grafcet:

1. The *initial state* of a Grafcet model is characterised by its initial active *steps*.
2. All firable transitions are immediately fired.
3. The firing of a transition causes the *deactivation* of all the immediately preceding steps connected to the corresponding transition symbol by directed links and is followed by the *activation* of all the immediately following steps.
4. If several transitions can be fired simultaneously then all shall be fired simultaneously.
5. When a step is simultaneously activated and deactivated, it remains active.

SFC (also widely called Grafcet in industry) is a graphical language defined in the international standard IEC1131 Part 3 [IEC 93]. Although SFC inherits from Grafcet [David and Alla 92], there are a few differences between these two models. Some "peculiarities" of Grafcet, which represent a major divergence from Petri nets such as the simultaneously firing rule for conflicting situation, are absent in SFC. Also some programming concepts are included in SFC; this makes SFC more programming oriented rather than description oriented. That is, SFC is defined more from the consideration of the "interior system" rather than the "exterior"

SFC has the same syntax as Grafcet which partitions the control system into a set of steps and transitions interconnected by directed links. Associated with each step is a (set of) actions and associated with each transition is a transition condition. An action can be a command (an output statement) or a piece of program and a condition is a no side-effect logical expression, which are expressed by the programming languages such as ST defined in IEC1131 (see Chapter 1). The evolution rules defined for SFC are the same as that defined for Grafcet, except for the conflicting situation (i.e., a step is shared by more than one transition). In SFC, if a step is shared by more than one transition then only one condition associated with the shared transitions can become true at any instant, otherwise it is treated as an error [IEC 93]. Specifically the similarities and differences between SFC and Grafcet can be stated as:

Similarities:

i) SFC and Grafcet follow the same discipline for structural construction; in addition a set of basic well-formed constructs are defined in SFC.
ii) The evolution rules defined for SFC are the same as that defined for Grafcet, except for the conflicting problem.
iii) Both SFC and Grafcet cannot model the non-determinism.
iv) Implicit communication between transition and step (see Chapter 3) is allowed in both models.

Differences:

i) SFC is more programming oriented than Grafcet because it provides a good interfaces with programming languages via its action and condition descriptions.
ii) Simultaneous firing of transitions in conflicting situation is not allowed in SFC; instead it is treated as an error if this situation occurs.
iii) The receptivity concept of Grafcet is absent in SFC definition; instead transitions in SFC can be associated with a much wider condition [Mallaband 91].
iv) Action duration is not negligible (i.e. instantaneous) in SFC.

Although SFC is defined more from the consideration of the "interior system" (i.e., from implementation point of view) rather than the "exterior" (i.e., from description point of view), SFC is not a complete programming language because in PLC applications SFC has to be used in conjunction with other low level programming language such as ST (Structured Text) and LD (Ladder Diagrams) [IEC 93]. The reason is that there is no notation in SFC to be used to describe the conditions or actions.

2.2.4 Petri Nets

2.2.4.1 Basic Petri Nets

The theory of Petri nets was initiated by Carl A. Petri in 1962. The aim was to develop a comprehensive mathematical theory for modelling communication, synchronisation, choice and concurrency. Since then, a great deal of research has been done on Petri net theory [Peterson 81, Reisig 85, Murata 89, Rozenberg 88, 89, 90, 91, 92]. The chief attraction of Petri net theory is the way in which the basic aspects of concurrent systems are identified both conceptually and mathematically. As pointed out by Rozenberg and Thiagarajan [87], the emphatic separation of non-determinism (i.e. a non-deterministic choice in a conflicting situation) and concurrency (i.e. two independent executing processes) at a fundamental level has had a deep influence in the subsequent development of the theory such as the trace theory [Mazurkiewicz 87, 88] and the partial order semantics [Castellano *et al* 87, Reisig 88a, Reisig 88b, Vogler 91].

Algebraically, a Petri net is defined as a 5-tuple [Murata 89],

$$PN = (P, T, F, W, M_0) \tag{2.3}$$

such that:

$P = \{p_1, p_2, ..., p_n\}$ is a finite set of places;
$T = \{t_1, t_2, ..., t_n\}$ is a finite set of transitions;
$F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation);
W: $F \rightarrow \{1, 2, 3, ...\}$ is a weight function;
M_0: $P \rightarrow \{0, 1, 2, ...\}$ is an initial marking (or state);
$P \cap T = \varnothing$ and $P \cup T \neq \varnothing$.

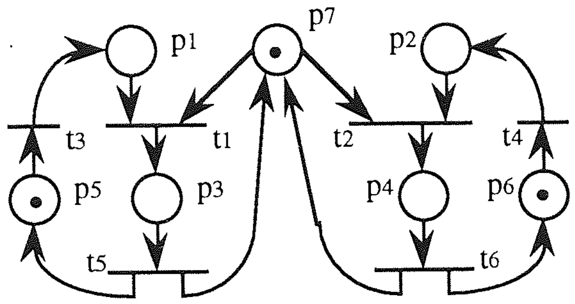
A Petri net structure $N = (P, T, F, W)$ without any specific initial marking is denoted by N. A Petri net with the given initial marking is denoted by (N, M_0).

A graph structure is often used to illustrate a Petri net where a circle represents a place and a bar represents a transition. The state of a net is represented by its marking in which tokens are assigned to places; this is shown graphically by placing dots in the places. The graphical model describes the static properties of a system. The dynamic aspect or the behaviour of Petri nets is denoted by changes in the markings which occur during *execution* of a Petri net. The execution of a Petri net is determined by exercising *enabling* and *firing* rules [Murata 89]:

i) A transition $t \in T$ is *enabled* if and only if each input place $p \in P$ of t is marked with at least $w(p,t)$ tokens, where $w(p,t)$ is the weight of the arc from p to t.

ii) an enabled transition may or may not fire (depending on whether or not the event actually takes place).

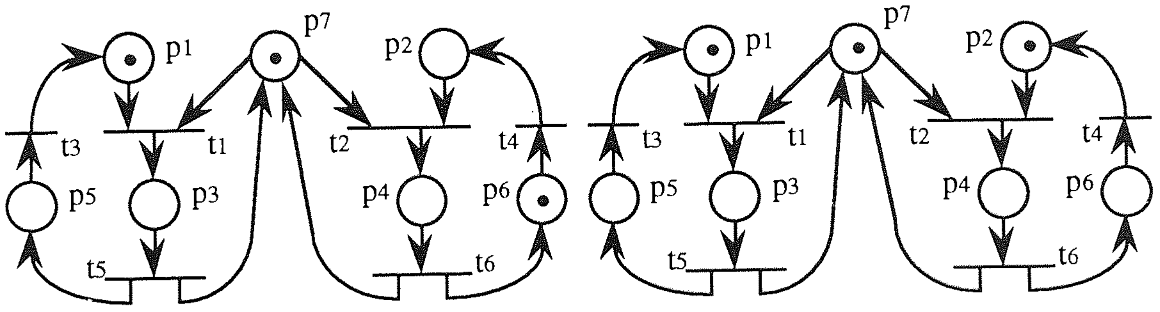
iii) A firing of an enabled transition t removes $w(p,t)$ tokens from each input place p of t, and adds $w(t, p)$ tokens to each output place p of t, where $w(p,t)$ and $w(t, p)$ are the weights of the arcs from p to t and t to p, respectively.



A Petri net with Initial marking

$P = \{p1, p2, p3, p4, p5, p6, p7\}$
$T = \{t1, t2, t3, t4, t5, t6\}$
$F = \{<p1, t1>, <p7, t1>, <p2, t2>,$
$<p7, t2>, <p5, t3>, <p6, t4>,$
$<p3, t5>, <p4, t6>,$
$<t1, p3>, <t2, p4>, <t3, p1>,$
$<t4, p2>, <t5, p5>, <t5, p7>,$
$<t6, p6>, <t6, p7>\}$
$M_0 = \{p5, p6, p7\}$

A Petri net definition



Petri nets with the next markings

Fig.2.4 A Petri net showing concurrency and non-determinism

A Petri net is said to be *ordinary* if all of its arcs weights are 1's. All Petri nets considered in this thesis are ordinary. For the above rule of transition enabling, it is assumed that each place can accommodate an unlimited number of tokens. Such a Petri net is referred to as an *infinite capacity net* [Murata 89]. For modelling many physical systems, it is natural to consider an upper limit to the number of tokens that each place can hold. Such a Petri net is referred to as a *finite capacity net* [Murata 89]. For a finite capacity net (N, M_0), each place p has an associated capacity $K(p)$, the maximum number of tokens that p can hold at any time. For finite capacity nets, for a transition t to be enabled, there is an additional condition that the number of tokens in each output place p of t cannot exceed its capacity $K(p)$ after firing t. A Petri net is called a *condition/event* (C/E) net if the capacity $K(p)$ for each place p is restricted to $\{0, 1\}$. A Petri net modelling concurrency (t_3 and t_4) and non-determinism (between t_1 and t_2) after transitions t_3 and t_4 have fired is shown in Fig.2.4.

Reachability

Reachability is a fundamental method of studying the dynamic properties of any system modelled by Petri nets. The firing of an enabled transition will change the token distribution (marking) in a net according to the firing rules defined for Petri nets. A sequence of firings will result in a sequence of markings. A marking M_n is said to be *reachable* from a marking M_0 if there exists a sequence of firings that transforms M_0 to M_n. A firing or occurrence sequence is denoted by $\delta = M_0\, t_1\, M_1\, t_2 \ldots\ldots t_n\, M_n$ or simply $\delta = t_1 t_2 \ldots t_n$. In this case, M_n is reachable from M_0 by δ and we write $M_0[\delta > M_n$. The set of all possible markings reachable from M_0 in a net (N, M_0) is denoted by $R(N, M_0)$ or simply $R(M_0)$. The set of all possible firing sequences from M_0 in a net (N, M_0) is denoted by $L(N, M_0)$ or simply $L(M_0)$.

The *reachability problem* for Petri nets is the problem of finding if $M_n \in R(M_0)$ for a given marking M_n of a net (N, M_0). For general Petri nets, the reachability, although decidable, has been shown to be exponential space-hard [Reutenauer 88]. However the equality problem is undecidable [Reutenauer 88], i.e., there is no algorithm for determining if $L(N, M_0) = L(N', M'_0)$ for any two Petri nets (N, M_0) and (N', M'_0).

Boundedness

A Petri net (N, M_0) is said to be *k-bounded* or simply bounded if the number of tokens in each place does not exceed a finite number k for any marking reachable from M_0, i.e., $M(p) \leq k$ for every place p and every marking $M \in R(M_0)$. A Petri net (N, M_0) is said to be *safe* if it is *1*-bounded. For example, the net shown in Fig.2.4 is safe. When places in a Petri net represent buffers, a bounded net, say k, will guarantee that there will be no overflows in the buffers if the buffer size > k, no matter what firing sequence is taken.

Liveness

A Petri net (N, M_0) is said to be *live* (or equivalently M_0 is said to be a *live marking* for N) if, no matter what marking has been reached from M_0, it is possible to ultimately fire *any* further transition of the net by progressing through some further firing sequence. The concept of liveness is closely related to the absence of deadlocks. If a Petri net (N, M_0) is live, then this means that deadlock-free is guaranteed, no matter what firing sequence is chosen.

In Petri net theory, following symbols are often used for a pre-set and a post-set:

$\bullet t = \{p \mid (p, t) \in F\}$ = the set of input places of transition t.
$t \bullet = \{p \mid (t, p) \in F\}$ = the set of output places of transition t.
$\bullet p = \{t \mid (t, p) \in F\}$ = the set of input transitions of place p.
$p \bullet = \{t \mid (p, t) \in F\}$ = the set of output transitions of place p.

The following is a theorem, called *token invariance property*, of Petri nets which has been considered as one of the powerful proof techniques associated with nets [Rozenberg and Thiagarajan 87, Murata 89, Manna and Pnueli 92, Reisig 92].

Theorem:
Given a Petri net $PN = (N, M_0)$, S is a subnet of PN. If S satisfies following requirements:

i) $\forall t \in S, |\bullet t \cap S| = |t\bullet \cap S|$

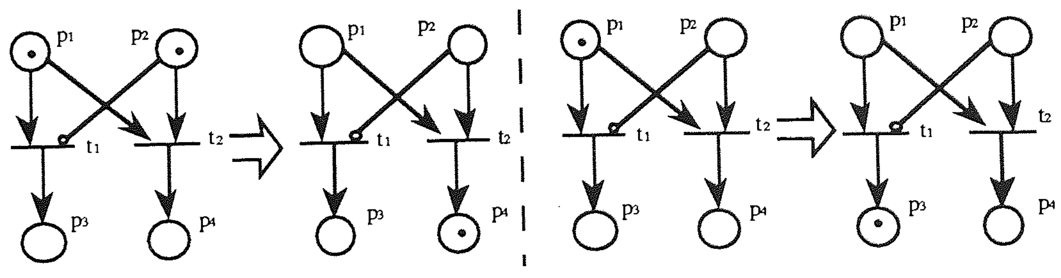
ii) $\forall p \in S, (\bullet p \cup p\bullet) \subseteq S$

then S has the property that the number of tokens in S remains invariant.

Proof:
Requirement i) means that for each transition t in S the number of its predecessor places in S equals the number of its successor places in S. That is, every transition t in S removes and places exactly the same number of tokens in S when t is fired. Requirement ii) means that for each place p in S all transitions removing and placing tokens from p are in S. That is, no transition outside S can deposit tokens in S or remove tokens from S. This guarantees that the number of tokens in S at any stage of the execution of net equals the number of tokens in S at the initial marking. □

2.2.4.2 Inhibitor Arc Petri Nets

The introduction of the 'inhibitor' arc [Peterson 81] produced a net which is often simpler than an equivalent (non -inhibitor arc) C/E net. An inhibitor arc from place p_i to a transition t_j has a small circle rather than an arrowhead at the transition. The firing rule is changed as follows: a transition is enabled when tokens are in all of its (normal) input places and zero tokens are in all of its inhibitor input places. The transition fires by removing tokens from all of its (normal) inputs. A Petri net with inhibitor arc and its pre- and post-firing markings are illustrated in Fig.2.5(a) and Fig.2.5(b).



(a) t_1 is disabled (b) t_1 is enabled
Fig.2.5 A Petri net with inhibitor arc and its pre- and post- markings

Although Petri nets have the desirable features of modelling non-determinism and concurrency as well as graphical notation, Petri nets have been criticised in following aspects: i) the lack of compositionality (nets need to be completed before any properties are proven); ii) the difficulty of automating analysis techniques in the case of infinite state reachability graph; iii) lack of a high-level specification language; iv) lack of a satisfactory verification system.

Some of these problems of Petri nets have been addressed already by research workers. To model large-scale systems and keep the states manageable, various high level Petri nets have been proposed; these include Predicate/Transition nets (PrT) by Genrich and Lautenbach [81] and coloured Petri nets by Jensen [81a]. Additionally, Petri nets have been combined with temporal logic [Suzuki and Lu 89, He and Lee 90, Sagoo and Holding 90, 91] (see section 2.2.7) in order to provide a formal specification language and a proof system.

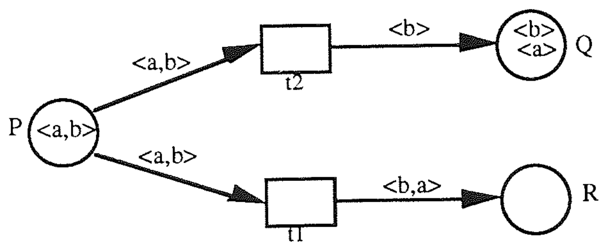
2.2.4.3 High Level Petri Nets

For the needs of this thesis, only the brief concepts of high-level Petri nets are described from descriptive aspect. More detailed discussion of these nets is beyond the scope of this thesis and can be found in [Genrich and Lautenbach 81, Jensen 81a].

For a given condition/event (C/E) net, $PN = (P, T, F, M_0)$, P can be viewed as a set of propositional variables with changing truth values. A marking of the net indicates the current truth values of the propositional variables. The occurrence of an event causes, in an obvious way, changes in the truth values of the proportional variables in its neighbourhood [Rozenberg and Thiagarajan 87]. PrT nets were developed by Genrich and Lautenbach [81] who raised these ideas to the level of first order logic. That is, PrT nets can be considered as a structurally folded version of conventional Petri net [Murata 89]. The unique feature of PrT nets is that allows large complex systems to be modelled.

In the structure of a PrT net, a *predicate* of fixed arity is associated with each place and a domain of tuples (of the same arity) of individuals with each place to provide the interpretation. A *token pattern* is associated with each arc which specifies the extent of change (in the corresponding predicate extensions) caused by firing of the transition. A marking will indicate the current extension of each predicate in terms of the set of tuples of individuals that currently satisfy the predicate.

A transition t is said to be enabled in a PrT net if the token patterns on its input arcs are satisfied by the tuples associated with its input places and the token patterns on its output arcs are not satisfied by the tuples associated with its output places. The firing of a transition will change the current extensions of the predicates in its neighbourhood by removing the satisfied tuples from its input places and adding the tuples specified on its output arcs to its output places. Here is a simple example given in [Rozenberg and Thiagarajan 87].



$D_P = D_R = \{<a,b>, <b,a>, <a,c>, <c,a>\}$

$D_Q = \{<a>, <b>, <c>\}$

Fig.2.6 An example of Predicate/Transition nets

P and R are binary predicates and Q is a unary predicate. Let D_P, D_Q, and D_R denote the domains (of interpretations) of P, Q, and R respectively. At the marking shown the current extension of P is {<a,b>}. In other words, just <a,b> satisfies P whereas, for instance, <a,c> does not. Similar remarks apply to Q and R. For the transition t_1 to fire at a marking *M*, P<a,b> must be true and R<b, a> must be false at *M*; after the firing of t_1, whenever it fires, P<a,b> is no longer true and R<b,a> becomes true. In other words, the extent of change caused by firing t_1 consists of removing the tuple <a,b> from P and adding the tuple <b,a> to R. This is expressed through token patterns on the arcs surrounding transition t_1. Thus at the marking shown on the PrT net, t_1 is enabled and t_2 is not enabled because Q<b> is currently true.

From the example it can be seen that PrT nets are very powerful in a descriptive aspect and can describe a system as a first order system. However, PrT nets have a serious drawback in an analysis aspect. Although the invariant analysis techniques developed by Lautenbach [87] can be adopted to PrT nets, the analysis becomes mathematically complicated and needs much simplification and improvements to be practicable [Genrich 87]. This complication primarily arises because the elements of the incidence matrix are no longer integers (as in conventional Petri nets) but logical expressions.

Coloured Petri nets (CP nets) were developed from PrT nets by Jensen [81a], but have a more refined invariant calculus [Jensen 81b]. The difference between CP nets and PrT nets is that, in CP nets, each token has attached a data value called the *token colour* which can be investigated and modified by firing transitions; each place has a *colour set* which specifies the kind of tokens which may reside on the place; each transition has a *guard* (Boolean expression containing some of CP net variables); and each arc has an *arc expression* (containing some of the CP net variables). A transition is said enabled *iff*:

- there are enough tokens of the correct colours on each input place;
- the guard expression evaluates to true.

When a transition is enabled, it may fire. A transition may fire concurrently with other transitions if there are enough tokens that each transition can get its "own share". After firing:

- a multi set of tokens is removed from each input place;
- a multi set of tokens is added to each output place.

Compared to PrT nets, CP nets are powerful in an analysis aspect. This is because the invariant analysis technique has been extended to CP nets [Jensen 81b]. Although the invariant technique is powerful, in many cases it is applicable only to special subclasses of Petri nets or special situations. This is also true for CP nets.

The concept of time is not explicitly given in the definition of Petri nets. In the Petri net model, transitions are fired in a nondeterministic way and do not relate in any way to specific times. However, sometimes, it is necessary and useful to introduce time delays associated with the transitions and/or places in a net in order to do some time-related analysis, such as performance evaluation or solving a scheduling problem of a system. To extend Petri net model to real-time systems, various modifications have been applied to the basic model.

2.2.5 Time-Related Petri Nets

To model time the basic Petri net model must be enhanced. There exist several proposals for extending standard Petri nets including time. Ranchandani [74] proposed associating delays with transitions. The proposed model is often called a "timed Petri net". This approach has been used also by Ramamoorthy and Ho [80] and Zuberek [85]. Merlin and Farber [76] proposed using two values, Min and Max times, to define a range of delays for each transition. This model is often called a "time Petri net". Sifakis [80] proposed instead associating the delays with places, this model is often called a "timed place transition net" (TPTN). Coolahan and Roussopoulos [83] employed an approach similar to Sifakis. Associating delays with places does not increase the power of the model (e.g. TPTN and timed Petri nets are equivalent [Sifakis 80]), but does retain the instantaneous firing feature of the basic Petri net model. Razouk [84] has proposed using enabling times along with firing times. Both timed and time Petri nets have been proved very useful for expressing many temporal and timing constraints. Although they are different from each other and not equivalent (a timed Petri net can be modelled by using a time Petri net, but the converse is not true [Berthomieu and Diaz 91]), they are complementary to each other. Compared to time Petri nets, timed Petri nets are useful in hard real-time systems where most timing delays are fixed, this also simplifies system performance analysis. Time Petri nets are more general than timed Petri nets and can be used to model ranges of delays which are sometimes difficult to express using only fixed firing durations. The following are the basic descriptions for each proposed model:

1) Merlin and Faber [76] associate with each transition two time units, *a* and *b*, where $a \leq b$; *a* is the minimum delay before a transition can fire starting from the time at which the transition is enabled; *b* is the maximum delay before a transition must fire. The enabling rule is the same as in Petri nets.

2) Ranchandani [73], Ramamoorthy & Ho [80] associate a finite firing duration with each transition of Petri nets. The enabling rule is the same as in Petri nets but the firing rule is modified as: a) firing a transition will take a fixed amount of time, and during this period cf time the token is reserved; b) a transition can be fired when it is enabled.

3) Zuberek [85], similar to Ranchandani [73], associates a finite firing duration with each transition of a Petri net. The only difference is that a transition must start to fire at the moment it is enabled.

4) Sifakis [80] associates the time with each place. The advantage of this approach is that it preserves the convention of transitions as instantaneous events. A token is either *unavailable* (from the moment the token is deposited into a place until the time associated with place elapses) or *available* (after the time associated with the place has elapsed). Only available tokens are considered for enabling conditions.

5) Razouk [84] associates two times with each transition, one is the *enabling time* and the other is *firing time*. Each transition must remain enabled for the enabling time. After the enabling time, a transition becomes firable and it must begin firing at that instant of time (unless disabled by the firing of a conflicting transition). During the firing, the tokens are absorbed by the transition and do not appear on the output places until the transition finishes its firing.

2.2.6 Temporal Logic

Temporal logic is an extension to propositional and predicate calculus, with new operators being introduced in order to express properties relating to time. Temporal logic provides a formal system for qualitatively describing and reasoning about the occurrence of events in time, and in fact, *even* for the occurrence of infinitely many events [Emerson and

Srinivasan 88]. In any system of temporal logic, various temporal operators are provided to specify how the *truth* of the properties of a temporal system vary over time. Typical temporal operators such as $\Box$ (henceforce), $\Diamond$ (eventually), $\bigcirc$ (next), $\mathcal{U}$ (until) [Manna and Pnueli 88] permit expression of such important properties of temporal systems as *invariances* (assertions that describe properties that are always true of a temporal system), *eventualities* (assertions that specify a property that must become true at some future instant of time), and *precedences* (assertions that state that one event must occur before another).

In the past temporal logic has been found useful for the specification of real-time and concurrent systems [Pnueli 86, Manna and Pnueli 88, Emerson and Srinivasan 88, Ostroff 89, Manna and Pnueli 92]. After using temporal logic as a language to describe the system specification, the properties of systems can be formally verified, in which a formula consisting of temporal logic operators is interpreted over a *model* which is a computational structure of states (e.g., a sequence or tree of states) defined to be the semantics of a program [Manna and Pnueli 88].

Three types of semantics which can be assigned to programs are identified by Manna and Pnueli in [88] as:

- Linear Semantics. In this semantics a program is the set of all computations, where each computation is a (possible infinite) sequence of states generated by performing the basic actions of the program one at a time. Concurrent activity of two parallel processes in the program is represented by the *interleaving* of their atomic actions. This simpler (or perhaps less natural) representational model leads the following:

i) program $a(b + c)$ is considered to be (semantically) equivalent to the program $ab + ac$ in linear semantics, even though the first program decides between taking b or c only after performing a, while the second program decides a prior whether it is going to perform ab or ac.

ii) program $ab + ba$ is considered to be semantically equivalent to $a \| b$ (where "+" stands for alternative and "||" stands for parallel), even though in the first program there is a non-deterministic choice in the program for the single process, while in the second program each of the processes is deterministic.

That is, the linear approach is unable to distinguish between the non-determinism caused by a non-deterministic choice existing in the program, and the one being introduced by the interleaved representation of concurrency [Manna and Pnueli 88].

To compensate for the limitations of the simple representation of concurrency by interleaving, the notion *fairness* is added to the interleaving computational model. One of the main functions of the fairness requirements is to exclude the interleaving between two processes in which, beyond a certain point all the actions taken are taken from only one of the processes, while the other has some enabled actions which never get executed [Manna and Pnueli 88].

- Branching Semantics. In this semantics the semantics of a program is a single (possibly infinite) tree of states, where each node representing a state s in the computation has as direct descendants all the states that can be derived from s. This semantics certainly distinguishes between the program $a(b + c)$ and the program $ab + ac$, which differ in the point at which the choice between the two possibilities is made. On the other hand, this approach still represents concurrency by interleaving, and considers the programs $a \| b$ and $ab + ba$ equivalent.

- Partial Order Semantics. In partial order semantics the semantics of a program is a (possibly infinite) structure of states (or events), on which two basic relations are recognised. One is a partial order which represents the *precedence* ordering between events, and constrains certain events to occur *following* some other events. The other relation is the *conflict*. Two events are considered to be conflicting if they can never participate in the same execution. The conflict relation is extended by the precedence relation. If a is in conflict with c, and a precedes b and c precedes d, then it follows that each of $\{a, b\}$ is in conflict with each of $\{c, d\}$. If two events are neither related by the precedence relation, nor are in conflict, we say that they are independent, which can be interpreted as being concurrent, i.e. can be executed in parallel. An execution is any maximal substructure which does not contain two conflicting actions. Partial order semantics is the only one which identifies concurrency as a unique phenomena which is not translatable to any interleaved representation [Manna and Pnueli 88, Katz and Peled 90]. Thus, partial order semantics distinguish between non-determinism (caused by non-deterministic choice) and concurrency.

This variety of semantics leads to different temporal logics: Linear Temporal Logic (LTL) [Pnueli 86, Manna and Pnueli 88, 92], Branching Temporal Logic (BTL) [Clarke *et al* 86, Emerson and Srinivasan 88], and Temporal Logic over Partial Order Semantics [Reisig88a, 88b, Katz and Peled 90]. LTL is interpreted on a set of linear executions; formulas in this logic are defined over sequences. BTL is interpreted on a branching structure that can be viewed as a tree; formulas in this logic are defined over such trees. Temporal Logic over Partial Order is interpreted on structures of partial order; formulas in this logic are defined over independent executions.

Although there is a consensus among many theoreticians and practitioners that temporal logic constitutes a promising approach to the problem of designing correct parallel real-time systems, there is not yet a consensus regarding which specific temporal logic is best suited for this purpose [Emerson and Srinvasan 88]. LTL will be considered in this thesis because of its simpler structure and the well developed proof system, compared to BTL and temporal logic based on partial order.

A temporal formula is constructed from *state formulas* to which temporal operators, Boolean connectives, and qualification are applied. A state formula is a first-order predicate which does not contain any temporal operators. A state formula is evaluated at a single state in a sequence and expresses properties of a single state and a temporal formula is evaluated over a sequence of states.

The computational model of LTL consists of a possibly infinite sequence of states

$$\sigma = s_0,\ s_1,\ \ldots \qquad (2.4)$$

where each state s_i provides an interpretation for the temporal formula. For a state sequence σ, let the sequence σ^k be the *k*-shifted sequence given by:

$$\sigma^k = s_k,\ s_{k+1}\ \ldots \qquad (2.5)$$

then the semantics of temporal operators

$$\Box\ (\textit{always}),\quad \Diamond\ (\textit{eventually}),\quad \bigcirc\ (\textit{next}),\quad \mathcal{U}\ (\textit{until}) \qquad (2.6)$$

are defined as follows:

(i) if φ is a classical formula (which is constructed from propositions or predicates and logical operators such as NOT ($\neg$), AND ($\wedge$), OR ($\vee$) and implication ($\rightarrow$)) containing no temporal operators, then the notion of φ holding at a state s is defined by

$\sigma \models \varphi \quad$ iff $\quad s_0 \models \varphi \qquad (2.7)$

φ can be interpreted over a single state in σ, which is the initial state s_0,

(ii) temporal operator *always* ($\Box$) is defined by

$\sigma \models \Box\varphi \quad$ iff $\quad \sigma^k \models \varphi$ for all $k \geq 0$, $\qquad (2.8)$

$\Box\varphi$ holds on σ iff φ is holding at any state s in σ,

(iii) temporal operator *eventually* ($\Diamond$) is defined by

$\sigma \models \Diamond\varphi \quad$ iff $\quad \sigma^k \models \varphi \quad$ for some $k \geq 0$, $\qquad (2.9)$

$\Diamond\varphi$ holds on σ iff φ is holding at some state s_k in σ,

(iv) temporal operator *next* ($\bigcirc$) is defined by

$\sigma \models \bigcirc\varphi$ iff $\sigma^1 \models \varphi$ (2.10)

$\bigcirc\varphi$ holds on σ iff φ is holding at state s_1 in σ,

(v) temporal operator *until* ($\mathcal{U}$) is defined by

$\sigma \models \psi\mathcal{U}\varphi$ iff there exists a $k \geq 0$, such that $\sigma^k \models \varphi$,

and for every i, $0 \leq i < k$, $\sigma^i \models \psi$ (2.11)

$\psi\mathcal{U}\varphi$ holds on σ iff ψ holds continuously until sometime φ holds.

2.2.7 Temporal Petri Nets

Both Petri nets and temporal logic have been widely used in real-time and concurrent systems. Petri nets are appropriate to model the system causal behaviour explicitly, while temporal logic is appropriate to specify the system properties and constraints. For example, certain important properties of real-time and concurrent systems such as the *eventuality* (certain transitions must eventually fire; certain places must eventually have a token), *fairness* (if a transition becomes firable infinitely often, then it must fire infinite often), and constraints (two places must be mutual exclusive), cannot be described explicitly by Petri nets, but they can be described explicitly by temporal logic. In fact, Petri nets lack a generally accepted high level specification language [Ostroff and Wonham 90]. Since one formalism can complement the other, a combination of two formalisms is a highly desirable approach if they can be combined together consistently. By combining both together, one can model a real-time concurrent system operationally by Petri nets and specify the system properties declaratively by temporal logic. This work has been addressed by some researchers in recent years [Suzuki and Lu 89, He and Lee 90, Uchihira and Honiden 90, Sagoo and Holding 91]. The difference between them is how to combine a Petri net and temporal logic. Suzuki and Lu [89] defined a new class of Petri nets, called temporal Petri nets, in which general Petri nets are combined with the propositional LTL. Formally, a temporal Petri net is defined as the pair:

$$\text{TN} = (PN, f) \tag{2.12}$$

where PN is a Petri net which is defined by Equation (2.3) and f represents a set of temporal formulas that describe the temporal behaviour of PN. The formulas f are described in a language which is based on LTL of Manna and Pnueli [88]; in the temporal Petri net defined by Suzuki and Lu this language was referred to as L_{PN}. The syntax for (propositional) temporal formulas of L_{PN} over PN defined in [Suzuki and Lu 89] can be summarised simply as follows:

i) atomic propositions: (*p has a token*), (*t is firable*), (*t fires*), where $p \in P$ and $t \in T$.

ii) If $g, h \in L_{PN}$, then $g \wedge h$, $g \vee h$, $\neg g$, $g \rightarrow h$, $\bigcirc g$, $\Box g$, $\Diamond g$, and $g\,\mathcal{U}\,h \in L_{PN}$

The formal semantics of L_{PN} are given as follows. Let α be a possible infinite firing sequence from a marking M. For each i, $0 \leq i \leq |\alpha|$, let β_i and γ_i be the sequences such that $|\beta_i| = i$ and $\alpha = \beta_i\gamma_i$. That is, β_i is the prefix of α with length i, and γ_i is the postfix of α excluding β_i. Let M_i be the marking such that $M_i\ [\beta_i> M$. For a formula g the validity of g under pair $<M, \alpha>$, written $<M, \alpha> \models g$, is defined as:

$<M, \alpha> \models (p\ has\ a\ token)$ *iff* p has at least one token at M (2.13)

$<M, \alpha> \models (t\ is\ firable)$ *iff* t is firable at M (2.14)

$<M, \alpha> \models (t\ fires)$ *iff* $\alpha \neq \lambda$ (λ represents an empty sequence) and $t = \beta_1$ (2.15)

$<M, \alpha> \models g \wedge h$ *iff* $<M, \alpha> \models g$ and $<M, \alpha> \models h$ (2.16)

$<M, \alpha> \models \neg g$ *iff* not $<M, \alpha> \models g$ (2.17)

$g \vee h$ and $g \rightarrow h$ are shorthands for $\neg(\neg g \wedge \neg h)$ and $\neg g \vee h$ respectively

$<M, \alpha> \models \bigcirc g$ *iff* $\alpha \neq \lambda$ and $<M, \gamma_1> \models g$ (2.18)

$<M, \alpha> \models \Box g$ *iff* $<M_i, \gamma_i> \models g$ for every $0 \leq i \leq |\alpha|$ (2.19)

$<M, \alpha> \models \Diamond g$ *iff* $<M_i, \gamma_i> \models g$ for some $0 \leq i \leq |\alpha|$ (2.20)

$<M, \alpha> \models g\, \mathcal{U}\, h$ *iff*

($<M_i, \gamma_i> \models g$ for every $0 \leq i \leq |\alpha|$) or

(for some $0 \leq i \leq |\alpha|$, $<M_i, \gamma_i> \models h$ and $<M_j, \gamma_j> \models g$ for every $0 \leq j < i$) (2.21)

Specially, the places and transitions of a temporal formula in temporal Petri nets can be described in the following notation:

$p \equiv (p\ has\ a\ token)$

$t\ ok \equiv (t\ is\ firable)$

$t \equiv (t\ fires)$

$\neg p \equiv (p\ has\ no\ token)$

$t\ \neg ok \equiv (t\ is\ not\ firable)$

The work of Suzuki and Lu [89] defines a temporal Petri net (which is given by the pair: TN = (PN, f)) such that f is interpreted as a restriction on the firing sequences generated from PN; thus only those firing sequences that satisfy f are allowed to occur.

He and Lee [90] integrated the Predicate/Transition (PrT) nets with first order temporal logic. Only predicates (e.g., the set of places in PrT nets) and individual tokens are considered in the translated temporal logic system. The temporal logic system, L, translated from a PrT net, consists of system independent temporal logic axioms and inference rules, such as the axioms and inference rules developed by Manna and Pnueli [88], and system dependent axioms and inference rules which are derived from the initial marking and firing rules. The system independent temporal logic axioms and inference rules are independent of any specific PrT system. The system dependent axioms and inference rules differ from one PrT net to another. For example, the system dependent axiom and inference rule for the example of a PrT net shown in Fig.2.7 are:

System dependent axiom: $p_1(a) \wedge p_1(c)$ (2.22)

System dependent inference rule:

$p_1(x) \wedge p_1(y) \wedge \neg p_2(y) \wedge (x < y) \Rightarrow \bigcirc(p_2(y) \wedge \neg p_1(x) \wedge \neg\, p_1(y) \wedge (x < y))$ (2.23)

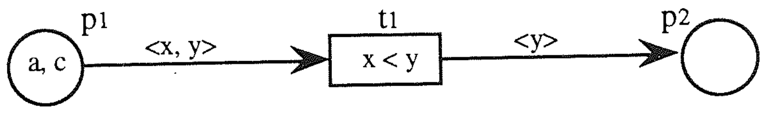


Fig.2.7 A fragment of a Predicate/Transition net

An algorithm was developed for translating PrT nets into temporal logic systems and a *refutation* proof technique was proposed to assist in the verification of both safety (invariance) and liveness (eventuality) properties [He and Lee 90]. A refutation proof of a formula p in a logical system L is a syntactical derivation of a sequence of formulas F_0, F_1, ... F_n such that $F_0 = \neg p$, F_n = false; and F_{i+1} is derived from F_i by one of the inference rules of the system L.

The algorithm developed by He and Lee has been extended to condition/event (C/E) nets by Sagoo and Holding [91]. The extended algorithm is applied to C/E nets in the following manner: given a C/E net with initial marking $M_0 = \{p_1, p_2, \ldots\ldots p_k\}$, the algorithm is used to express M_0 as a formula of propositional logic, i.e. $p_1 \wedge p_2 \wedge \ldots\ldots \wedge p_k$ and it is referred to as the system dependent axiom. Each transition in the net is converted into a system dependent inference rule that defines the firing of the transition in terms of pre-and post-conditions. For instance an inference rule for transition t has the form $U \Rightarrow \bigcirc U'$, where U contains a formula comprising the conjunction of all the input places in $\bullet t$ and the conjunction of the negation of all the places in $(t\bullet - \bullet t)$, and U′ contains a formula which is symmetric to U. For example, $p_1 \wedge p_2 \wedge \neg p_3 \Rightarrow \bigcirc \neg p_1 \wedge \neg p_2 \wedge p_3$ is a system dependent inference rule derived from transition t in the fragment of a C/E net shown in Fig.2.8.

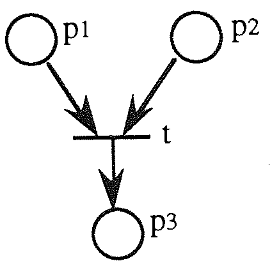


Fig.2.8 A fragment of a Condition/Event net

Similar work has also been reported by Queille and Sifakis [82] in which BTL instead of LTL is considered as the specification language and the atomic proposition in BTL logic corresponds to a place. BTL is potentially more powerful than LTL since it can express all the properties LTL can, and more [Emerson and Srinivasan 88] (e.g, it is able to distinguish the nondeterminism caused by nondeterministic choice). However it has a high time complexity for its decision procedures (i.e., an algorithm that determines whether a given formula is valid or not) because of the expressiveness of its basic modalities [Emerson and Srinivasan 88].

One of common motivations of the work above is to formally specify and verify Petri net properties with temporal logic. However, the automatic verification problem was not discussed in these work. To automatically verify the Petri net properties with temporal logic, the combined temporal Petri nets must be decidable. The decidability problem is whether or not there exists a legal firing transition sequence satisfying a given temporal logic formula on a given Petri net.

A different combination of general Petri nets and linear time propositional temporal logic is proposed by Uchihira and Honiden [90], in which the atomic proposition in temporal logic corresponds to transition firing only rather than to tokenised places and enabled transitions as in [Suzuki and Lu 89, He and Lee 90, Sagoo and Holding 91]. Based on such a correspondence, Uchihira and Honiden show that their combined temporal Petri nets are decidable if a given Petri net *PN* is modified into *PN′* by adding some places, transitions, and arcs to *PN* such that *PN′* is deadlock-free. However, some property about places cannot be verified. This problem arises because the correspondence between transitions in Petri nets and atomic propositions in temporal logic does not consider places and thus lacks the information necessary to distinguish between different states of a system, as modelled by a Petri net [He and Lee 90]. Such a correspondence does not seem appropriate for analysing real-time process-control systems modelled by Petri nets because both events (transitions) and states (places), especially states, are important to describe the system behaviour.

2.2.8 Petri Nets in Real-Time Systems

In the past years Petri nets, with their various extensions, have been proved to be valuable for modelling, analysing, and controlling real-time process-control systems which exhibit concurrency, synchronisation and co-ordination among their subsystems. For example, Tyrrell and Holding [86] proposed a method for error detection and recovery in distributed systems, in which Petri net model was used to identify formally both the state and the state reachability tree of a distributed system. Leveson and Stolzy [87] proposed a technique for using time Petri net without generating the entire Petri net reachability graph for analysing the safety properties of real-time systems. The idea is to work backwards from high-risk states to determine if these hazardous states are reachable. Holloway and Krogh [90] proposed a method for solving forbidden state control problems for a class of controlled discrete event systems based on an extended Petri nets called controlled Petri nets (CPN), in which binary control inputs can be applied to exogenous conditions for enabling transitions in the net. The significance of CPN's is that the control logic is separated from the Petri net model of the controlled system by the introduction of external control places. Cofrancesco *et al* [91a] proposed an approach to the design of software for real-time control in which the Petri net is used to describe the control structure and the net is then translated into executable programs. The use of Petri net formalisms as a framework to describe a control system is reported by Cofrancesco *et al* [91b]. They also describe the development of a software workbench for control system design. Le Bail *et al* [91] proposed a hybrid Petri net model in which both discrete and continuous behaviours of systems can be modelled. The hybrid Petri net model overcomes the deficiency of conventional time and timed Petri nets which can only model discrete event systems. An excellent survey about Petri nets in manufacturing systems is reported by Silva and Valette in [89].

The application of the temporal Petri net techniques in real-time process-control systems has been shown by Sagoo and Holding [90, 91], and application in Flexible Manufacturing Systems (FMS) is described by Zurawsky and Dillon [91].

With specific references to logic controller design, the use of Petri nets has also increased. A comparative study between PLC and Petri nets was presented by Silva and Velilla [82]. Various implementation for programmable logic controllers (PLCs) based on Petri nets are discussed including details of the Petri net definition languages, the internal representation (data structures) of Petri nets, the interpretation formalism (synchronous versus. nonsynchronous). Similar work was also reported by Courvoiser *et al* [83]. The approach to combining Petri net based PLC and artificial intelligence is investigated by Atabkhche *et al* [86] and Sahraoui *et al* [87].

A similar approach is adopted by Murata *et at* [86] who proposed an enhanced safe Petri net model, called control-net or C-net for short, for describing industrial sequence control specification. The C-net is a Petri net extended with process I/O functions and process status functions. These extended functions are used to define process interfaces and process status. A C-net is very similar to the Grafcet except that the communication in a C-net is explicit while the communication in Grafcet can be implicit (see Chapter 3).

Zhou and DiCesare [89] presented a methodology for the design of Petri net controllers; this accommodates error recovery while preserving desirable properties of the system. Wilson and Krogh [90] proposed a rule-based approach to Petri net controller design for discrete event processes. Ferrarini and Maffezzoni [91] proposed a software development environment for logic controller design based on Petri nets. Petri net analysis techniques using p-invariants and t-invariants [Lautenbach 87] are included into the environment as tools to prove the correctness of the controller (instead of allowing only graphical editing and simulation). Barker and Song [92] defined an extended Petri net called the programmable logic controller net (PLCNet) and presented synthesis rules for their PLC, and discussed the design of simulation tool for PLCs based on PLCNet.

The major characteristics of Petri nets that make them suitable for real-time process-control systems are:

- Petri nets support explicit representation of causal dependencies and independencies.
- Petri nets can be used to model a system at different levels of abstraction, without changing the language used in the modelling. That is, Petri nets provide a family of tools for use in many phases of the design process including system modelling, qualitative verification, performance evaluation and implementation.
- The Petri net provides a natural means to describe formally the parallelism and synchronisation in engineering environments.
- Petri nets make the modelling of real-time process-control systems easier because of
  a) the correspondence between the concepts of Petri nets and the concepts of systems such as events, activities, state change;
  b) the relatively straightforward graphical representation;
  c) the explicit modelling of states and events.
- Petri nets provide the ability to check the system for undesirable properties such as deadlock and boundedness, and to analyse system performance, and to generate supervisor control code directly.

2.2.9 Comparison Between Petri Nets and Grafcet

2.2.9.1 Similarities Between Petri Nets and Grafcet

Two early work works on Petri nets are particularly interest from the point of view of the concepts included in Grafcet. Azema *et al* [76] discussed Petri nets as a tool for describing, designing and verifying the concurrent systems. Their model consisted of three parts: a control graph, a data graph and an interpretation. The control graph is based upon an interpreted Petri net in which each transition is associated with a predicate and an action. The nodes of the data graph are of different types and represent operators, memory cells, predicate cells, input/output registers, and input/output lines. Transition cannot be fired if its associated predicate is "false". If it is "true" then the transition is fired if enabled. The action is executed each time the transition is fired. Predicates are Boolean conditions and actions are elementary modifications or transformation functions. As the authors explained, in order to avoid the association of both the inputs (Boolean conditions) and the outputs (actions) with transitions it is possible to associate the action with the places during implementation or simulation. In fact, this idea is very similar to that used later in the interpreted Petri nets by Queille and Sifakis [82], the controlled Petri nets proposed by Holloway and Krogh [90], and the principles adopted by Grafcet. Also the verification of safeness and liveness of Petri nets based on the reachability tree was discussed in [Azema *et al* 76]. The unique feature of this work is that the control and data graphs were separated at the abstract level and were closely combined together by the interpretation of the net structure.

A similar idea is also discussed by Velilla and Silva [88] in which the application software of a real-time control system is decomposed into two co-operating parts: a Control Part (CP) and an Operative Part (OP). The control part (CP) is modelled by means of Petri nets which include synchronisation, concurrency, and control logic of the application software. The operative part (OP) is composed of a set of sequential procedure associated with the transitions of the control part net model plus data structures. The sequential procedures implement predicates and/or actions (data transformation). If a transition is enabled and the predicate is evaluated to be true, then the transition is fired. The firing of a transition allows the execution of the associated actions (local PID regulators, etc.). Such a CP-OP decomposition and the Petri net based implementation provide a safe and flexible design for application software of real-time control systems. The important thing is that the separation between the data processing and control synchronisation logic is seen again in this work.

Since Grafcet has been adopted by the IEC as international standards [IEC848 88, IEC 93], many PLC manufacturers and software producers have chosen it as an input language for control and proposed implementations on computers or controllers. As the level of industrial interest and use is growing very fast, many researchers begin to study the Grafcet either from a theoretical or a practical point of view. Much of their work has been done in France. In 1992, a congress of Grafcet (Grafcet 92/Function Charts for Control Systems, Theory and Applications, Paris, 25/26, March 1992) was held at Paris and later a special issue for Grafcet was published by *Automatic Control Production Systems* in 1993. The special issue involves many aspects from theory to practice including semantics, times, validation, verification, and performance evolution.

Although Grafcet has been adopted widely by industry, it is becoming well known that Grafcet has some problems regarding to its semantics concerning timing, evolution and execution [Frachet and Colombari 93, Marce and Le Parc 92]. The problem concerns a lack of precision and completeness which leads to ambiguity such that the specifier and user of a Grafcet model may have different understanding of its behaviour and equally valid interpretations. To tackle the problem of semantical ambiguity in Grafcet, the semantics of Grafcet was formally defined in [Marce and Le Parc 92] by using a formal synchronous language SIGNAL [Le Guernic *et al* 91]; and consequently the tools developed for SIGNAL can be used to verify Grafcet. In [Frachet and Colombari 93] the semantics of time in Grafcet is investigated. Based on a mathematical framework called Non-Standard Analysis, the authors propose a time model for modelling dynamic systems and then apply this time model to the interpretation of timing aspects of a Grafcet model. Two time scales are introduced, *external* and *internal*. The external time scale is used to characterise the "evolution speed" of the stimuli that are external to the system under modelling. The internal time scale is used to characterise the execution of algorithm associated with a step in Grafcet. The principle of the interpretation proposed by the authors is based on following two hypotheses:

i) There is a strict simultaneous between the input and the output. That is, the "causality delay" from input to output is "non-exist" in the external time scale.

ii) An event has a sole time in the external time scale. At any instant, one and only one of the events can occur.

The authors state that the non-simultaneous event hypothesis does not come from the consideration of the "interior system" being very fast with respect to the exterior; it is a property of the exterior itself, running from its own coherence. Regarding the implementation the authors [Frachet and Colombari 93] suggest the use of a processor in which the sample period (in the sense of its sample of the occurrence of event) is "ideally small" with regard to the time interval separating the successive event occurrences. A similar implementation idea has also been mentioned by Benroniste and Berry [91] in their synchronous model of real-time system. Also this notion has been stated as a hypothesis by Marce and Le Parc in [92]. However, in [Andre and Peraldi 93] the hypothesis ii) is not considered as a fundamental one, because the authors believe that the non-simultaneous event is a matter of interpretation which is not directly relevant to the model itself.

In [Aygaline and Denat 93] the validation (qualitative and quantitative) of the functional Grafcet model and its performance evaluation based on the framework of Petri nets are proposed. Qualitative validation is performed using Autonomous Petri nets (i.e., untimed Petri nets in which firing times are either unknown or not indicated [David and Alla 92]). By associating time delay with each place, the analytical performance evaluation in the normal working condition and quantitative validation can be performed based on the obtained TPTN [Sifakis 80]. Their Petri net model of Grafcet is obtained based on the following transformation rules:

Rule1:

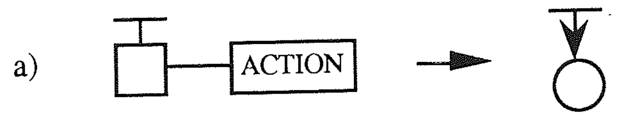


Fig.2.9 Transformation from Step-to-Place

Rule2:

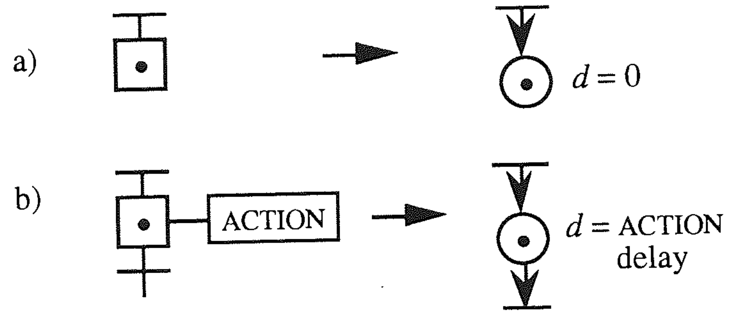


Fig.2.10 Net marking transformation

That is, places are used to model the steps and the token unavailable time defined in TPTN is used to model the delay of the action associated with the step. Two constraints are imposed by the authors in this work:

i) the system must be represented by a single Grafcet.
ii) the Grafcet model must not have reactivative steps.

Here 'reactivative' means that a step will be activated while it is still active. The approach adopted is similar to the parallel independent research reported in this thesis. The authors argue that constraint i) "is a little restrict because it is always possible to establish a single Grafcet" in which all the activity flags are made graphical explicit. In a communication from the authors [Aygaline and Denat 94] they illustrate their method by proposing that for the two co-operating Grafcets with implicit communications (i.e., a transition depends on a step's state but there is no explicit graphical connection between them) shown in Fig.2.11 it is possible to establish a single Grafcet with explicit communications shown in Fig.2.12 (where $\uparrow a$ means an event and X_1 means the active step flag of step1, transition T_2 depends on external event $\uparrow b$ and active state of step1, further discussion see Chapter 3).

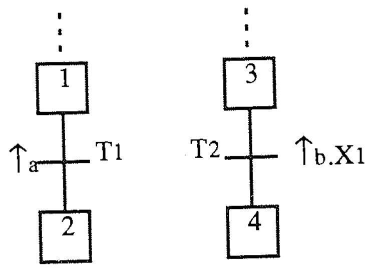


Fig.2.11 Grafcet's co-operation by implicit communication

However, detailed examination of this representation reveals several problems. To illustrate the problem, consider the original Grafcet (Fig.2.11), it can be elaborated by the addition of an action to step1. Also, without losing generality, let us assume that, in Fig.2.11 event ↑x is associated with the input transition of step1, event ↑y is associated with the input transition of step3, and the action associated with step1 performs "z := z +1" and z is initially zero as shown in Fig.2.13.

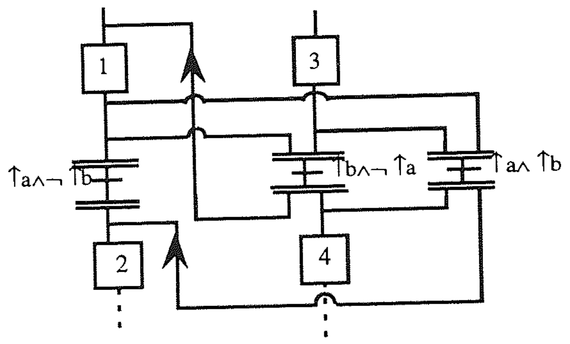


Fig.2.12 Single Grafcet representation of Fig.2.11

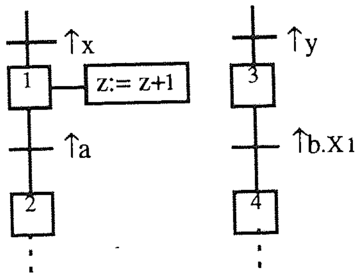


Fig.2.13 Elaboration of Grafcet of co-operating processes (Fig.2.11)

Now, consider the following possible input sequences of events and the behaviours of both Grafcets:

1) Suppose the input sequence is "↑x, ↑y, ↑b, ↑a". According to the semantics of Grafcet, the value of variable z will be 1 in Fig.2.13 when step2 is active. However, in the Aygaline and Denat's single Grafcet shown in Fig.2.12, z will have value 2 when step2 is active. Note that Fig.2.13 has no reactivation but Fig.2.12 does for the given input sequence.

2) Suppose the event $\uparrow a = \uparrow b$ and the input sequence is "$\uparrow x$, $\uparrow a$". For this case, the step2 will become active after event $\uparrow a$ occurs in Fig.2.13, but in the single Grafcet (Fig.2.12) it is not because $\uparrow a \bullet \neg \uparrow b = \uparrow b \bullet \neg \uparrow a$ = FALSE, and the transition with which $\uparrow a \bullet \uparrow b$ is associated is not enabled since step 3 is not active.

It follows that, in certain circumstance, it is not possible to represent the two co-operated Grafcets (Fig.2.11) by a single one in which all the implicit communication are made explicit using the method of Aygaline and Denat.

The reason is that the implicit communication using a step's active flag not only describes the dependency between a step in one Grafcet and a transitions in another Grafcet but also, (and perhaps more importantly), implicitly expresses a dynamic *priority* (i.e., a master-slave relationship) between two tasks modelled by two Grafcets. This notion is explained in more detail in Chapter 3. In general, in order to represent the same sequence of states in both Grafcet and Petri net models, Grafcet must not include reactivating situation because steps in Grafcet work as "flip-flops" and places in Petri nets work as "counters" [Silva and Valette 89].

Since Petri nets provide a powerful design notation, their use in Grafcet-based systems is not restricted to the modelling and analysis of Grafcet structures. For example, the control of real-time process systems which, due to complexity, are normally decomposed into a hierarchy of abstraction level such as planning, scheduling, co-ordination of sub-systems, and local control of each sub-system [Silva and Valette 89]. The last two levels of control, co-ordination and local control, are investigated by Sayat and Ladet [93] using Petri nets (for co-ordination) and Grafcet (for local control). They define an extended Petri net called *co-ordination Petri net* in which a task is represented by a place named PT_i (i is the task number) whose input and output share the same transition (i.e. they form a self-loop). Each such transition T_j is associated with a pair $\{A_j, E_j\}$, where A_j is the impulse action of launching the task and E_j is the occurrence of the termination of the task (see graph shown in Fig.2.14).

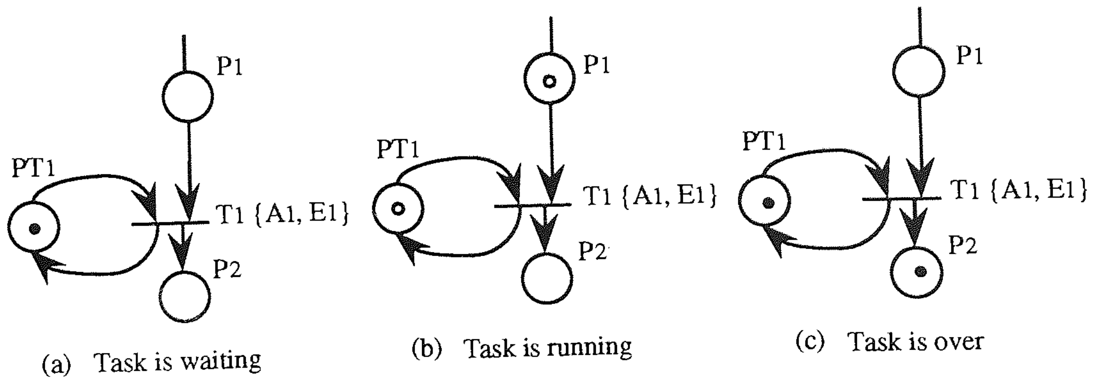


Fig.2.14 Modelling a task by a co-ordination Petri net

The state of the task is the state of the token located in place PT_i. If the task is being executed the token is reserved (represented by a small circle in PT_i) and does not participate in the enabling of the transition (see 2.14(b)). If the token is not reserved, then the task is waiting. The execution of the task is launched at the moment when the transition is enabled, and the end of task is indicated by the occurrence of the termination of the task E_j. There are distinct similarities between co-ordinated Petri nets and Grafcet when local control is taken into account. The authors present in co-ordination Petri net structures for typical tasks such as sequential tasks, the synchronisation of two sequences of tasks, and mutual exclusion between two tasks. To provide a more generic description model, high level Petri nets [Jensen 81a] are considered.

In co-ordinated Petri nets the execution of a task is launched at the moment when the transition is enabled. This leads to problems when considering non-deterministic choice since it is not clear from the semantics of co-ordination Petri nets how only one task can be launched following a non-deterministic choice.

2.2.9.2 Differences Between Petri Nets and Grafcet

The first important difference between Petri nets and Grafcet lies in the simultaneous firing rule defined in Grafcet as shown in Fig.2.15 and Fig.2.16. Petri nets can model not only concurrency but also non-determinism. However, compared to Petri nets, Grafcet lacks of the ability to model the non-determinism because of its simultaneous firing rule.

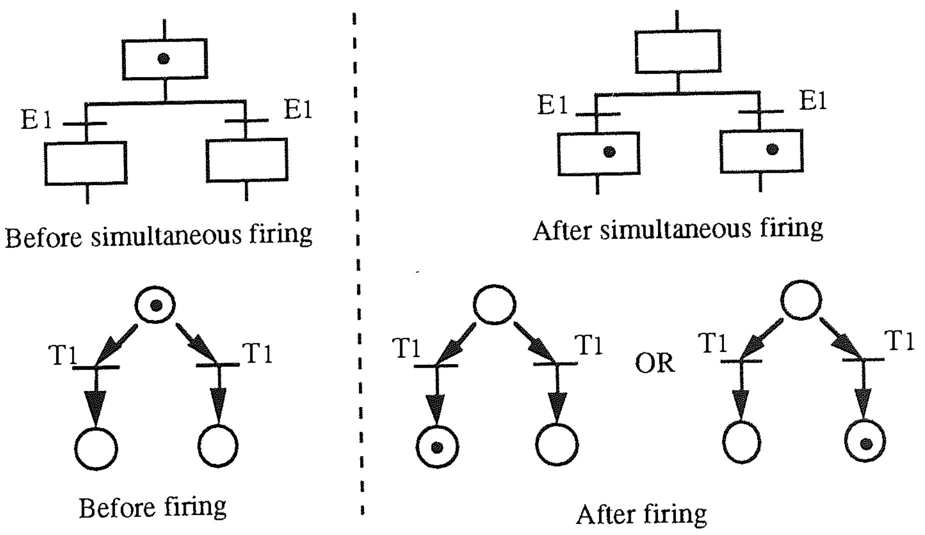


Fig.2.15 Difference-1 caused by simultaneous firing

The second important difference concerns reactivation. In Petri nets, a place holds the tokens and makes them visible and a transition transports the tokens and changes the tokens distribution in marking. For Petri nets which are not 1-bounded, a place works like a *counter*. Compared to Petri nets, a step in Grafcet works like a *flip flop* as shown in Fig.2.17. This difference means that Grafcet cannot benefit from the powerful analysis technique *P-invariant* [Lautenbach 87] developed for Petri nets, even though Grafcet was inspired from Petri nets [Silva and Valette 89]. That is why Aygaline and Denat [93] have to make assumptions about the reactivation when they use a Petri net to model and analyse Grafcet.

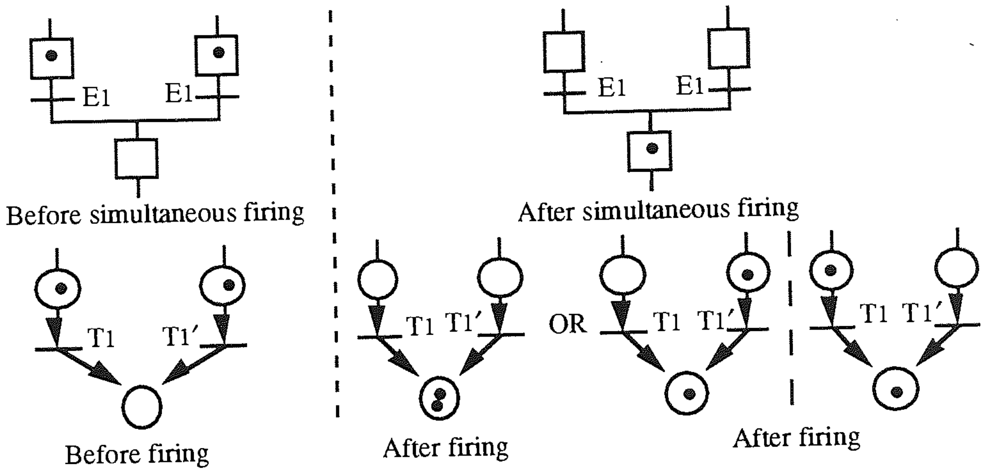


Fig.2.16 Difference-2 caused by simultaneous firing

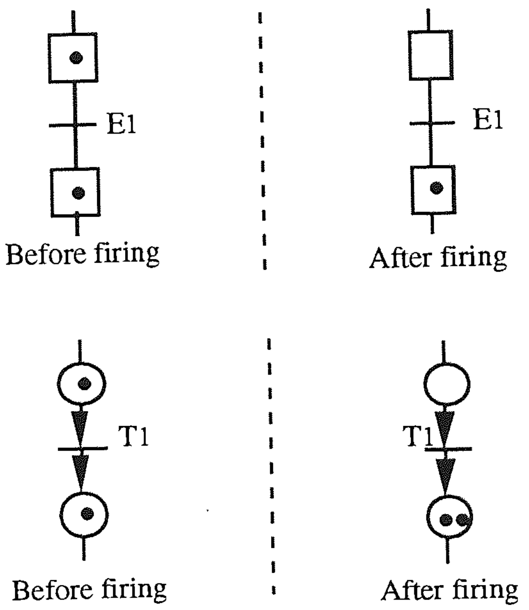


Fig.2.17 Difference-3 caused by reactivating firing

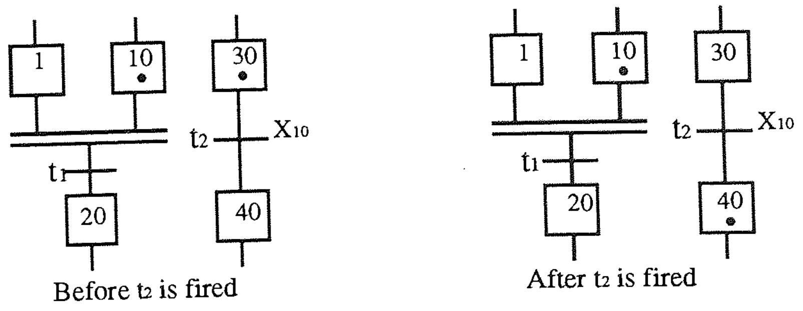


Fig.2.18 Implicit communication in Grafcet

A third important difference between Petri nets and Grafcet is that in Grafcet communication between processes may be implicit (see Fig.2.18) while in Petri nets all communications are explicit. More details of implicit dependency about Grafcet will be discussed in Chapter 3.

Although Grafcet is different from Petri nets, there are significant similarities between these two models. As stated by David and Alla [92], if Grafcet does not include the differences mentioned above, then it can be treated as an "interpreted" Petri net.

2.3 Design Methodology

2.3.1 Introduction

A method is a guide to using a formal language or technique in the development of systems, it may be defined in terms of heuristics or a collection of specific techniques. The problem of complexity in real-time system development is often highlighted by the fact that formal language developers rarely provide any method with which to utilise their language [Scholefield 90]. Strutt [89] in a survey of formal 'methods' (of general applicability) asserts that this lack of methods is one of the prime reasons for hostility towards formal techniques (quoted from [Scholefield 90]):

"Most methods are not methods at all, but merely formal languages or notations...... The provision of a method (perhaps by integrating existing structured methods with formal languages) is essential."

For example, the emphasis of FSM's was originally on the notation rather than the method. One of well known software requirement methods for real-time system development, based on FSM's, is the SREM (Software Requirement Engineering Methodology) [Alford 77, 85]. In SREM, each process is defined as a set of graph models called R-nets which define paths from input to output interfaces. Each R-net has a single entry, which may be an interface to the environment, and one or more exits, which are either interfaces to the environment or terminators. R-nets are composed of subnets, which are similar to R-nets and can be further decomposed. The limitation of SREM is that nondeterminism cannot be modelled within SREM. Another popular method, Statecharts [Harel 87], provides an abstraction mechanism based on FSM's. Statecharts not only solve the concurrency and modularity problems of FSMs, but also offer extensive prescriptive guidelines. Although Petri nets and Grafcet have been applied to the developments of real-time process-control systems, few of the applications support a design method. Most applications support only drawing, simulation and analysis of Petri nets and Grafcet because of the lack of a design method. However, recent works have appeared which propose design methods for the development of real-time process-control using Petri nets; these include a rule-based approaches [Wilson and Krogh 90, Etessami and Hura 91] and a hybrid approach (i.e. combination of top-down and bottom-up approaches) [Zhou and DiCesare 89, 93].

In the hybrid approach proposed by Zhou and DiCesare, a real-world system is modelled at the highest level of abstraction as a Petri net with single place and a single transition. The top-down design is accomplished by gradually replacing the place or transition with more complicated subnets. Each successive step contains increasing detail and guarantees that desirable system properties of the Petri nets such as freedom from deadlock are preserved. Top-down design of Petri nets is also called as "*stepwise refinement*" [Valette 79, Suzuki and Murata 83]. Since refinements are often local and are required to have no side-effects, interaction among subnets is very difficult to handle using such a strategy. Therefore, bottom-up design is also proposed in [Zhou and DiCesare 89, 93]. The bottom-up design problem is to design correct interactions among the existing sub-nets (which are themselves obtained by the top-down design). Both top-down and bottom-up designs are discussed for the generic resources-sharing control problem in [Zhou and DiCesare 89, 93].

Rule-based methods aim to describe the functional requirements of the real-world system by a set of rules. The rules are then mapped onto Petri nets and the system properties of the Petri net are analysed using Petri net theory. If the analysis shows that desired properties hold in the Petri net model, then they can be inferred in the set of rules.

In fact, the rule-based formalism can be considered as the basis of all the system modelling techniques involving the Petri net model, because each transition in a Petri net can be considered as a transition rule with precondition(s) and postcondition(s). The similarities between the rule and net descriptions, and between the token player of the Petri net and the inference engine of rule-based system, have been investigated in [Atabakhche *et al* 86, Sahraoui *et al* 87].

A rule-based approach can be considered as a bridge to fill the gap between the informal world and the formal world and the 'human experts generally find it easy to express methods for solving problems in their application areas by using a rule formulation' [Hayes-Roth 85].

The reason that rule-based schemes have been selected and used in real-time process-control systems is the proven convenience of the notation, and the close relationships between rule-based scheme and Petri nets [Atabakhche *et al* 86, Sahraoui *et al* 87, Willson and Krogh 90, Etessami and Hura 91], and the great success of Petri nets for representing concurrency as a graphical language [Rozenberg and Thiagarajan 87].

2.3.2 Rule-Based Development

The production rule formalism has become a popular method for knowledge representation in expert systems [Waterman 86]. However, in the last decade, rule-based approaches have been used also by researchers in real-time process-control systems [Komoda *et al* 84, Tashiro *et al* 85, Pathak and Krogh 89, Wilson and Krogh 90, Etessami and Hura 91], Programmable Logic Controller (PLC) designs [Barker *et al* 89, Barker and Song 92], and information system development and maintenance [Assche *et al* 88, Loucopoulos and Champion 88, Loucopoulos and Layzell 89, Poo and Layzell 90].

To provide highly flexible and maintainable control software for discrete event systems, a rule-based control software development method was developed by researchers of Hitachi in Japan [Komoda *et al* 84, Tashiro *et al* 85]. In rule-based control software development, the control logic is described by IF-THEN forms, and control actions are automatically inferred from the controlled system situation using the production system methodology. Since control logic (as rules) is embedded in control software (as data rather than procedures), control logic can easily be understood and independently modified. Thus, when the control system requirements change, the specification description expressed in rule forms can be easily modified. To reduce the searching time in the inference process, a *Meta-rule*, defined as "IF conditions THEN SELECT <rule group name>", is used to localise the scope of the search without specifying any actions. Although the hierarchical structure of rules is discussed, method of how to construct a set of rules when the system is obscure is not given. Additionally, the concurrency and real-time issues are not discussed.

Pathak and Krogh [89] developed an executable non-procedural concurrent operation specification language, COSL, for discrete manufacturing system control. The COSL user can specify the control logic for a discrete manufacturing application in terms of a collection of *operations*. The execution of a COSL operation represents a control step required to achieve the overall control objective. Each operation is defined by specifying a set of *conditions* and *actions*, i.e., by rule-based forms. The specification expressed in COSL can be automatically transformed into an executable control program. The deficiency of COSL is that the communication mechanism between the concurrent operations is not clearly defined. Also verification for the specification specified by COSL is not investigated.

Rule-based formalism has also been applied in the development of PLC's by Barker *et al* [89] in which a rule-based procedure for the automatic generation of code for a PLC was proposed. To provide an integrated environment to the development of discrete event dynamic systems, Barker and Song [92] also proposed a rule-based simulation tool for PLC high level development based on Petri nets. An extended Petri net called a programmable logic controller net (PLCNet) is defined by the authors, which forms the basis of the rule-based simulator. The input to the simulator is in rule-based forms. Any errors in the rule-based description can be detected by the simulator (using PLCNet semantics) and corrected before implementation. Although it is very useful to use a simulation technique to diagnose the design fault, Petri net analysis technique such as net invariants [Lautenbach 87] could also need to be considered. In this work, the problem of requirement elicitation using rules is not discussed and no method is described.

Wilson and Krogh [90] proposed a methodology for the development of discrete event systems. The methodology addresses three issues:

- the specification of the system behaviour using a rule-based formalism,
- the generation of models from specification using Petri nets,
- analysis of the Petri net modelled specification.

Two steps for constructing the rule-based specification are stressed. The first step is to identify a set of discrete state variables that characterise the operation of the system components. The second step is to identify the causal relation for state transition rules in those variables. The basic elements of the specification formalism are the *state variables* and the *transition rules*. Each state variable has a set of *discrete values*. In generating the Petri net model, each state value in the system specification is mapped into a distinct *place*, and state transition rules from the system specifications are mapped into Petri net *arcs* and *transitions*. The reachability method is used to analyse the system behaviour. To efficiently analyse the behaviour of system subcomponents in isolation from the overall state behaviour of the system, an algorithm is developed to reduce the reachability graph. It should be pointed out that the firing rule of Petri nets is changed by the authors as follows:

> If a state value is used as a condition on two or more transitions that are simultaneously enabled, all of the enabled transitions may fire simultaneously.

To support this change, the authors state:

> Physically speaking, however, the conditioning of one state variable's state transition on another's state value should have no effect on other state variables that also happen to be conditioned on that state value.

Curiously, in the example used by [Wilson and Krogh 90], the marking which would be generated using simultaneous firing rule is not reflected in the author's reachability graph. Also the author's change, of firing rule, which is apparently simple, implicitly introduces a priority in contention nets. Consider the Petri net shown in Fig.2.19. It can easily be shown that the priority of transition t_2 is higher than or equal to the priority of transition t_1. This topic is discussed in Chapter 3. Wilson and Krogh's proposed method is limited because of the lack of decomposition in the rule-based formalism. Also, although the reachability graph is useful to analyse the system behaviour, a formal verification logic or technique may need to be developed because of the deficiency of Petri nets in describing declarative constraints for systems.

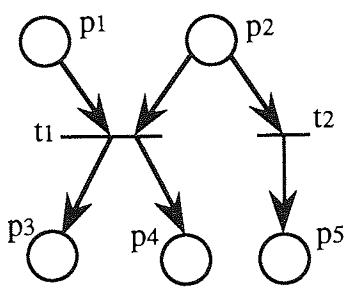


Fig.2.19 An Example of Petri nets

A rule-based design methodology for control problems has also been proposed by Etessami and Hura [91] who use an Abstract Petri Net (APN) which combines the features of timed and coloured Petri nets. In APN the timing is expressed by associating a delay function with each place and an activation time function with each transition. According to their rule-based design methodology, a designer first identifies the activities of the system, then formalises, by means of set of (behaviour) rules, the conditions under which the identified activities are affected. Each activity is represented by an APN transition. Additionally, the designer also needs to formalise a set of transient attributes representing the status of a system. Transient attributes are represented by means of places and token types of the APN model. Each behavioural rule is defined as a set of input predicates associated with each transition, these provide the enabling conditions for the transition in APN.

From the point of view of rule-based formalisms, the rule-based method proposed by Etessami and Hura [91] is similar to the method proposed by Wilson and Krogh [90], although APN is more powerful in a descriptive aspect than the Petri net model used in [Wilson and Krogh 90]. However, it is difficult to model a system at different levels of detail since the APN is not hierarchical, and decomposition and composition methods are not discussed.

In addition to the applications in industrial control and discrete event systems, the rule-based formalism was also applied to the information system development. To seek a new approach which is not only applicable to the information system development but also good at maintaining the representation of the system specification, a rule-based development environment, called RUBRIC (rule-based representation of information-systems concepts) is proposed by Assche *et al* [88], Loucopoulos and Layzell [89]. The aim of the RUBRIC project is to investigate the rule-based specification of information processing systems. This approach involves building a knowledge base of business facts, rules, policies, and decisions which can be used to control application programs. This has the effects that any change in the business environment directly relates to modifications in the knowledge base. Rules are classified into two types in RUBRIC named: *static* and *dynamic*. A static rule has the conventional IF-THEN construct. A dynamic rule can be regarded as consisting of a *trigger*, which is an expression describing the conditions under which an action part should be considered for execution. *Preconditions*, which are expressions, must be true if an action part is to be executed, given the occurrence of the associated trigger, and an *action part*. It becomes clear that dynamic rule has a richer structure than the simple IF-THEN construct, and that one could distinguish between triggers and preconditions. Here, preconditions are evaluated only when a trigger has occurred. A trigger causes a dynamic rule to be considered for firing. Following is an example of a dynamic rule from [Loucopoulos and Layzell 89], in which the WHEN clause is the trigger; the IF clause is the pre-condition, and THEN clause is the action.

```
WHEN   end-of-month
IF     product <> 'A' or 'B' or 'C'      AND
       quantity of product < reorder-level of product
THEN   arise-order (* reorder stock *)
```

A trigger is actually used to describe a timing point at which some condition acts on the conventional rule. This indicates that the concept trigger might be valuable for describing a heavily timing related condition in a real-time process-control systems, especially when the condition is controlled by the physical system under computer control. This has been shown by Heninger [80], Faulk and Parnas [88], and David and Alla [92] in which condition and condition-changing are treated separately.

It is worth pointing out that the rule-based structures proposed in [Wilson and Krogh 90, Etessami and Hura 91] for real-time process-control systems are static rules. In fact, Estessami and Hura [91] have already identified the objects described by dynamic rules in terms of activities, conditions, and transient attributes, but these objects were not formalised by a unique rule-based formalism. A static rule structure is suitable for describing the state-based causal behaviour (i.e. what to do), but it is not suitable to describe the stimuli (events) which give rise to the state-based causal behaviours (i.e. when to do) because it fails to distinguish between the state and the stimuli (events). Distinguishing between "what to do" and "when to do" is important for real-time process control systems because a real-time process control system is often embedded in, and forms part of, a physical environment and state transitions are often trigged by the stimuli (events) coming from environment. A dynamic rule provides the possibility of such a distinction because:

i) most real-time process-control systems are stimuli (events) driven and stimuli (events) are usually generated by the physical processes under (computer) control.
ii) a dynamic rule structure explicitly distinguishes between states (represented by preconditions) and stimuli (events) coming from the environment (represented by triggers).
iii) a dynamic rule structure provides a means of describing the interface between the behaviour of a controlled physical system and the action performed by the computer system.

Summary

In this chapter, certain mathematical models of discrete event systems have been examined and attention has been focused on the notion of states and events. Petri nets and Grafcets and their variants have been investigated in more depth and the relationships between the formalisms have been examined. In fact, to some degree, the formalisms express equivalent ideas. For instance, ESMs and CRSMs are quite similar. Finally, a rule-based development method was reviewed and it was concluded that for such approaches the dynamic rule is superior to the static rule for describing real-time systems.

Chapter 3

Make Implicit Grafcet Communication Explicit

3.1 Introduction

The syntax and the operational semantics of Grafcet were introduced in Chapter 2. This chapter investigates Grafcet as defined in [IEC848 88, David and Alla 92] in the following aspects:

i) analysis of the Grafcet model;
ii) identification of implicit communication in Grafcet;
iii) methods of removing implicit communication.

The purpose of this chapter is to provide a method to make the implicit communication explicit in Grafcet such that those possible behaviours can be analysed explicitly.

3.2 Grafcet Model Formal Analysis

3.2.1 Analysis of Evolution Rule

Among all the evaluation rules defined in Grafcet (see 2.2.3 in Chapter 2), the simultaneously firing rule is extremely important because its application results in a deterministic behaviour. It is this rule that constrains Grafcet such that it cannot model non-determinism due to non-deterministic choices (i.e. situations in which two or more alternative transitions are firable but only one can be fired). The simultaneously firing rule, which invalidates the notion of non-deterministic choice, is often called "interpreted parallelism" in Grafcet (see Fig.3.1).

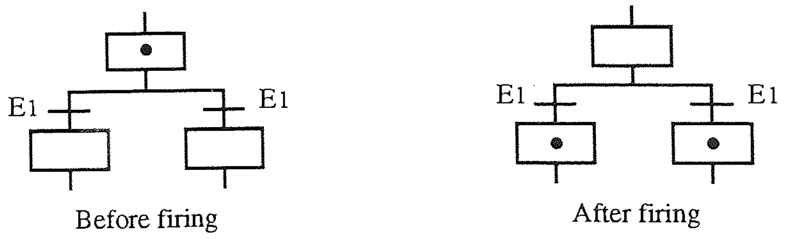


Fig.3.1 Simultaneously firing in Grafcet

In addition to the simultaneously firing rule, reactivative firing is another peculiar feature of Grafcet. Reactivative firing means that a step is activated again while it is still active. After reactivative firing, a step remains active according to the evolution rules defined in Grafcet. However, by observing the control-flow aspect of Grafcet only (i.e. without looking at the execution effects of action associated with the active step), it is difficult to determine whether an active step has been activated twice because the step works like '*flip flop*' (see Fig.3.2).

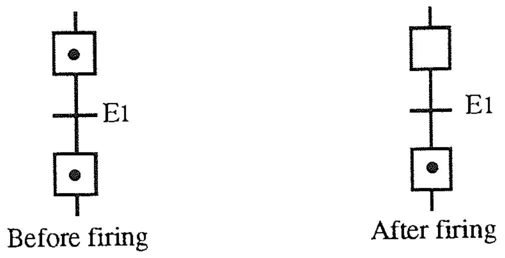


Fig.3.2 Reactivative firing in Grafcet

Now let us discuss the roles of steps and transitions in Grafcet. Each transition in Grafcet acts as a guard passing "control" from one or more its predecessor steps to one or more its successor steps. A transition is active in the sense that it has the ability to split one control flow into multithreading or merge multithreading control flows into one. Compared to the roles of transitions, steps in Grafcet seem to be "passive" because the control resources (i.e. tokens) held by them are transported from one to another by transitions. However, in the case of simultaneous firing, a step is capable of splitting or merging control flow(s) as well. Let us consider the Grafcet shown in Fig.3.3. At beginning, there are two threads of control flows. After simultaneous firing, these control flows are merged into one. Note that each transition has only one preceding step. So the merging is carried out by the shared step rather than by transitions. Similarly, Fig.3.1 is an example in which a step splits the control flow under simultaneous firing.

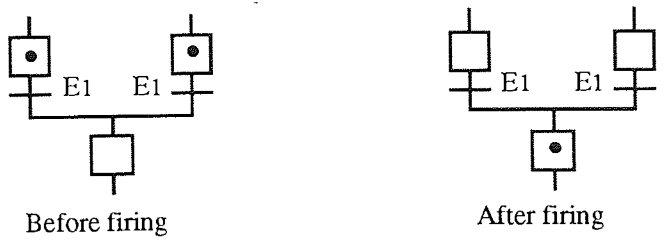


Fig.3.3 An example of merging flows by a step

3.2.2 Receptivity and Event

Two types of variables are defined in Grafcet: *external* and *internal*. An external variable is a Boolean variable which may either come from the controlled process such as a physical process or the outside world such as an operator. An internal variable is a Boolean variable relating either to the situation of Grafcet such as the step active flags or to an associated Grafcet process or function such as a calculation and a counter function.

Receptivity is defined based on concepts of *condition* and *event*. A condition is a Boolean expression of the external and internal variables such as $C_1 = a + b \bullet X$, where a and b can be external or internal variables and X is a step's active flag. Special case is the 'always true' condition, $C = 1$.

An event, denoted by $\uparrow a$ or $\downarrow a$, represents a rising or falling edge of an external variable (or of an external variable function) such as $E_1 = \uparrow a$ and $E_2 = \downarrow(a+b)$, where a and b are external variables. Special case is the 'always occurrence' event, $E = e$.

A receptivity can always be considered as an expression E•C consisting of two parts, an event E and a condition C. For each transition in Grafcet, a receptivity has to be defined.

In Grafcet, an event occupies a point in time. From point of view of time, an event is formally defined as [David and Alla 92, David 93]:

Let a be an external variable such that $a = 1$ in the time intervals $[t_1,t_2)$, $[t_3,t_4)$, and $a = 0$ in the time intervals $[t_0, t_1)$, $[t_2, t_3)$ etc., such that $t_1 < t_2 < t_3 < t_4$. The event $\uparrow a$ occurs at times t_1, t_3 and the event $\downarrow a$ occurs at times t_2, t_4 (as shown in Fig.3.4). Let b be an external variable and C be a condition (i.e., an expression of Boolean variables based on the logic operator "•" and "+").

1) $\uparrow(a \bullet b)$ is an event which occurs at the same time as $\uparrow a$ each time that $b = 1$ at the corresponding point in time, (or vice versa).
2) $\uparrow(a+b)$ is an event which occurs at the same time as $\uparrow a$ each time that $b = 0$ at the corresponding point in time, (or vice versa).
3) $\uparrow a \bullet C$ is an event which occurs at the same time as $\uparrow a$, each time that $C = 1$ at the corresponding point in time.
4) the product of an event E and a condition C, E•C, is an event. If the condition is true then the event E•C occurs when event E occurs.
5) if E_1 and E_2 are two events based on 1) to 4), then
   i) $E_1 \bullet E_2$ is an event which occurs when both E_1 and E_2 occur simultaneously, (such as $E_1 = \uparrow a \bullet c$ and $E_2 = \uparrow a \bullet b$). When E_1 and E_2 are independent, $E_1 \bullet E_2$ will never occur;
   ii) $E_1 + E_2$ is an event which occurs when either E_1 or E_2 occurs.

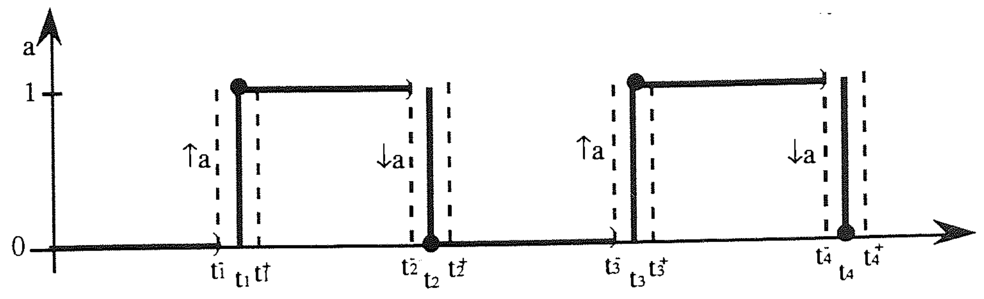


Fig.3.4 Graphical representation of an event

Representations of typical condition and event are shown in Fig.3.5 (where, at t_3 it is assumed that the events are not independent)..

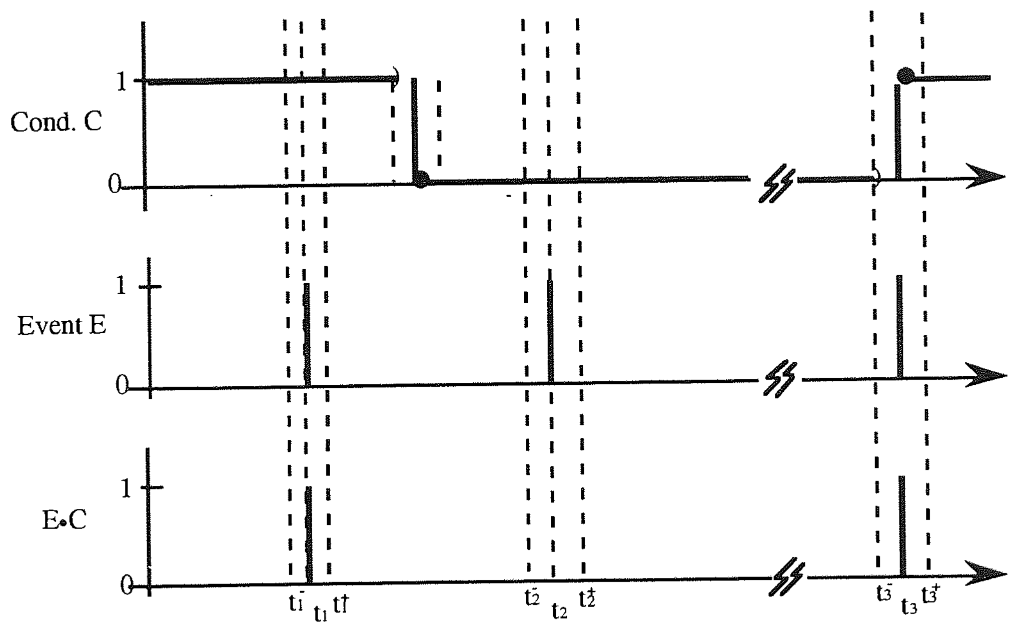


Fig.3.5 Graphical representation of condition and event of Grafcet

Following are two hypotheses made in Grafcet:

Hypothesis-1

Two independent external events (or simply events) never occur simultaneously.

Hypothesis-2

Grafcet changes over from one to another stable situation with zero duration.

A situation in Grafcet is said to be *stable* if no transition can be fired without occurrence of an event, otherwise it is called as *unstable* situation.

The following are some properties about events based on the hypotheses and the definition that symbol ↑ takes priority over the operators "•" and "+" (e.g. $\uparrow a \bullet b = \uparrow(a) \bullet b$ and $\uparrow a + b = (\uparrow a) + b$).

1) $\uparrow a = \downarrow a'$ (3.1)

(a' is the complement of a, i.e. if $a=0$ then $a'=1$, and if $a=1$ then $a'=0$)

2) $\uparrow a \bullet a = \uparrow a, \quad \uparrow a \bullet a' = 0, \quad \downarrow a \bullet a' = \downarrow a, \quad \downarrow a \bullet a = 0$ (3.2)

3) $\uparrow a \bullet \uparrow a = \uparrow a, \quad \uparrow a \bullet \uparrow a' = 0, \quad \uparrow a \bullet e = \uparrow a$ (3.3)

4) If a and b are two independent variables, then

$$\uparrow(a \bullet b) = \uparrow a \bullet b + \uparrow b \bullet a, \quad \uparrow(a + b) = \uparrow a \bullet b' + \uparrow b \bullet a' \tag{3.4}$$

5) If a, b and c are three independent variables, then

$$\uparrow(ab) \bullet \uparrow(ac) = \uparrow a \bullet bc \tag{3.5}$$

Property 5) shows that the two events $\uparrow(ab)$ and $\uparrow(ac)$, which are not independent since ab and ac depend on the same variable a, can occur simultaneously when a changes from 0 to 1, if b and c have the value 1 at this time.

3.2.3 Analysis of Implicit Dependency

In Grafcet, communication between different activities associated with steps can be described implicitly via the receptivity associated with each transition. Specifically, the step's active state can be *inspected* by a transition which wishes to fire. In the process of inspection, the step's active state is checked but not changed by the transition. For example, in Fig.3.6(a) the receptivity $R = X_{20}$ on the transition between step40 and step50 means that the active state of step20 is inspected by the transition. If step20 is active, then the transition may fire (and the active state of step20 is unaffected by the firing). Inspection implicitly describes a causal order between steps. However, the implicit nature of the dependency makes it difficult to distinguish whether steps occur in a particular intended order. For example, in Fig.3.6(a), step occurrence sequence σ=[10||40];[20||50];[30], where "||" stands for parallel and ";" stands for sequential, is actually an *unfeasible* state sequence for this model due to implicit dependency (because whenever the system is in step40, condition $R = X_{20}$ requires step20 to be in the active state for the transition between step40 and step50 to fire). Similarly, in Fig.3.6(b), transition (6) will never fire since step3 has already become inactive when transition (6) is enabled.

Clearly, implicit dependency should be avoided in Grafcet if possible. For example, the Grafcet shown in Fig.3.7(b) converts the implicit dependency shown in Fig.3.7(a) into explicit one [IEC848 88 pp.35], where (a) is functional equivalent to (b). Two Grafcets are *equivalent* if for all input sequences, possible according to the specifications, they produce the same output sequence (i.e., the same actions/state sequences) [David and Alla 92]. If a Grafcet contains implicit dependency, it is desirable to make this kind of dependency explicit to facilitate analysis of the system behaviour.

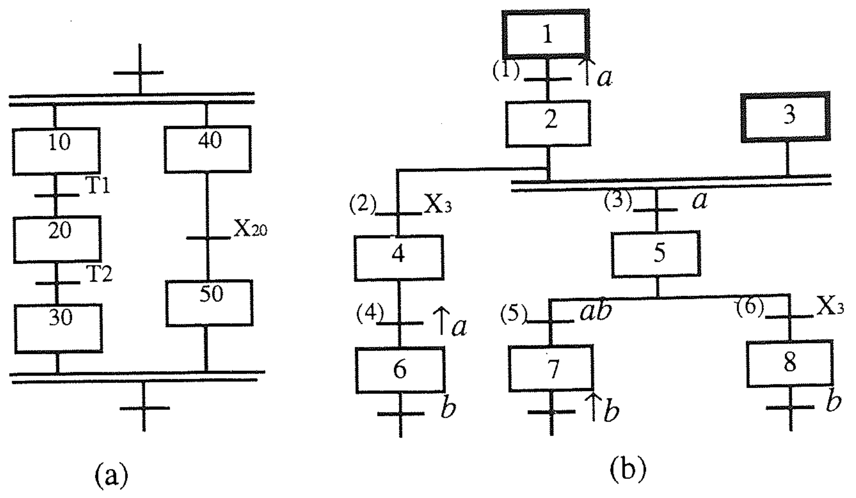
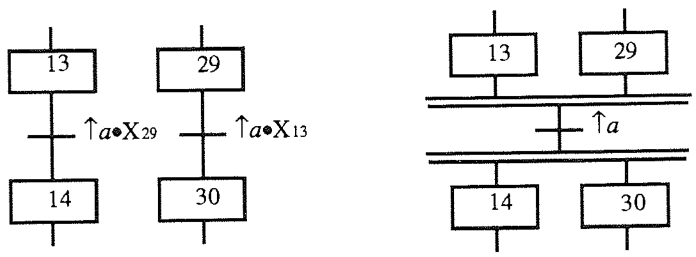


Fig.3.6 Grafcets including implicit dependency



(a) Grafcet with implicit dependency (b) Preferred Grafcet

Fig.3.7 Suggested and not suggested Grafcets

In fact, the "inspection mechanism" effectively describes the co-operation between two processes in which one is the *master* and the other is a *slave*, where the master is not obliged to consider the slave but the slave's behaviour depends on the master and has no effect on it. Although the inspection mechanism is very powerful and useful, unfortunately there is no graphical means of describing it explicitly in Grafcet (see page 49). Without considering context, the "inspection mechanism" expresses a *test* of the active state of a step, say s. It can be intuitively substituted by a "self-loop" to a dummy step s' if it is guaranteed that s' is active iff (if and only if) step s is active, (where the dummy step has no action associated with it). Alternatively it may recast as a zero test of the active state of step s' if it is guaranteed that s' is active iff step s is inactive.

This analysis indicates that the implicit dependency of a step's active state could be represented explicitly by introducing some extra step or by the equivalent of a Petri net inhibitor arc [Peterson 81]. However, this research has shown that it is extremely difficult to make the implicit dependency explicit in a fully functionally-equivalent manner without enforcing restrictions on the steps or transitions involved in the implicit dependency. The reason is that the dynamic behaviours arising from combinations of simultaneous firing and reactivation involving implicit dependency can be very complex.

To make a feasible discussion, the following assumption is made: that no reactivation exists for the steps and transitions involving the implicit dependency in Grafcet. The following discussion examines the inhibitor arc approach and the "self-loop" approach based on this assumption.

Consider the simple case of two transitions triggered by the independent events, as shown in Fig.3.8(a).

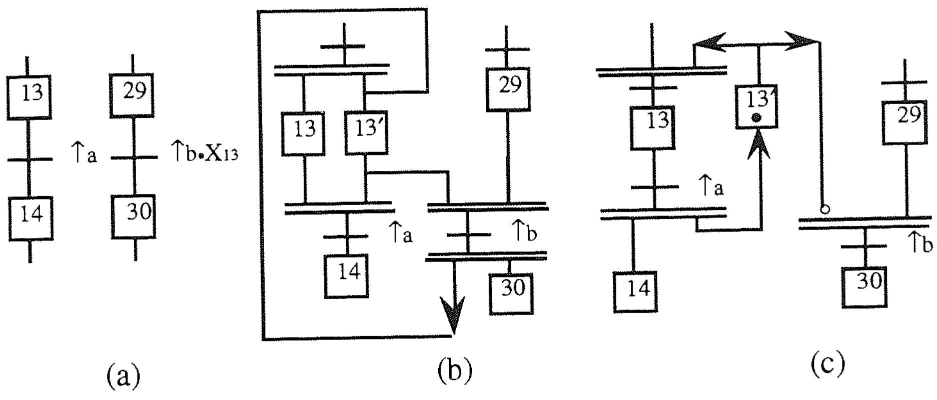


Fig.3.8 Grafcet and inhibitor arc Grafcet - 1

The dynamic behaviour of such a system can be described by following traces:

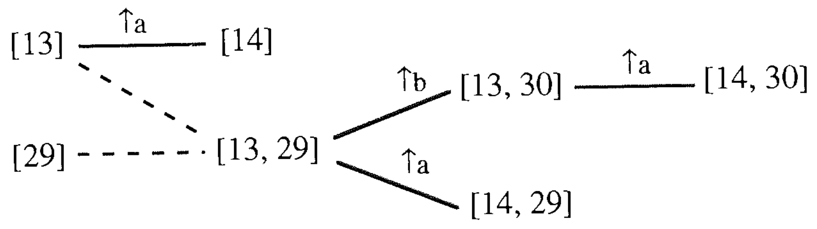


Fig.3.9 Traces of Fig.3.8(a)

A Grafcet without the implicit dependency can be formed using the "self-loop" approach as shown in Fig.3.8(b). Fig.3.8(b) is functionally equivalent to Fig.3.8(a) except for the internal structure control state S13′ which has no external effect. The dynamic behaviour of Fig.3.8(b) can be described by the following traces:

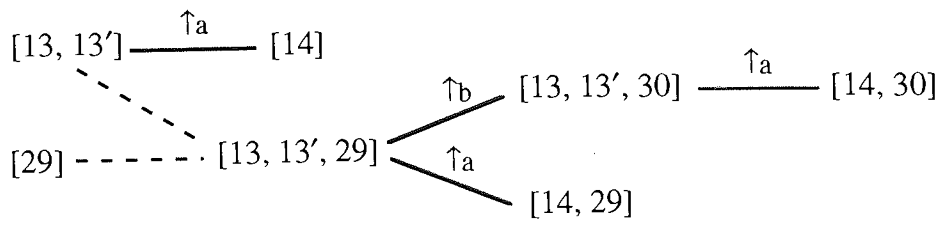


Fig.3.10 Traces of Fig.3.8(b)

However, limitations of this approach arise when two transitions are triggered by the same event. Consider for example the case of two transitions depending on the same event with an implicit state dependency as shown in Fig3.11(a).

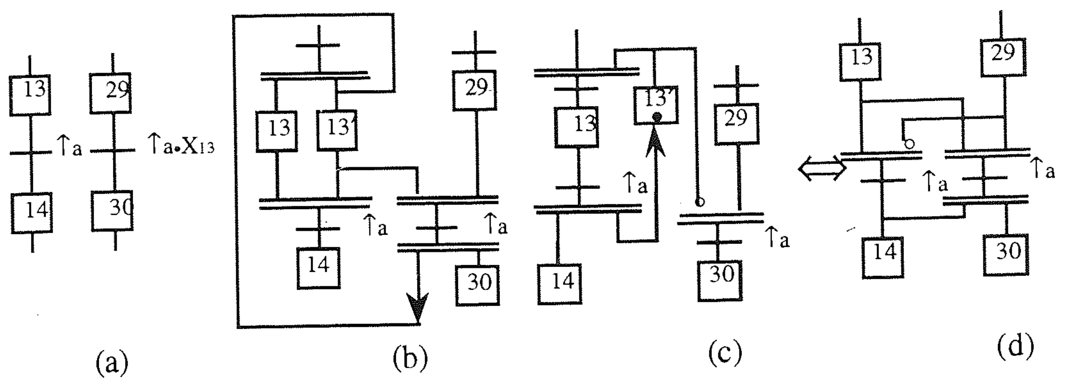


Fig.3.11 Grafcet and inhibitor arc Grafcet - 2

The behaviour of this system shown in Fig3.11(a) can be described by the traces:

$$\begin{tikzcd}
{[13]} \arrow[r, no head, "\uparrow a"] \arrow[dr, dashed, no head] & {[14]} & \\
{[29]} \arrow[r, dashed, no head] & {[13, 29]} \arrow[r, no head, "\uparrow a"] & {[14, 30]}
\end{tikzcd}$$

Fig.3.12 Traces of Fig.3.11(a)

If this model is translated into an explicit communication model using the "self-loop" technique, as shown in Fig3.11(b), then a different outcome occurs. The behaviour of model Fig.3.11(b) is described by the traces:

$$\begin{tikzcd}
{[13, 13']} \arrow[r, no head, "\uparrow a"] \arrow[dr, dashed, no head] & {[14]} & \\
{[29]} \arrow[r, dashed, no head] & {[13, 13', 29]} \arrow[r, no head, "\uparrow a"] & {[13', 14, 30]}
\end{tikzcd}$$

Fig.3.13 Traces of Fig.3.11(b)

In the lower part of the trace diagram, Fig.3.13, following the simultaneous firing of these transitions whose receptivity depends on $\uparrow a$, state S13′ is still active which is inconsistent with step S13 being inactive. Thus, the post-firing behaviour of the model of Fig.3.11(b) differs from the behaviour of the model of Fig3.11(a).

This problem can be overcome by modelling the system using an "inhibitor" arc approach as shown in Fig.3.8(c) and Fig.3.11(c). The behaviour of the inhibitor arc model shown in Fig.3.8(c) is defined by the traces:

$$\begin{tikzcd}
{[13']} \arrow[r, dashed, no head] & {[13]} \arrow[r, no head, "\uparrow a"] \arrow[rdd, dashed, no head] & {[13', 14]} & & \\
& & & {[13, 30]} \arrow[r, no head, "\uparrow a"] & {[13', 14, 30]} \\
{[29]} \arrow[rr, dashed, no head] & & {[13, 29]} \arrow[ru, no head, "\uparrow b"] \arrow[rd, no head, "\uparrow a"'] & & \\
& & & {[13', 14, 29]} &
\end{tikzcd}$$

Fig.3.14 Traces of Fig.3.8(c)

Similarly, the behaviour of the inhibitor arc model as shown in Fig.3.11(c) is defined by the traces:

$$\begin{tikzcd}
{[13']} \arrow[r, dashed, no head] & {[13]} \arrow[r, no head, "\uparrow a"] \arrow[rd, dashed, no head] & {[13', 14]} & \\
{[29]} \arrow[rr, dashed, no head] & & {[13, 29]} \arrow[r, no head, "\uparrow a"] & {[13', 14, 30]}
\end{tikzcd}$$

Fig.3.15 Traces of Fig.3.11(c)

An alternative inhibitor arc structure for Fig.3.11(c) is shown in Fig.3.11(d). This model has the same behaviour as the original model (Fig.3.11(a)) and avoids the need for a control state c.f. Fig.3.11(c). The behaviour of the inhibitor arc model Fig.3.11(d) is defined by the following traces which are exactly the same as the traces of Fig.3.11(a) shown in Fig.3.12.

$$\begin{tikzcd}
{[13]} \arrow[r, no head, "\uparrow a"] \arrow[rd, dashed, no head] & {[14]} & \\
{[29]} \arrow[r, dashed, no head] & {[13, 29]} \arrow[r, no head, "\uparrow a"] & {[14, 30]}
\end{tikzcd}$$

Fig.3.16 Traces of Fig.3.11(d)

Since the inhibitor arc models (Fig.3.11(c) and Fig.3.11(d)) avoid the inconsistencies of the self-loop model Fig.3.11(b), the inhibitor arc approach is used throughout this chapter.

In the following, Grafcet will be extended by introducing the inhibitor arc into it. Then an algorithm, to transform a Grafcet with implicit dependency into a functionally equivalent inhibitor arc Grafcet in which all the implicit step's state dependencies are described explicitly (or visually), will be presented.

3.3 Inhibitor Arc Grafcet — IAGrafcet

In the following the notion of inhibitor arc Grafcet will be introduced formally by forming analogies with inhibitor arc Petri nets which were introduced in Chapter 2. In an inhibitor arc Petri nets, the set of inhibitor arcs, say B, can be considered as a relation which is defined as B ⊂ P×T, where P represents the set of places and T represents the set of transitions of Petri nets. As discussed earlier, each inhibitor arc is represented by a directed arc which connects a place and a transition and ends with a small circle. If place *p* is an inhibitor place of a transition *t* then that transition *t* is only enabled if place *p* contains zero tokens.

Grafcet can be extended by introducing the inhibitor arc in a similar way. The extended Grafcet will be called *inhibitor arc Grafcet* or IAGrafcet. The inhibitor arc in extended IAGrafcet has a similar meaning to that defined in inhibitor arc Petri nets and the proposed notation and firing rule is shown in Fig.3.17. An inhibitor arc from step *s* to transition *t* means that transition *t* cannot be enabled when step *s* is active.

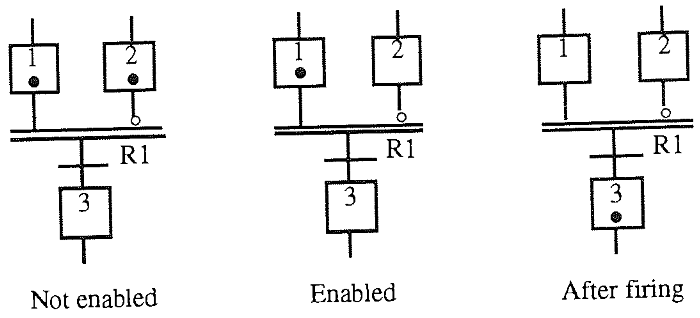


Fig3.17 Inhibitor arc Grafcet

3.4 Eliminating Implicit Dependency Using Inhibitor Arcs

Suppose that the state of step *s* is implicitly used in receptivity R which is associated with transition *t* (see Fig.3.18(a)). Let IIT(*s*)/IOT(*s*) represent the set of Immediate Input Transitions/Immediate Output Transitions of *s* and IIS(*t*)/IOS(*t*) represent the set of Immediate Input Steps/Immediate Output Steps of *t* (see Fig.3.18(a)). We assume that *s* is not self-looped, otherwise we can easily introduce a transition and a step to change the self-loop into a loop containing two transitions and two steps.

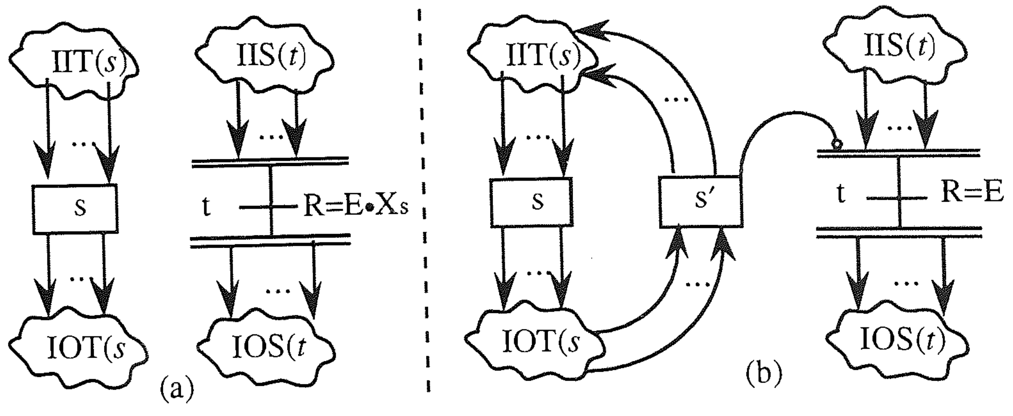


Fig.3.18 A Grafcet and its transformed inhibitor arc Grafcet

The transformation to eliminate the implicit state dependency with the help of inhibitor arc can be done by the following steps:

Algorithm1
For each transition t with an implicit dependency on step s:

Step1:	introducing a dummy step s'.
Step2:	define s' as the input step for all the transitions belonging to IIT(s) and connect s' with each of them by an normal arc.
Step3:	define s' as the output step for all the transitions belonging to IOT(s) and connect s' with each of them by an normal arc.
Step4:	introduce an inhibitor arc which leaves s' and reaches to t, i.e. put s' into IIS(t).
Step5:	if s does not belong to the initial state, then put s' into the initial state.
Step6:	delete the state of step s from receptivity R.

The dummy step s' has the following characteristic (see the proof below):

s' is active iff step s is inactive and s' is inactive iff s is active.

By following the steps defined above, all the implicit usage of step's active flags in receptivities of Grafcet model can be made explicit with inhibitor arcs and dummy steps as shown in Fig.3.18(b).

Theorem3.1

If G is a Grafcet including implicit dependency and G′ is the transformed IAGrafcet derived using the Algorithm1, then G and G′ are functional equivalent.

The correctness of this algorithm can be shown by proving that the active state of s is still the *necessary* condition for t to be enabled in G′, and the inactive state of s is still the *sufficient* condition for t to be disabled in G′.

Lemma

By definition of inhibitor arc from s' to t in G′, we have
t is enabled $\Rightarrow$ s' is inactive and s' is active $\Rightarrow$ t is disabled

Proof of Theorem3.1:

If we can prove that s and s' are not only mutual exclusive but also complementary, then based on the deductive inference, we can derive that:

t is enabled $\Rightarrow$ s' is inactive $\Leftrightarrow$ s is active (1)

and s is inactive $\Leftrightarrow$ s' is active $\Rightarrow$ t is disabled (2)

From Step2 of the Algorithm1, we have

$$\forall t' \in \mathrm{IIT}(s),$$
$$s' \in \mathrm{IIS}(t') \wedge <s',t'> \in \mathrm{NA} \wedge s' \notin \mathrm{IOS}(t') \wedge s \notin \mathrm{IIS}(t') \wedge s \in \mathrm{IOS}(t') \wedge <t',s> \in \mathrm{NA} \quad (3)$$

where each arc is denoted as a pair $<s, t>$ (s — step, t — transition) and NA is the set of all normal arcs. Equation (3) means that s' will be changed from active into inactive whenever s is changed from inactive into active by firing $t' \in \mathrm{IIT}(s)$. That is,

$$s \text{ is active} \Rightarrow s' \text{ is inactive} \quad (4)$$

Symmetrically, from Step3 of the Algorithm1, we have

$$\forall t'' \in \mathrm{IOT}(s)$$
$$s' \in \mathrm{IOS}(t'') \wedge <t'',s'> \in \mathrm{NA} \wedge s' \notin \mathrm{IIS}(t'') \wedge s \notin \mathrm{IOS}(t'') \wedge s \in \mathrm{IIS}(t'') \wedge <s,t''> \in \mathrm{NA} \quad (5)$$

Equation (5) means that s will be changed from active into inactive whenever s' is changed from inactive into active by firing $t'' \in \mathrm{IOT}(s)$. That is,

$$s' \text{ is active} \Rightarrow s \text{ is inactive} \quad (6)$$

Based on (3) and (5),

$$\forall t' \in \mathrm{IIT}(s) \text{ and } \forall t'' \in \mathrm{IOT}(s)$$
$$s' \in \mathrm{IIS}(t') \wedge <s',t'> \in \mathrm{NA} \wedge s' \in \mathrm{IOS}(t'') \wedge <t'',s'> \in \mathrm{NA} \quad (7)$$

Equation (7) means L=$\{t',s,t'',s'\}$ is a loop. By examining Step5 since only one step in L is initialised, results (3), (5) and (7) yield

$$s' \text{ is inactive} \Rightarrow s \text{ is active}$$
$$\text{and} \quad s \text{ is inactive} \Rightarrow s' \text{ is active} \quad (8)$$

(4), (6), (8) show that mutual exclusion and complementation between s and s' are guaranteed.

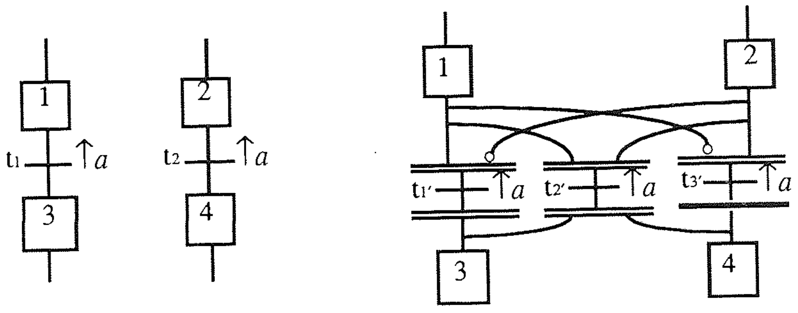
3.5 Simultaneous Firing Rule Analysis

The inhibitor arc can be used not only to eliminate implicit dependency, but also to eliminate simultaneous firing for a *class* of Grafcet. That is, for a special class of Grafcet, the simultaneous firing rule is not a fundamental restriction in inhibitor arc Grafcet. This section examines how this class of Grafcet can be transformed into a functional equivalent inhibitor arc Grafcet in which only one transition can be fired at any instant.

Consider the class of Grafcet which satisfies the following requirement: any receptivity associated with transition contains only a basic event such as e, $\uparrow x$ or $\downarrow x$, where e is the always occurrence event and x is an external variable.

In the specified class of Grafcet, simultaneous firing is denoted by transitions triggered by the same event. The problem is to show how such transitions can be transformed into a functionally equivalent set of transitions in which any two of them are mutually exclusive. Before giving the algorithm, consider the following example.

Example: Let $\uparrow a$ be the event associated with transitions t_1 and t_2 as shown in Fig.3.19(a). Since t_1 and t_2 are triggered by the same event, they may be fired simultaneously when a changes from *0* to *1* if they are both enabled. The problem is to transform the Grafcet of Fig.3.19(a) into a functionally equivalently inhibitor arc Grafcet in which all the transitions are mutually exclusive, such as that shown in Fig.3.19(b).



a) Ordinary Grafcet b) Inhibitor arc Grafcet

Fig.3.19 Grafcet and its equivalent transformed inhibitor arc Grafcet

The basic idea is quite straightforward. Suppose there are k transitions triggered by the same event. The worst case is that all k transitions are enabled and could be fired simultaneously. By the functionality of inhibitor arc, we can simply use 2^m-1 transitions, where m is the total number of input steps of the k transitions, to substitute these k transitions and make the 2^m-1 transitions mutually exclusive based on mutually exclusive *binary* code (see Fig3.19).

Algorithm2

Step1: Let T = $\{t_1, t_2, \dots t_k\}$ be the set of transitions triggered by event $\uparrow a$.
Let IS and OS be the sets of Input Steps/Output Steps of all k transitions, where IS = $\cup$IIS(t_i) and OS = $\cup$IOS(t_i) (i=1 to k), and let NIA be the set of Normal Input Arcs of each transition denoted by NIA(t) = $\{<s, t> \mid s \in$ IIS(t)$\}$, where s is connected with t by a normal arc.

Step2: Define 2^m-1 transitions such that each one has m input steps, where m = #IS. Let us consider the m input steps as m ordinal binary bits such that the active state of each input corresponds to a "*1*" bit and the inactive state corresponds to a "*0*" bit. Let each of the 2^m-1 new transitions correspond to exact one binary code from *1* to 2^m-1. Generate 2^m-1 mutually exclusive transitions based on the following criteria:

If the bit in the binary code associated with transition is "1", then connect the corresponding input step (identified by its ordinal bit) and the transition using a normal arc, otherwise using an inhibitor arc.

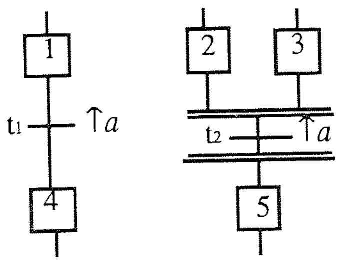
Step3: For each new transition t' and each transition $t_i \in$ T, if the steps in IIS(t_i) are all connected with t' using normal arcs, i.e. NIA(t_i) $\cap$ NIA(t') = NIA(t_i) (i = 1 to k), then define the steps in IOS(t_i) as the output steps of t'.

Step4: Associate R = $\uparrow a$ with each new transition.

Step5: For each new transition t', if t' has no output steps, then delete t' and all the arcs connected with it.

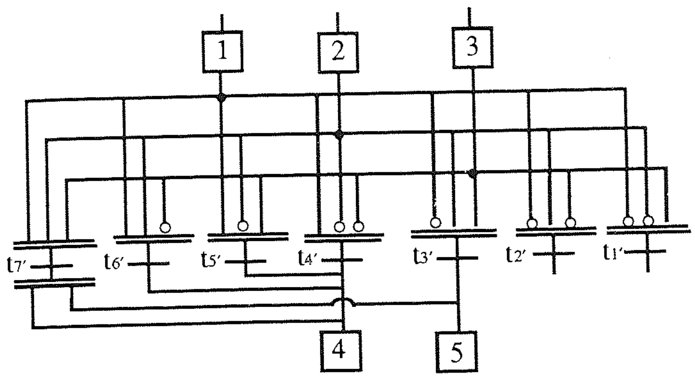
Step6: For each new transition t' and each transition $t_i \in$ T,
if NIA(t_i) $\cap$ NIA(t') $\neq \varnothing \wedge$ NIA(t_i) $\cap$ NIA(t') $\neq$ NIA(t_i)
then for each $f \in$ NIA(t_i) $\cap$ NIA(t'), where $f = <s, t'>$
i) delete f;
ii) change R associated with t' as R$\bullet$X$_s$.

Fig.3.20 shows how the Algorithm2 works by an example. Step1 is illustrated by Fig.3.20(a). Step2 to Step4 are illustrated by Fig.3.20(b). Step5 and Step6 are illustrated by Fig.3.20(c). Note that Fig.3.20(c) can be further transformed using Algorithm1 presented in this chapter in order to eliminate the implicit dependency.



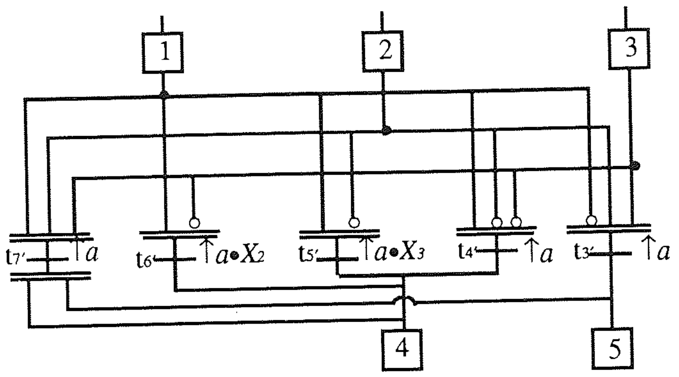
T = $\{t_1, t_2\}$
IS = $\{s_1, s_2, s_3\}$
OS = $\{s_4, s_5\}$
IIS(t_1) = $\{s_1\}$
IOS(t_1) = $\{s_4\}$
IIS(t_2) = $\{s_2, s_3\}$
IOS(t_2) = $\{s_5\}$

(a) Two transitions are triggered by the same event



Note:
The receptivity associated with each new transition $t_{i'}$ has the same form R = ↑a

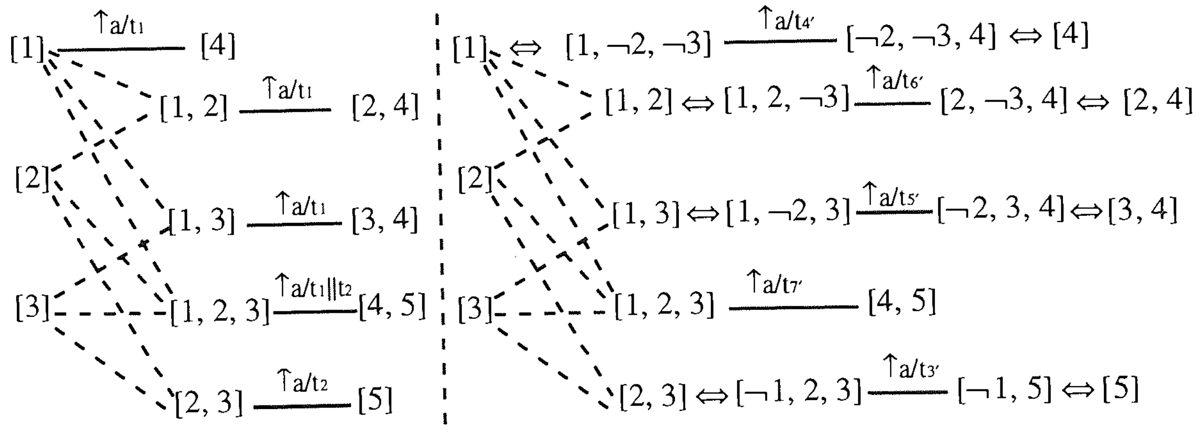
(b) Only one transition can be fired at any instant



(c) Functional equivalent inhibitor arc Grafcet

Fig.3.20 Graphical illustration of Algorithm2

The behaviours of the Grafcet of Fig.3.20(a) and Fig.3.20(c) are represented by the following traces:


[1] ↑a/t1 [4]
[1, 2] ↑a/t1 [2, 4]
[2]
[1, 3] ↑a/t1 [3, 4]
[3]
[1, 2, 3] ↑a/t1||t2 [4, 5]
[2, 3] ↑a/t2 [5]
[1] ⇔ [1, ¬2, ¬3] ↑a/t4′ [¬2, ¬3, 4] ⇔ [4]
[1, 2] ⇔ [1, 2, ¬3] ↑a/t6′ [2, ¬3, 4] ⇔ [2, 4]
[2]
[1, 3] ⇔ [1, ¬2, 3] ↑a/t5′ [¬2, 3, 4] ⇔ [3, 4]
[3]
[1, 2, 3] ↑a/t7′ [4, 5]
[2, 3] ⇔ [¬1, 2, 3] ↑a/t3′ [¬1, 5] ⇔ [5]

(a) Behaviour of Fig.3.20(a) (b) Behaviour of Fig.3.20(c)

Fig.3.21 Behaviour comparison of G and G′

From the comparison shown in Fig.3.21, it can be seen that G and G′ are functionally equivalent, and not more than one transition will be fired simultaneously in G′. For the implicit dependencies introduced by Algorithm2 they can be eliminated using Algorithm1.

Summary

This chapter has presented Grafcet and its analysis. To make the implicit dependency in Grafcet models explicit, Grafcet has been extended using inhibitor arcs. Note that all the hypotheses and firing rules of Grafcet are still applicable in the inhibitor arc Grafcet except the enabling rules. A method of removing the implicit dependency based on inhibitor arc has been presented. The advantage of inhibitor arc Grafcet is that it allows one to describe explicitly the co-operation between two processes, where one is master and the other is slave, i.e., inhibitor arc provides the means to describe the (dynamic) priority. Finally, the power of inhibitor arc is illustrated by showing that for a class of Grafcets the simultaneous firing rule is not a fundamental restriction.

Chapter 4 Sequential Function Chart (SFC) Analysis

4.1 Introduction

Sequential Function Chart (SFC) is a graphical representation for specifying the function and behaviour of the discrete-event part of a control system. SFC, defined in international standard IEC1131 [IEC 93], has gained widespread acceptance amongst manufacturers and has been implemented as the software front end for many control system products, such as programmable logic controllers (PLC's) produced by C.J. International, Eurilor, and Telemecanique in France, and Eurotherm Controls in U.K. Since SFC is often used in the design of synchronisation logic and control logic for multiple independent processes in applications which have implications for safety, an ability to analyse and to reason about SFC designs is a necessary pre-requisite for safety-critical system design. In this chapter, IEC standard SFC is introduced and the SFC model is analysed in detail. Secondly, the evolution rules of SFC are investigated. It is shown that SFC semantics are ambiguous and a revised definition based on a set of formal notations is proposed. Finally, the possibility of modelling and analysing SFC by Petri nets is discussed. To establish a framework for the analysis, an extended timed Petri net model and the transformation method from SFC to the extended model are defined; this provides the basis for the formal analysis and verification of SFC presented in Chapter 5.

4.2 SFC

4.2.1 SFC Model and Analysis Assumption

SFC is a semi-formal graphical notation for modelling the operation of a control system. The SFC model is derived from Grafcet and can be used to describe the system control flows and the instigation of actions in the system. For example, an SFC model will define what should be happening at a particular stage in a process and what assertions (or conditions) must be met before progressing to the next stage in the process. As discussed in Chapter 2, SFC has the same syntax as Grafcet which partitions the control system into *steps* and *transitions* interconnected by *directed links*, and SFC has the same evolution rules as those defined for Grafcet, except for the conflict situation in which a step is shared by more than one transition.

SFC is defined more from the consideration of the "interior system" (i.e., from implementation point of view) rather than from respect to the "exterior" (i.e., from description point of view). However, SFC is not a complete programming language although it is defined in IEC1131 as a language. In PLC applications, SFC has to be used in conjunction with a programming language such as ST (Structured Text) or LD (Ladder Diagrams) [IEC 93] because there is no notation in SFC which can be used to describe conditions or actions.

The use of a programming language to describe the conditions and actions with SFC provides a freedom of expression which is not necessarily consistent with a formal or deterministic approach. If restrictions are not imposed on the composition of such conditions and actions, then problems may arise in the analysis of the SFC concurrent behaviour due to low-level implicit communication in SFC which may have been introduced by unstructured programming. The reason is that the data flow (i.e., the relationship between the definition and usage of variables) expressed by a programming language has important effect on system execution behaviour but may not necessarily follow the explicit SFC control flow model. The lack of *consistency* between the explicit control flow expressed by SFC and the implicit data flow expressed by programming languages may make it difficult to analyse a design due to various possible behaviours caused by the implicit data flow dependency. For example, in Fig.4.1(a) if the condition c associated with transition T_1 involves the result generated by executing action a associated with step50, where c and a are described by a programming notation such as ST, then the state sequence (or system execution behaviour) derived from SFC model [10;40;20;50;30] is actually infeasible because it is impossible for step20 to be active before step50 according to the data flow dependency. However, such an infeasible state sequence is difficult to "spot" without both analysing the SFC model and performing a data flow analysis of the low-level (ST) program.

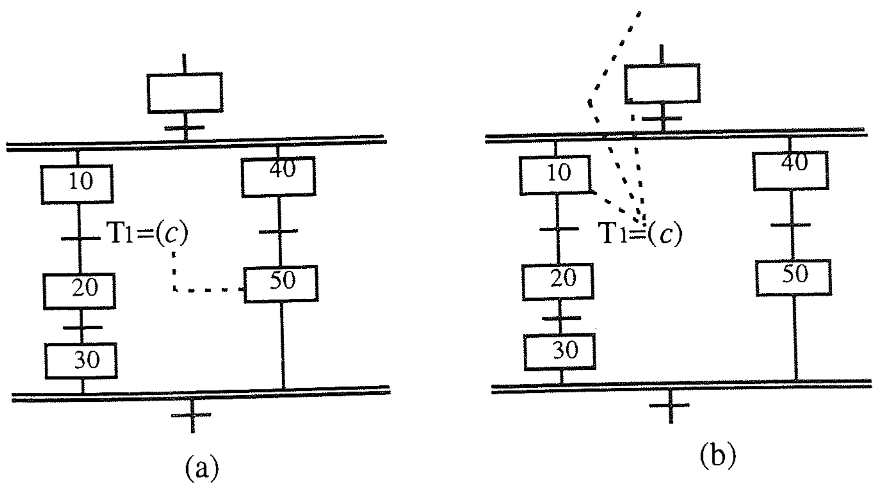


Fig.4.1 Data flow dependency of condition in SFC

To overcome the above problem, we may either consider the semantics of the programming language associated with SFC and its use to express the conditions and actions with SFC in our analysis. Alternatively, we can make an assumption or place a restriction on the application of the language in SFC in order to simplify the analysis. Considering a particular language will involve program analysis rather than system design analysis at the abstract SFC level, and is considered beyond of the scope of this thesis. About program analysis, the readers are referred to [Waters 82, Weiser 84, Jiang *et al* 91]. Therefore, to concentrate on the synthesis of system design, this thesis adopts the approach of imposing restrictions on the description of conditions and actions as implemented in the programming language. The following is the assumption made about conditions and actions:

Assumption:

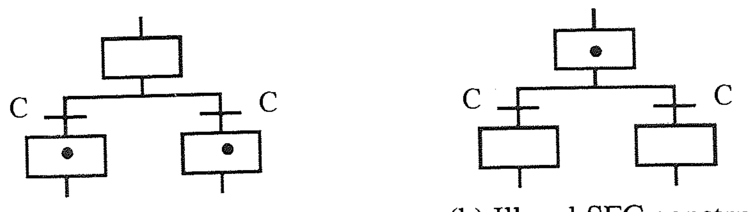
All the data flow dependencies involving conditions follow the explicit control flow expressed by SFC.

In another words, the values of variables appearing in a condition have to be defined at some predecessor step by executing action or are controlled by the environment (i.e., are controlled by an operator, operating system, or a physical system). A graphical representation of the assumption is shown in Fig.4.1(b).

This assumption implies that the condition associated with a transition must eventually become true after the transition is enabled. Also this assumption rules out the implicit dependency between a condition and the active state of a step. However, the latter can actually be lifted if inhibitor arc technique are considered as discussed in Chapter 3.

4.2.2 Evolution Rule Analysis

One important feature of SFC evolution rules is that SFC deals with the conflict situation differently from Grafcet. In SFC all conditions associated with transitions which share a common input step are forced to be mutually exclusive; otherwise if more than one condition becomes true simultaneously it is treated as an error (see Fig.4.2(a) and 4.2(b)). This feature means that non-deterministic choice cannot be modelled by SFC.



(a) Allowable Grafcet construct (b) Illegal SFC construct

Fig.4.2 Difference between Grafcet and SFC

SFC does not define clearly how to cope with reactivation (i.e., a step is active and its associated action is being executed when it is reactivated Fig.4.3(a)) and simultaneous reactivation (i.e., a step may be simultaneously activated by two input transitions Fig.4.3(b)). Although these conditions are unlikely to occur, the existence of the conditions constitutes a potential source of error.

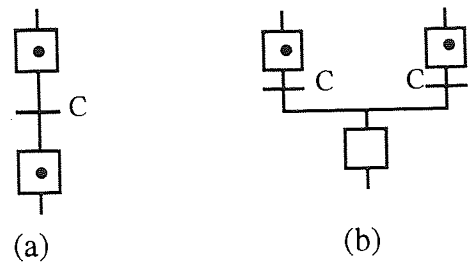


Fig.4.3 Reactivation and simultaneous reactivation

Specifically, these undefined situations cause problems because different SFC implementations based on IEC1131 may have different behaviour. Thus consistency between manufacturing products conforming with IEC1131 standard SFC is not guaranteed. Obviously this is not desirable. SFC design should always generate the same logical behaviour on any product which supports IEC1131 standard SFC as long as the executing speed is not considered. The above analysis indicates that a unified evolution system should be defined precisely.

Before presenting a formal definition of a unified set of evolution rules for SFC, the chapter analyses SFC to identify precisely the various areas of concern.

4.2.3 Step and Action

In SFC a step is either active or inactive at any given moment. A step is said to be in "active state" after its predecessor transition is fired and before its immediate successor transition is fired, otherwise it is said to be in its "inactive state". The duration of each step is determined by the firing of the transitions between which the step is situated. From an implementation point of view, "active" can be described logically as "1" with "inactive" described by logic value "0". However this definition is incomplete because it gives no information concerning the action execution and it does not show the correspondence between action duration and step duration, i.e. there is not a clear distinction between the *evolution system* of SFC and its *action's execution* system.

Each action has two external states: *running* and *stop*. Running means that an action is being performed and stop means that the action is not being performed (Fig.4.4). In this thesis it is assumed that each action has a fixed duration associated with its execution.

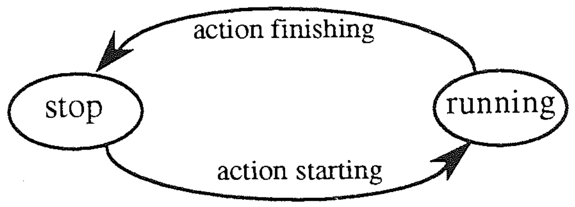


Fig.4.4 State diagram of action

Often, the duration of an action is supposed to be equal to the length of the period during which the corresponding step is active, but the actual correspondence is not always like this. The duration of an action can be different from the duration of the step with which the action is associated. This is particular true when a condition associated with a transition has to wait for some information coming in from the environment after the action associated with the transition's preceding step has been performed. Consider the two structures shown in Fig.4.5. Let conditions c_1 and $c_{1'}$ depend on the results generated by executing action "A" and condition c_2 depend on the information coming in from the environment. Then the "equivalence" between the duration of action and the duration of step holds for Fig.4.5(a) only. It does not hold for Fig.4.5(b) because there may be a time delay between the time point at which action "B" finishes its execution and the time point at which the information comes in from the environment. The different correspondence between the duration of a step's active state and the duration of an action's execution will become more obvious when action qualifiers (see below) defined for SFC are considered.

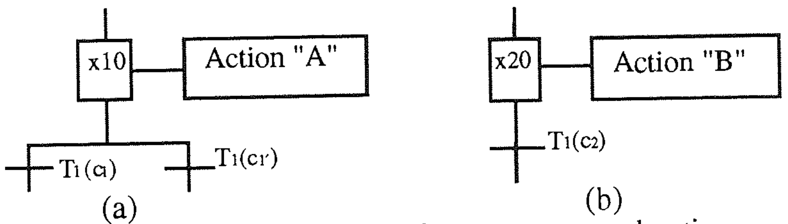


Fig.4.5 Correspondence between step and action

4.2.4 Action Qualifiers

SFC defines a set of qualifiers which can be used to describe or constrain the timing and duration of actions. The basic qualifiers defined in IEC1131 are shown in Table 4.1. By associating one or a combination of these qualifiers with each action more precise timing constraints can be defined to model real time tasks in a control system. The concept of associating qualifiers with action is, in principal, very powerful and proper combinations of them can describe a real-time task very well.

Unfortunately, the way in which the qualifiers have been defined gives rise to ambiguity which arises in the D-qualifier, see Fig.4.6(a). It may also lead to a mismatch between step state and action execution. For example, based on the definitions of the qualifiers, a system can be in a situation in which a step has become inactive but its associated action is still being executing (as in the qualifier S shown in Fig.4.6(b)) or is scheduled to be executed in some time future (as in the SD qualifier). Although this kind of mismatch between the evolution of SFC and the execution of action is allowed according to IEC1131, it is undesirable because the set of current active steps is inadequate to reflect the system real situation. These problems have generated a lot of queries world-wide requesting clarification from the IEC1131 working group [Lewis 92]. For example, it is not clear in IEC1131 which qualifier (P or N) should be used to execute the D qualified action after the specified delay elapses. Also, it is not clear whether a transition should be fired when its associated condition becomes true but the action associated with the transition's preceding step is still being delayed. Similar problems also arise in L qualified actions.

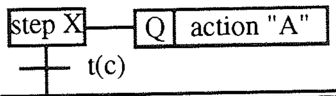
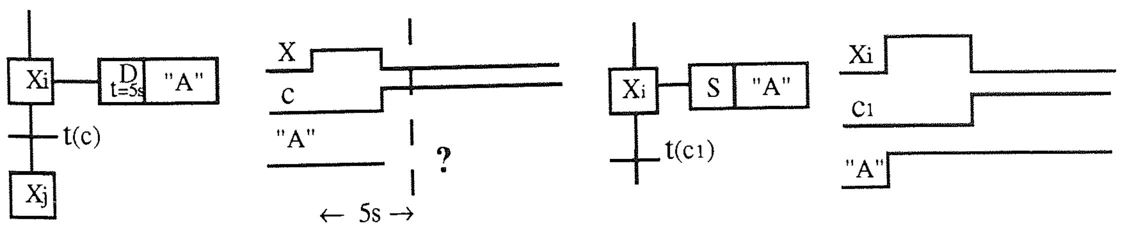
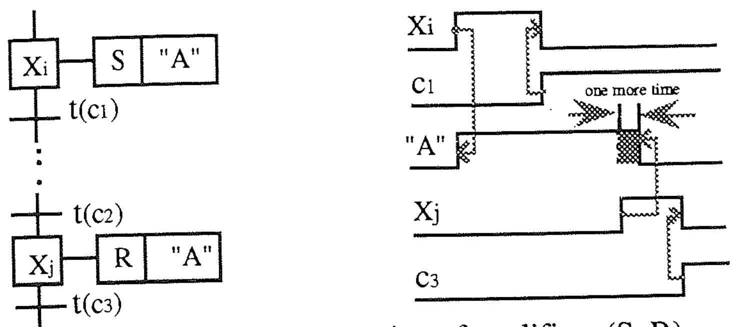
No.	Qualifier	Explanation (see figure on the right) 
1	N	Non-stored—A is started when X is activated and will keep executing continuously until c becomes true. A will be executed one final time after c becomes true.
2	S	Set (stored)—A is started when X is activated and will keep executing continuously even after c becomes true and X is deactivated. A is terminated by the R qualifer
3	R	Non-stored— A started by the S qualifier will be terminated when X is activated and A will be executed one final time.
4	L	time Limited — A is forced to be executed and finished within a specified time limit after X is activated.
5	D	time Delayed — A is delayed a specified time to start its execution after X is activated.
6	P	Pulse — A is started when X is activated and will be executed once only.
7	SD	Stored and time Delayed — concatenation of qualifiers S and D.
8	DS	Delayed and Stored — concatenation of qualifiers D and S.
9	SL	Stored and time Limited — concatenation of qualifiers S and L.

Table 4.1 Action Qualifiers defined in SFC

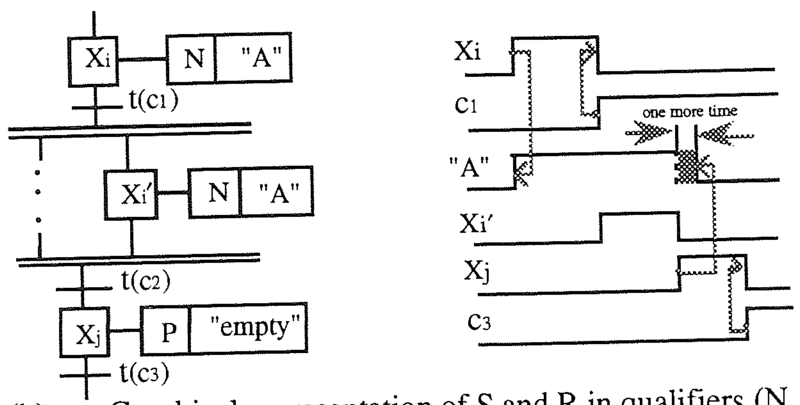


(a) (b)
Fig.4.6 Ambiguity and mismatch of qualifiers in SFC

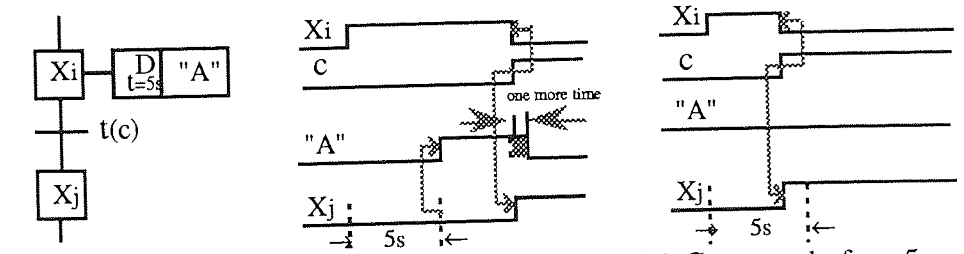
To avoid mismatch between the evolution and execution of SFC, many PLC manufactures have implemented only a reduced set of qualifiers. A survey of several PLC software systems shows that the qualifiers {N, P} are supported widely. In industry, N qualified action is often called *continuous* action and P qualified action is often called *oneshot* action. Although {N, P} are only a reduced set of the SFC qualifiers defined in IEC1131, PLC manufactures claim that they have found this subset is adequate for a lot of control strategies and applications because the functionalities of many qualifiers can be indirectly implemented using programming features and qualifiers N and P.



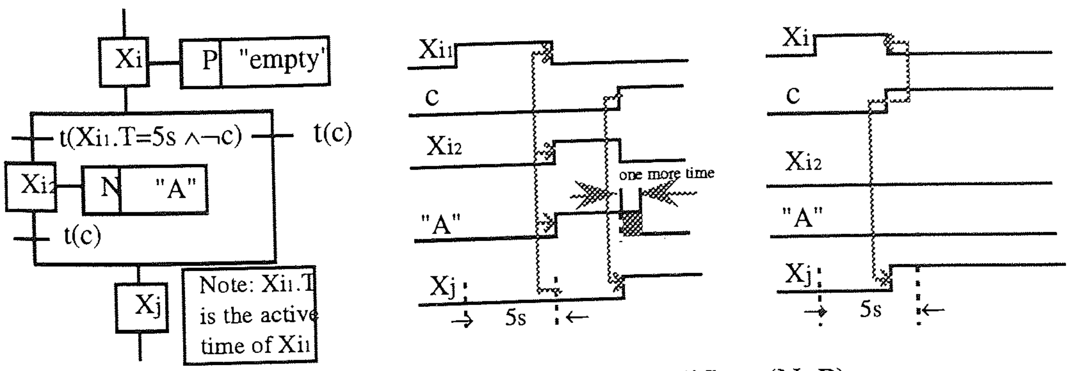
(a) Graphical representation of qualifiers (S, R)



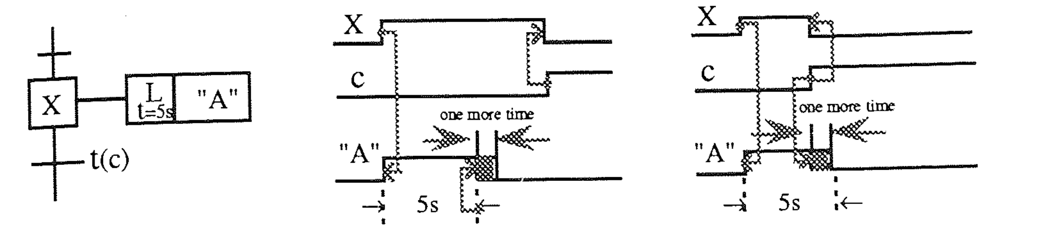
(b) Graphical representation of S and R in qualifiers (N, P)
Fig.4.7 Relationship between qualifiers (S, R) and qualifiers (N, P)



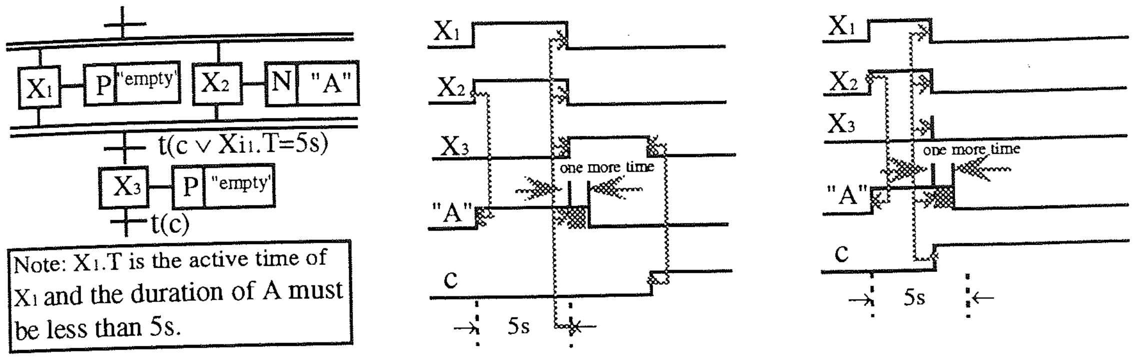
(a) D representation (b) C occurs after 5 secs. (c) C occurs before 5 secs.



(d) Graphical representation of D in qualifiers (N, P)
Fig.4.8 Relationship between qualifier D and qualifiers (N, P)



(a) L representation (b) C occurs after 5 secs. (c) C occurs before 5 secs.



(d) Graphical representation of L in qualifiers (N, P)
Fig.4.9 Relationship between qualifier L and qualifiers (N, P)

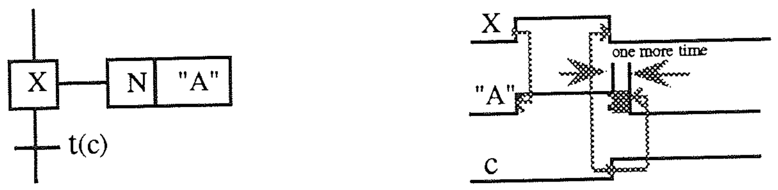
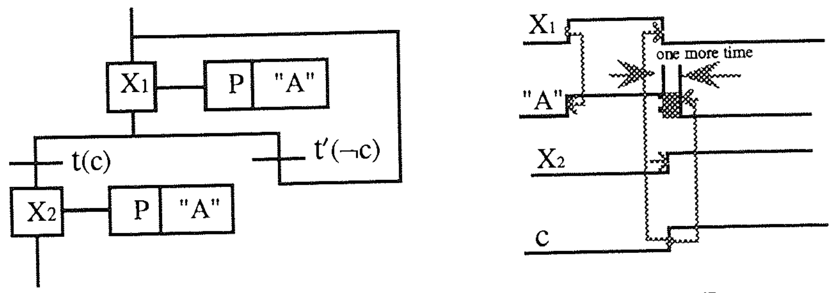


Fig.4.10 (a) N representation



(b) Graphical representation of N in qualifier P
Fig.4.10 Relationship between qualifier N and qualifier P

In this research, it has been found that all the qualifiers can be transformed into the combination of N and P. Fig.4.7 shows how the functionalities of qualifiers S and R can be implemented using qualifiers N and P; Fig.4.7 also lifts the mismatch between the evolution of SFC and the execution of action caused by S qualifier. The D qualifier with the supposed interpretations that it terminates as N can be implemented using N and P as shown in Fig.4.8. Similarly, Fig.4.9 shows that if the L qualifier is interpreted as terminating as N, then L can be implemented using N and P. Obviously, qualifiers SD, DS, and SL can be implemented using individual qualifier {S, D, L} because they are concatenation of these individual qualifiers. In this research, it is also found that the functionality of qualifier N can also be implemented using P (see Fig.4.10).

Since all qualifiers can be replaced by a logical equivalent expression that uses the P qualifier only, qualifier P can be considered as a minimum set for all qualifiers defined in IEC1131. That means, it is sufficient to consider qualifier P only in SFC verification rather than all qualifiers. From this point onwards, this thesis will consider the set of all qualifiers as being P. Qualifier P has a nice property for control system: if all the actions are qualified by P then the system situation at any moment can be explicitly defined by the set of current active steps.

4.2.5 Representation of State

To refine and analyse the execution system of SFC, let the active state of a step be further divided into two sub-states: *execution* and *completion*. Execution means that the step is in the active state and the action is being performed (executed), but the step's active state is *unavailable* to enable its successor transition. Completion means that the step is in the active state, but the action has completed its execution and the step's active state is *available* to enable its successor transition. The various phases of the state of a step are illustrated in Fig.4.11. This division is based on the fact that an action in SFC takes time to be performed (executed) and a transition is often delayed in being fired since its condition may require either the results generated by the action execution, or information coming in from the environment, or a combination of both results and environmental information. The distinction between 'execution' and 'completion' is used extensively in formal definition of the evolution rules of SFC in section 4.3.

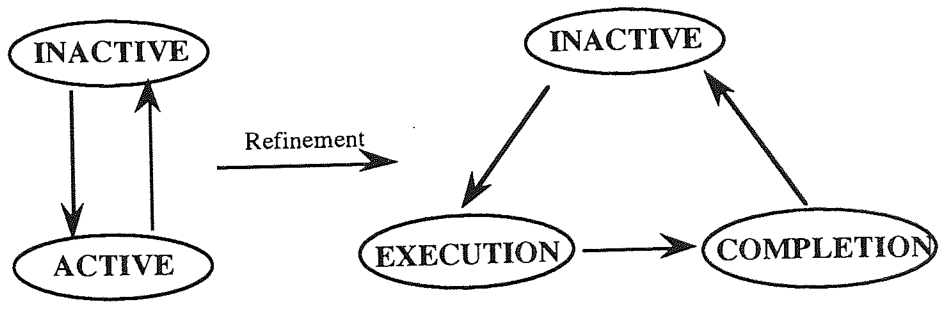


Fig.4.11 State transformation diagram of step

4.2.6 Notions of Time and Timing Analysis

Timing constraints are central to the requirements specification and design of any real-time control system. A timing constraint actually imposes a temporal restriction on system behaviour.

4.2.6.1 Representation of Time

In a discrete event system time may be represented in two ways: a time-point-based representation and a time-interval-based representation [Levi and Agrawala 90]. In a time-point-based representation the world view of a system consists of events occurring at a single moment of time, take zero time to occur, and result in the state of the system changing. In a time-interval-based representation the world view of a system consists of activities that take a finite amount of time, and have start and stop events associated with them. Although the time-point-based representation is different from the time-interval-based representation, they are complementary in many applications rather than in conflict. In fact, a description of a real time system often needs both representations. For example, the proposed hybrid system models in [Grossman *et al* 93, Langmaack *et al* 94] involve both time-point-based and time-interval-based representations.

4.2.6.2 Transition Firing Time and Step Activation Time

It is known that an SFC step represents a situation in which its associated action will be executed and a transition is a "switch" which passes control from steps to steps. Since the firing time of a transition is negligible with respect to the duration of step's activation, a transition's firing time can be considered as *point-based* and a step's activation as *interval-based.* Also if the firing time of a transition is ignored, then the state space for system analysis can be reduced.

4.2.6.3 Timing Constraints

A timing constraint is generally represented by the description of how events are related to other events. In [Chandrasekharan *et al* 85, Dasarathy 85], timing constraints are classified into three types: *Duration length* — an activity must last for *t* amount of time; *Maximum* — no more than *t* amount of time may elapse between the occurrences of two events; and *Minimum* — no less than *t* amount of time may elapse between the occurrences of two events. [In this context, an event is defined as either a stimulus to the system from its environment (input), or as an externally observable response that the system makes to its environment (output)].

Since we assume each SFC action have a fixed duration for its execution (see 4.2.3), it is very natural to describe its execution period using the duration length timing constraint.

4.3 Formal Definition of the Evolution Rules for SFC

All the evolution rules (semantics) of SFC are defined in natural language. Such informal definitions often cause ambiguous problems.

In this section, a formal definition of evolution rules for SFC will be defined using a set of formal notations based on the concepts of enabling and firing rules in Petri nets. This clear and precise definition is necessary for both industrial implementation and the formal analysis of SFC.

The approach adopted is similar to IEC848 [88], which presents a mathematical description of Grafcet based on Petri nets. In IEC848 [88] a Grafcet is defined as a quadruple G = <X, T, L, S_0>, where:

$X = (x_1, x_2, ..., x_m)$	is a finite, non empty, set of steps;
$T = (t_1, t_2, ..., t_n)$	is a finite, non empty, set of transitions;
X and T represent the nodes of the graph;	
$L = (l_1, ..., l_p)$	is a finite, non empty, set of directed links, linking either a step to a transition or a transition to a step.
$S_0 \subset X$	is the set of initial steps. These steps are activated at the beginning of the process and determine the initial state.

Moreover, the graph is interpreted, meaning that:

— with each step a command or action is associated;
— with each transition a logic transition condition is associated.

Since SFC has the same syntax as Grafcet, this description also applies to SFC.

Let SFC = < X, T, L, S_0 >. For each $t \in$ T and each $x \in$ X, the sets of input and output steps of a transition t (t_x^- and t_x^+ respectively) and the sets of input and output transitions of a step x (x_t^- and x_t^+ respectively) are denoted as:

$t_x^- = \{x \in X \mid \exists\, l \in L$, such that $l = <x, t>$ — a directed arc from x to t };
$t_x^+ = \{x \in X \mid \exists\, l \in L$, such that $l = <t, x>$ — a directed arc from t to x };
$x_t^- = \{t \in T \mid \exists\, l \in L$, such that $l = <t, x> \}$;
$x_t^+ = \{t \in T \mid \exists\, l \in L$, such that $l = <x, t> \}$;

To precisely define the evolution rules of SFC, the active state of each step needs to be divided into two sub-states (execution and completion respectively) as discussed in 4.2.5. Let $S = (S_e, S_c)$ be the state vector of an SFC where $S = S_e \cup S_c$ and $S_e \cap S_c = \varnothing$.

S is defined as:

$\forall x \in X$

$$S_e(x) = \begin{cases} 1 & \text{if step } x \text{ is in the execution state} \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

$$S_c(x) = \begin{cases} 1 & \text{if step } x \text{ is in the completion state} \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

$$S(x) = 1 \ (x \text{ is active}) \Rightarrow (S_e(x) = 1 \vee S_c(x) = 1) \wedge \neg (S_e(x) = 1 \wedge S_c(x) = 1) \tag{4.3}$$

$$S(x) = 0 \ (x \text{ is inactive}) \Rightarrow S_e(x) = 0 \wedge S_c(x) = 0 \tag{4.4}$$

The five evolution rules defined in Grafcet can be formally described for SFC as:

1) The initial state is given by the set of initial steps. That is:

$(\forall x \in S_0)\ S(x) = 1$ where S_0 is the initial active step set.

2) A transition should be fired if it is enabled and its condition is evaluated as TRUE.

Let $S = (S_e, S_c)$ be the state vector of SFC, then $\forall t \in T$, t is said to be enabled if:

i) $t_x^- \subseteq S_c$	input steps of t are all in completion states
ii) $t_x^+ \not\subset S_e$	output steps of t are not in executing states

After transition t is enabled, if the transition condition, c, is true (which depends on the system internal values and the system I/O), then it should be fired. Let C be the set of conditions, and **E** be a function defined from $T \times C$ to Boolean, where "$\mathbf{E}(t)(c)$" means evaluation of condition c associated with t. Then transition t should be fired if:

iii) $\mathbf{E}(t)(c) = \text{TRUE}$	the condition c associated with t is true

3) When a transition is fired, all the steps in its input set are deactivated and at the same point in time all the steps in its output set are activated simultaneously.

Let the $S' = (S'_e, S'_c)$ be the new state derived from state $S = (S_e, S_c)$, then after transition t is fired under S, S′ is computed as follows:

$\forall x \in X$:

$$S'_e(x) = \begin{cases} 1 & x \in t^+_x \\ S_e(x) & \text{otherwise} \end{cases} \tag{4.5}$$

$$S'_c(x) = \begin{cases} 0 & x \in t^-_x \\ 0 & x \in t^+_x \wedge S_c(x) = 1 \\ S_c(x) & \text{otherwise} \end{cases} \tag{4.6}$$

4) Several transitions shall be fired simultaneously if they can be fired simultaneously.

Let the $S' = (S'_e, S'_c)$ be the next state derived from state $S = (S_e, S_c)$.

If $\exists\, U \subset T$ and

I) $\forall t \in U$,

i) $t^-_x \subseteq S_c$

and ii) $t^+_x \not\subset S_e$

and iii) $\mathbf{E}(t)(c) = \text{TRUE}$

II) $\forall t_1, t_2 \in U$,

$(t1^-_x \cup t1^+_x) \cap (t2^-_x \cup t2^+_x) = \varnothing$

then after all the transitions in U are simultaneously fired under S, S′ is defined as:

$\forall x \in X$:

$$S'_e(x) = \begin{cases} 1 & x \in t^+_x, \text{ where } t \in U \\ S_e(x) & \text{otherwise} \end{cases} \tag{4.7}$$

$$S'_c(x) = \begin{cases} 0 & x \in t^-_x, \text{where } t \in U \\ 0 & x \in t^+_x \wedge S_c(x) = 1, \text{ where } t \in U \\ S_c(x) & \text{otherwise} \end{cases} \tag{4.8}$$

5) If a step must be activated and deactivated simultaneously, then it remains active. (see 2) and 3) defined above).

By examining the refined evolution rules above, it can be seen that reactivation is allowed in SFC evolution but not in SFC execution. That is, a transition t can fire only when its immediate successor step is in "inactive" or "completion" state. Formally, t fires iff:

$$t^-_x \subseteq S_c \wedge t^+_x \not\subset S_e \wedge \mathbf{E}(t)(c) = \text{TRUE} \tag{4.9}$$

This restricts reactivation to steps who's actions are complete. This is some advantage because reactivation in execution is not desirable in many practical applications as it may describe a potentially "unsafe" situation.

Consider for example the problems which would arise if reactivation in execution were allowed. In the simple case shown in Fig.4.12, if event x occurs (which represents the value change of the condition associated with the transition) at time τ, then the action a associated with step X will finish its execution at time $\tau+\tau(a)$ where $\tau(a)$ is the time needed by the computer system to respond to the event x. If reactivation in SFC execution system is allowed, then following event-action sequence is feasible:

If event x occurs at times τ_1, τ_2, ...,
then the associated action will complete execution at times

$$\tau_1+\tau(a),\ \tau_2+\tau(a),\ ...... \qquad (4.10)$$

When $\tau_2 < \tau_1+\tau(a)$ (i.e, the second occurrence of event is before the first action execution is complete), if the computer system is insensitive or blind to a new event occurrence while it is still performing an action, then the reactivation in execution will not be satisfied. To respond correctly and completely to each reactivation, the following restriction

$$\tau_2 \geq \tau_1+\tau(a),\ ......\ \tau_{i+1} \geq \tau_i+\tau(a) \qquad (4.11)$$

must be imposed on times at which event can occur.

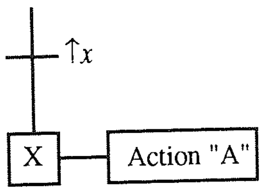


Fig.4.12 A simple SFC for reactivation analysis

However, in SFC [IEC 93], it is known that the occurrence of an event associated with a transition depends only on the state of transition's predecessor step(s) rather than its successor step(s). That is, there is no relationship between τ_1 and $\tau(a)$. This means there is no way of imposing the constraint (4.11) and for a deterministic system the reactivation situation should be excluded. Specifically, this restriction should be applied to both SFC defined in [IEC 93] and the revised form of SFC defined formally in this section.

Interestingly, even if the computer system can be interrupted when an event occurs and a new process for the action execution is created (which will be executed after the interruption), then reactivation may still cause problems since the total number of processes under execution may grow unbounded or exceed a limit. For this reason, all SFCs to be designed should avoid reactivation situation.

4.4 An Extended Petri Net Model for SFC

4.4.1 Introduction

Given a system design, an obvious question is how the logical and temporal properties of the system can be formally analysed? It is well known that the fundamental problem in the analysis of any system is to find a theory that allows the system to be modelled and manipulated in a formal analysis, that is, a mathematical representation of the system and an analysis technique based on the mathematical representation. The approach adopted in this thesis for SFC analysis is to transform SFC onto the framework of a time related Petri net model about which we already know how to do the analysis, rather than to establish an analysis technique for SFC. Following are the important reasons of using Petri nets as the analysis model of SFC:

i) there are significant similarities between Petri nets and SFC, particular SFC without reactivation;
ii) Petri nets provide an elegant means to model and analyse parallel activities and synchronisation;
iii) the extensions to Petri net theory give the designer the ability to verify timing properties of the system.

For example, an SFC model can be interpreted using a Petri net model in the following aspects:

Structural aspect:
1) the steps defined in an SFC can be substituted by the places of a Petri net;
2) the transitions defined in an SFC can be substituted by the transitions of a Petri net;
3) the directed links defined in an SFC can be substituted by the arcs defined in a Petri net;
4) the actions associated with steps in an SFC can be substituted by associating time delays with places in a Petri net such as TPTN [Sifakis 80].

Behavioural aspect:
1) the state defined in an SFC can be interpreted by the marking defined in a Petri net. An active step in SFC can be interpreted as a place with a token and the state transformation in SFC can be described by the marking change in Petri nets;

2) the *enabled* and *firing* rules defined in SFC, can be interpreted by the *enabling* and *firing* rules defined in Petri nets;
3) the evolution rule for simultaneous activations of several steps after firing one transition in SFC can be interpreted by distributing a token into different places simultaneously after a transition is fired in Petri nets;
4) the evolution rule for the simultaneous firing of several enabled transitions in SFC can be interpreted by the non-sequential (or partial order) behaviour extracted from the firing sequence of a Petri net.

Nevertheless there are some differences between SFC and Petri nets. For example, an SFC cannot model non-deterministic choice and an SFC is interpreted because it includes actions and conditions. However, these differences do no prevent interpreting SFC behaviour in terms of Petri nets, particular TPTN [Sifakis 80], because an SFC without reactivation and its corresponding Petri net model will generate the same state sequences [Aygaline and Denat 93]. For example, an SFC without reactivation can be interpreted using the condition/event net concept and the potential concurrent behaviour or simultaneous firing in SFC can be interpreted using the partial order description of the net behaviour.

4.4.2 An Extended Timed Place Transition Net (TPTN) Model

Although the TPTN model of Sifakis [80] can model SFC in many aspects such as steps, transitions, and actions, it cannot be used to model the conditions or guards associated with transitions and their effects on transition firings because TPTN lacks the means to model external inputs. In fact, it is not difficult to see that all time-related Petri net models reviewed in Chapter 2 cannot precisely model SFC conditions and their effect on transitions because these models are uninterpreted and lack elements to describe external inputs and outputs. This means that some extension needs to be considered in applying Petri net models to SFC. In this section, an extended time-related Petri net model based on TPTN is given and the transformation method from SFC to the extended model will be defined in next section.

4.4.2.1 Sifakis TPTN Model

Definition: An TPTN model is defined as a triplet [Sifakis 80]

$$N_t = (PN, \Upsilon, \upsilon) \qquad (4.12)$$

where

- $PN = (P, T, F, M_0)$ is a Petri net as defined by Equation (2.3) in Chapter 2;
- Υ is a totally ordered set by a relation $\geq$, we call <u>instants</u> the elements of Υ
- υ is a mapping of $P \times \Upsilon$ into Υ, called <u>time base</u> of N_t, where P is the set of places in PN, such that $\forall(p, \tau_i) \in P \times \Upsilon, \upsilon(p, \tau_i) \geq \tau_i$.

TPTN is defined by associating a delay $\upsilon(p, \tau)$ with each place $p \in P$. In TPTN, each token in a place may be in one of the following two states: *available* or *unavailable*. After a transition is fired at instant τ, the token that arrives at a place p is unavailable during the interval $<\tau, \upsilon(p, \tau)>$ and then it becomes available. At any instant τ, the marking M of a TPTN is the sum of two markings M_a and M_u, where M_a is the marking constituted of all the available tokens of M and M_u is the marking constituted of all the unavailable tokens of M. Transitions can only be enabled by available tokens. Transition firing is instantaneous in TPTN. For a TPTN model with constant unavailability times associated with each place, the mapping υ is defined as:

$$\forall p_i \in P,\ \forall \tau \in \Upsilon: \upsilon(p_i, \tau) - \tau = z_i \qquad (4.13)$$

That is, a token is delayed in place p_i by z_i time units.

4.4.2.2 Extended TPTN Model — XTPTN

In TPTN, Υ can take on non-negative real numbers. To simplify the analysis, the extended model in this thesis assumes that Υ only takes non-negative integers. Also the extended model only considers constant unavailability times associating with each place.

Definition: An extended timed place transition net is defined as

$$\text{XTPTN} = (PN, \Upsilon, \upsilon, \mathfrak{I}, \mathfrak{C}, \Psi, \text{TURT}^{\tau_0}) \qquad (4.14)$$

where

- (PN, Υ, υ) is a TPTN net as defined by Equation (4.12) and $PN = (P, T, F, M_0)$ is a Petri net as defined by Equation (2.3) in Chapter 2.
- $M_0 = M_{a0} \cup M_{u0}$, M_{a0} is the marking constituted of all the available tokens initially, and M_{u0} is the marking constituted of all the unavailable tokens initially.
- $\mathfrak{I}$ is a finite set of *interface places* with capacity $K(p) \leq 1$, where $p \in \mathfrak{I}$, and satisfies following requirement:
  $\forall p \in \mathfrak{I}, \forall \tau \in \Upsilon: \upsilon(p, \tau) - \tau = 0$ (i.e., unavailability time associated with p is zero);
- $\mathfrak{C}$ is a set of directed arcs associating interface places with transitions, $\mathfrak{C} \subseteq \mathfrak{I} \times T$.
- Ψ is a function which assigns a pair of non-negative integers of so called *firable interval* to each transition of the net, $\Psi: T \rightarrow \mathbf{I}^{+0} \times (\mathbf{I}^{+0} \cup \infty)$, where $\mathbf{I}^{+0}$ is a set of non-negative integers;
- TURT^{τ_0} is the vector of *token unavailable remaining time* for unavailable tokens at initial instant, where τ_0 is the initial instant. That is,
  $\forall p \in M_{u0}: 0 < \text{TURT}^{\tau_0}(p) \leq \upsilon(p, \tau_0) - \tau_0$

TURT is a function which assigns a time unit to each place in P at a given moment. That is,

$$\forall p_i \in P, \forall \tau \in \Upsilon: 0 \leq \mathrm{TURT}^{\tau}(p_i) \leq \upsilon(p_i, \tau) - \tau \tag{4.15}$$

It follows that XTPTN is a TPTN with the following extensions:

(1) Places

A set of interface places $\mathfrak{I}$ is added into TPTN and the unavailability time associated with each place $p \in \mathfrak{I}$ is zero. Since the unavailability time associated with each interface place in $\mathfrak{I}$ is defined as zero, we have:

$$\forall p \in \mathfrak{I}, \forall \tau \in \Upsilon: \mathrm{TURT}^{\tau}(p) = 0; \tag{4.16}$$

Places in $\mathfrak{I}$ can be used to model a guard associated with each transition. For example, $\forall p \in \mathfrak{I}$, if p is tokenised, then the evaluation of the guard can be interpreted as TRUE; otherwise as FALSE. If a transition in XTPTN has no place in $\mathfrak{I}$ as its input place, then it can be interpreted as that the transition in SFC is associated with an always TRUE guard.

(2) Transition

In XTPTN, each transition is associated with a pair of time units α and β, $0 \leq \alpha \leq \beta$, where

- α $(0 \leq \alpha)$, is the minimum time that must elapse, starting from the moment at which the transition is enabled, until this transition can fire;
- β $(\alpha \leq \beta \leq \infty)$, is the maximum time during which the transition can be enabled without being fired.

Thus, the meaning of this pair of numbers is the same as that defined in the time Petri nets [Merlin and Farber 76].

(3) States

To describe the token unavailable remaining time (TURT), the idea of RFT (remaining firing time) from [Holiday and Venon 87] is adopted for XTPTN. TURT(p) is used to describe the period from the point a token is deposited into a place p until it becomes available. After a token is deposited into a place p TURT(p) is set to the delay associated with p and then starts to decrease to zero. Since some tokens can remain unavailable in places while a marking change occurs, a state is characterised by both marking and the vector of TURT.

(4) Marking

A marking in XTPTN is composed of two parts, M_a (marking of available tokens) and M_u (marking of unavailable tokens) such that $M = M_a \cup M_u$. Corresponding to (4.16) we have:

$$\forall M \in R(M_0),\ M = M_a \cup M_u: \neg\exists p \in \mathfrak{I}\ [p \in M_u] \qquad (4.17)$$

(5) Enabling and Firing Rule

In TPTN, for a transition $t \in T$ it was not clearly specified whether its input and output places can contain unavailable tokens when t is enabled and fired. In XTPTN, a transition t is said to be enabled iff:

i) For each of its input place p, all tokens in p are *available*;

ii) For each of its output places p, either no token exists in p or all the tokens in p are *available*;

Let $(\bullet t) = \{\, p \mid p \in P: \langle p, t\rangle \in F \cup \mathfrak{C} \,\}$ (the input places of transition t);

$(t\bullet) = \{\, p \mid p \in P: \langle t, p\rangle \in F \,\}$ (the output places of transition t);

$M^\tau = M_a^\tau \cup M_u^\tau$ be a marking of XTPTN at time τ. Then for $\forall t \in T$, t is said to be *enabled* at τ iff:

i) $(\bullet t) \subseteq M_a^\tau \wedge (\bullet t) \cap M_u^\tau = \varnothing$ (4.18)

ii) $(t\bullet) \cap M_u^\tau = \varnothing$ (4.19)

After a transition t is enabled, say at time τ, transition t is said to be firable iff it is continuously enabled from τ to $\tau+\alpha$, where $\Psi(t) = \langle\alpha, \beta\rangle$. In XTPTN, it is assumed that the transition must fire between time $\tau+\alpha$ and $\tau+\beta$, unless it is disabled by the firing of another conflicting transition. Firing a transition is instantaneous.

(6) State Change

After a transition is fired, one available token will be removed from each of its input places and at the same time one token will be deposited into each of its output places. For each output place p, TURT(p) will be set as the delay associated with p.

Suppose a transition t becomes enabled at time τ under state $S^\tau = (M^\tau, TURT^\tau)$, where $M^\tau = (M_a^\tau, M_u^\tau)$, and is fired at time $\tau+\vartheta$, where $\Psi(t) = \langle\alpha, \beta\rangle$ and $\alpha \le \vartheta \le \beta$. Then the new state $S^{\tau+\vartheta} = (M^{\tau+\vartheta}, TURT^{\tau+\vartheta})$, where $M^{\tau+\vartheta} = (M_a^{\tau+\vartheta}, M_u^{\tau+\vartheta})$, reached from $S^\tau = (M^\tau, TURT^\tau)$ is defined as:

$\forall p \in P$:

$$TURT^{\tau+\vartheta}(p) = \begin{cases} \upsilon(p, \tau+\vartheta,) - (\tau+\vartheta) & \text{if } p \in (t\bullet) \text{ (set delay associated with } p \text{ to TURT)} \\ 0 \quad \text{if } 0 < TURT^\tau(p) \le \vartheta & \text{(unavailable at } \tau \text{ but available at } \tau+\vartheta) \\ TURT^\tau(p) - \vartheta & \text{if } \vartheta < TURT^\tau(p) \end{cases} \qquad (4.20)$$

$$M^{\tau+\vartheta}(p) = \begin{cases} M^{\tau}(p) - 1 & \text{if } p \in (\bullet t) - (t \bullet) \\ M^{\tau}(p) + 1 & \text{if } p \in (t \bullet) - (\bullet t) \\ M^{\tau}(p) & \text{otherwise} \end{cases} \tag{4.21}$$

$$M_a^{\tau+\vartheta}(p) = \begin{cases} M_a^{\tau}(p) + 1 & \text{if } 0 < \mathrm{TURT}^{\tau}(p) \leq \vartheta \\ & \text{or } p \in (t \bullet) \wedge \mathrm{TURT}^{\tau+\vartheta}(p) = 0 \\ M_a^{\tau}(p) - 1 & \text{if } p \in (\bullet t) - (t \bullet) \\ & \text{or } p \in (\bullet t) \cap (t \bullet) \wedge \mathrm{TURT}^{\tau+\vartheta}(p) \neq 0 \\ M_a^{\tau}(p) & \text{otherwise} \end{cases} \tag{4.22}$$

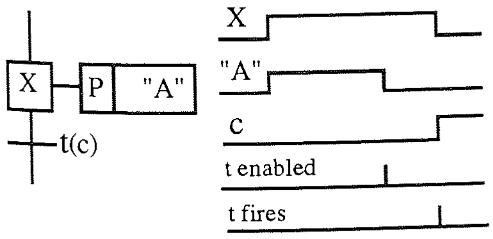
$$M_u^{\tau+\vartheta}(p) = \begin{cases} 1 & \text{if } \mathrm{TURT}^{\tau+\vartheta}(p) \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.23}$$

4.5 Modelling SFC by XTPTN

Since it is assumed that each action has a fixed duration for its execution and that the condition associated with an SFC transition was enforced to be true within a period after the transition is enabled in SFC, it seems reasonable to propose that the execution state of an SFC step can be modelled by a place with a delay and the condition can be modelled by an interface place in XTPTN. Also the timing property that all enabled SFC transitions should *immediately* be fired after their associated conditions become true can be modelled by defining the function Ψ of XTPTN as: $\forall t \in T$, $\Psi(t) = <0,0>$, where T is the set of transitions of XTPTN.

The discussion presented in Section 4.2.4 of Chapter 4 shows that it is sufficient to consider the qualifier P only for SFC verification because all qualifiers defined in SFC can be replaced by a logical equivalent expression that uses the P qualifier.

Fig.4.13 shows how SFC with qualifier P can be modelled using XTPTN, and tables 4.2 and 4.3 show translations of all SFC structural constructs defined in IEC1131 in terms of XTPTN. The next chapter considers the formal analysis of the translated constructs.



StepX $\xrightarrow{\text{TRAN}}$ place p
Duration (A) $\xrightarrow{\text{TRAN}}$ $\upsilon(p, \tau) - \tau = z_p$
Transition t $\xrightarrow{\text{TRAN}}$ transition t
Condition c $\xrightarrow{\text{TRAN}}$ interface place p_c
Immediate firing $\xrightarrow{\text{TRAN}}$ $\psi(t) = <0, 0>$

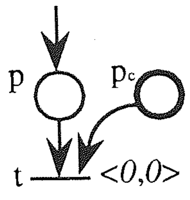


Fig.4.13 XTPTN model of SFC

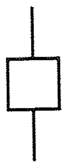
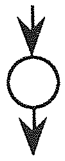
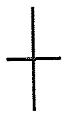
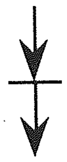
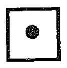
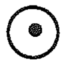
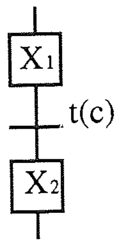
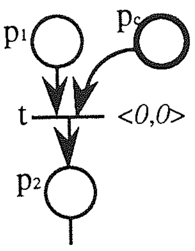
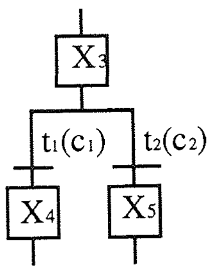
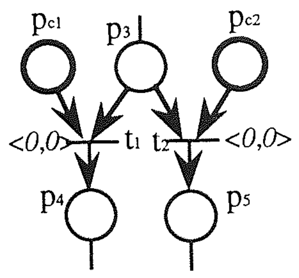
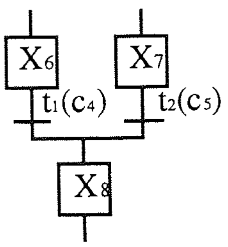
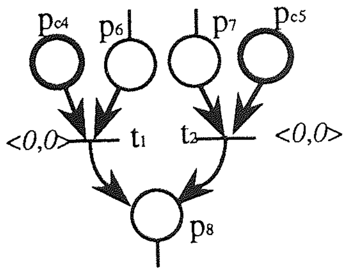
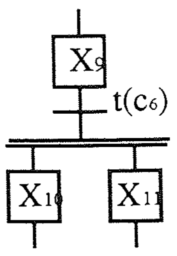
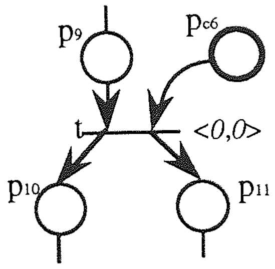
No	SFC	SFC Interpretation	XTPTN
1		Step	
2		Transition	
3		Active	
4		Single sequence An evolution from step X_1 to step X_2 shall take place iff X_1 is active and c is true.	
5		Divergence of sequence selection The designer must assure that conditions c_1 and c_2 are mutually exclusive.	 p_{c1} p_3 p_{c2} $<0,0>$ t_1 t_2 $<0,0>$ p_4 p_5
6		Convergence of sequence selection An evolution shall take place from X_6 to X_8 iff X_6 is active and the c_4 is true or from X_7 to X_8 iff X_7 is active and c_5 is true	 p_{c4} p_6 p_7 p_{c5} $<0,0>$ t_1 t_2 $<0,0>$ p_8
7		Simultaneous sequence-divergence An evolution shall take place from X_9 to X_{10} and X_{11}, iff X_9 is active and c_6 is true.	

Table 4.2 XTPTN representation of SFC constructs

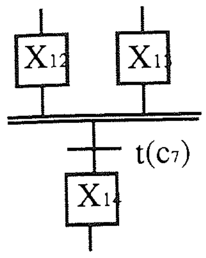
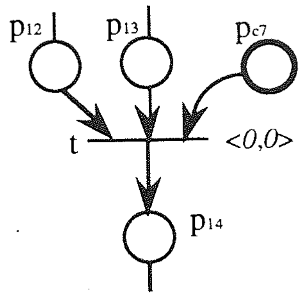
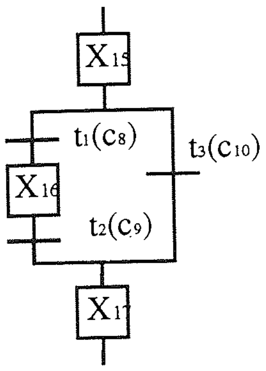
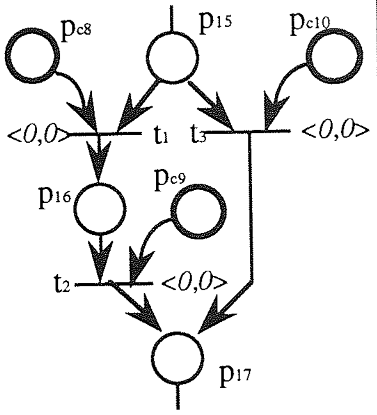
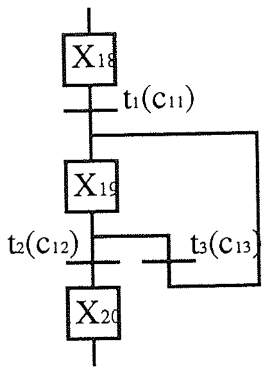
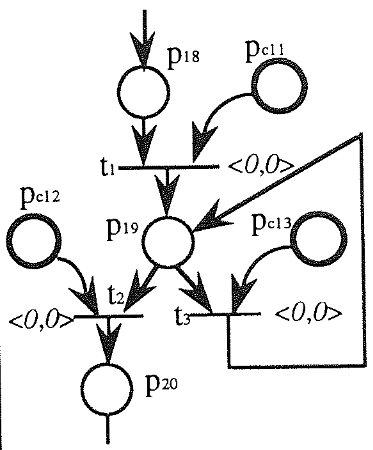
No	SFC	SFC Interpretation	XTPTN
8		Simultaneous sequence-convergence An evolution shall take place from X_{12}, X_{13} to X_{14} iff all steps above the double line are active and the c_7 is true.	
9		Sequence skip A "sequence skip" is a special case of sequence selection in which one or more of the branches have no steps.	 pc8 p15 pc10 <0,0> t1 t3 <0,0> p16 pc9 t2 <0,0> p17
10		See divergence of sequence selection	 p18 pc11 t1 <0,0> pc12 p19 pc13 t2 <0,0> t3 <0,0> p20

Table 4.3 XTPTN representations of SFC constructs

Summary

SFC has been presented and analysed in this chapter. After formally analysing SFC from different aspects, a formal definition of a unified set of evolution rules for SFC has been presented. This formal definition is necessary for both SFC implementation and its analysis. To accommodate the subtle nature of timing relationships in SFC, it was necessary to extend existing time-related Petri net models. An extended form of TPTN has been derived which allows a straightforward translation of SFC model while being analysable by existing Petri net techniques (see next chapter).

Chapter 5 Verification of SFC Designs

5.1 Preliminaries of XTPTN

Let N = (PN, Υ, υ, ι, $\mathcal{C}$, Ψ, $TURT_0$) be a XTPTN, where PN = (P, T, F, M_0) is a Petri net, $M_0 = M_{a0} \cup M_{u0}$ is the initial marking and $TURT_0$ is the vector of token unavailable remaining time for unavailable tokens at the initial instant. As discussed, a system state is characterised by both marking and vector TURT; thus the initial state is represented by S_0 = (M_0, $TURT_0$).

If $\exists t \in T\ [(\bullet t) \subseteq M_{a0} \wedge (\bullet t) \cap M_{u0} = \varnothing \wedge (t\bullet) \cap M_{u0} = \varnothing]$, then t is enabled. After t is continuously enabled for α but before β time units (where $\Psi(t) = <\alpha, \beta>$), t is said to be firable and it must be fired during this interval unless it is disabled by some other transition firing. On firing transition t, state S_0 is transformed into a new state S′ which consists of a new marking and a new vector TURT.

Let S [->> S′ $\Leftrightarrow \exists t \in$ T: S [t >> S′ e.g. state S′ is reached from state S by firing t. We can now define the reachable state class associated with the initial state S_0.

Definition
Let S_0 be the initial state of a given XTPTN, then the reachable state class of the XTPTN defined by S_0, denoted by [S_0>>, is given by:

i) $S_0 \in$ [S_0>> (5.1)
ii) If S′$\in$ [S_0>> and S′ [->> S″ then S″ $\in$ [S_0>> (5.2)

Since each transition t can be fired during an interval specified by $\Psi(t) = <\alpha, \beta>$ and the firing is instantaneous, a state's transformation from S to S′ by firing transition t, denoted by S[t>>S′, has following property:

Firing t at a different time point might lead to pre-firing and post-firing states S and S′ with different TURT vectors if TURT is not zero before firing. The markings associated with S and S′ will be unchanged. That is, for each marking M there exists a set of states which are all associated with M but have different TURT.

The set of states associated with the same marking can be considered as an equivalent class and the marking as an invariant of the equivalent class. Marking changes of XTPTN can be described using the reachability technique (see page 31) defined for conventional Petri nets. Let $M [\text{->} M' \Leftrightarrow \exists t \in T: M [t > M'$ e.g. marking M′ is reached from marking M by firing *t*. Similar to the definition of relation [>> for states above, we can now define the reachable marking class associated with the initial marking M_0.

Definition
Let M_0 be the initial marking of a given XTPTN, then the reachable marking class of the XTPTN defined by M_0, denoted by $[M_0 >$, is given by:

i) $M_0 \in [M_0>$ (5.3)
ii) If $M' \in [M_0>$ and $M' [\text{->} M''$ then $M'' \in [M_0>$ (5.4)

Definition
Let $N = (PN, \Upsilon, \upsilon, \imath, \mathfrak{C}, \Psi, TURT_0)$ be a XTPTN, where $PN = (P, T, F, M_0)$. Then a sequence of markings $\Gamma = M_0 \ldots M_i M_{i+1} \ldots$ is a member of the relation $[M_0>^*$ (which is the transitive closure of relation $[M_0>$) such that Γ starts with M_0 and for any two adjacent markings M_i and M_{i+1} in Γ, there is a transition $t \in T$ such that $M_i[t>M_{i+1}$.

An overview of all markings of a given XTPTN net can be obtained by constructing a reachability graph [Peterson 81, Murata 89] which represents the elementary changes of markings. A reachability graph, where the graph node is labelled by a marking and the arc is labelled by a transition, is defined as:

Definition
Let $N = (PN, \Upsilon, \upsilon, \imath, \mathfrak{C}, \Psi, TURT_0)$ be a XTPTN, where $PN = (P, T, F, M_0)$, its reachability graph is a labelled directed graph $G = (N, E, \mathcal{F}, \mathcal{H})$ here

i) N is a non-empty finite set of nodes;
ii) $E \subseteq N \times N$;
iii) $\mathcal{F}$: $N \rightarrow \mathbf{I}^{+0|P|}$ is the labelling function from node to marking,
where $\mathbf{I}^{+0}$ is the set of non-negative integers;
iv) $\mathcal{H}$: $E \rightarrow T$ is the labelling function from arc to transition.

The reachability graph is a fundamental tool for studying the behaviour in any Petri net. When a Petri net is bounded, the reachability graph contains all possible reachable markings. The disadvantage is that this is an exhaustive method. When a Petri net is unbounded, the graph representation will become infinitely large. To keep the graph finite, a special symbol ω is introduced to deal with the infinite number of tokens. It has the following properties for integer n,

$$\omega > n, \quad \omega + n = \omega, \quad \omega - n = \omega \quad \text{and} \quad \omega >= \omega \tag{5.5}$$

The following properties can be studied by using the reachability graph:

1) A given N = (PN, Υ, υ, ι, $\mathcal{C}$, Ψ, TURT$_0$), where PN = (P, T, F, M$_0$), is *bounded* and thus G(M$_0$) is finite iff ω does not appear in any node labelled marking in G.
2) A given N = (PN, Υ, υ, ι, $\mathcal{C}$, Ψ, TURT$_0$), where PN = (P, T, F, M$_0$), is *safe* iff only 0's and 1's appear in node labelled markings in G.
3) A transition is *dead* iff it does not appear as an arc label in G.

Definition
Let G = (N, E, $\mathcal{F}$, $\mathcal{H}$) be a reachability graph and $n_0e_1n_1e_2n_2...e_nn_n$ be a path starting from root n_0 of G such that:

$$\mathcal{F}(n_0)[\mathcal{H}(e_1)>\mathcal{F}(n_1),\ \mathcal{F}(n_1)[\mathcal{H}(e_2)>\mathcal{F}(n_2),\ \dots\ \mathcal{F}(n_{n-1})[\mathcal{H}(e_n)>\mathcal{F}(n_n),$$

then $\mathcal{H}(e_1)\mathcal{H}(e_2)\mathcal{H}(e_3)\dots\mathcal{H}(e_n)$ is called a *firing sequence* and can be written as

$$\mathcal{F}(n_0)[\mathcal{H}(e_1)\mathcal{H}(e_2)\mathcal{H}(e_3)\dots\mathcal{H}(e_n)>\mathcal{F}(n_n), \text{ where } \mathcal{H}(e_1)\mathcal{H}(e_2)\mathcal{H}(e_3)\dots\mathcal{H}(e_n) \in T^*$$

and T^* denotes the finite sequences over T.

Definition *Potential concurrent places*
Let N = (PN, Υ, υ, ι, $\mathcal{C}$, Ψ, TURT$_0$) be a XTPTN, where PN = (P, T, F, M$_0$). Then $\forall p_1, p_2 \in$ P: p_1 and p_2 are said to be potential concurrent, denoted by p_1 © p_2 iff

$$\exists\, M \in [M_0>: M(p_1) \neq 0 \wedge M(p_2) \neq 0 \tag{5.6}$$

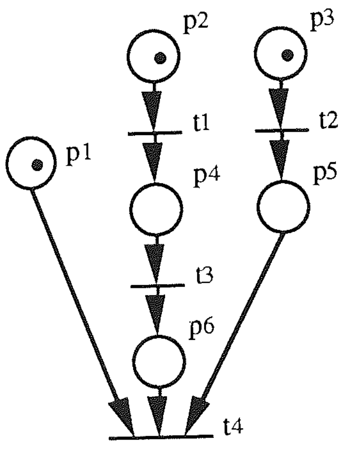
Definition *Concurrent set of place*
Let N = (PN, Υ, υ, ι, $\mathcal{C}$, Ψ, TURT$_0$) be a XTPTN, where PN = (P, T, F, M$_0$). The concurrent set of $p \in$ P, denoted by CS(p), is the set of all places that are potentially concurrent with it. That is, CS(p) = {p' | p © p'}

Similarly, the concept of potential concurrent transitions can be defined. The following is the definition of potential concurrent transition for a safe XTPTN (see page 31).

Definition *Potential concurrent transitions*
Let N = (PN, Υ, υ, ι, $\mathcal{C}$, Ψ, TURT$_0$) be a XTPTN, where PN = (P, T, F, M$_0$). Then $\forall\, t_1, t_2 \in$ T: t_1 and t_2 are said to be potentially concurrent, denoted by $t_1 \parallel t_2$ iff

i) $\bullet t_1 \cap \bullet t_2 = \varnothing$
ii) $t_1\bullet \cap t_2\bullet = \varnothing$
iii) $\exists\, M \in [M_0>$: M enables both t_1 and t_2

Concurrent places and concurrent transitions have a symmetric but not transitive nature. Let us consider the XTPTN shown in Fig.5.1(a) in which $t_1 \| t_2$, and $t_2 \| t_3$, but t_1 is not concurrent with t_3. Similarly, p_2 © p_3 and p_3 © p_4,(see markings M_0 and M_1 on Fig.5.1(b)) but p_2 is not concurrent with p_4. It is worth pointing out that the concurrent transition concept is different from the concurrent place concept. Let us look at the reachability graph shown in Fig.5.1(b) again in which $CS(p_1) = \{p_2, p_3, p_4, p_5, p_6\}$, but the transition t_4 which is related to p_1 is not concurrent with any other transitions.



$$\begin{tikzcd}
 & & n_0 = M_0(1,1,1,0,0,0) \arrow[dl, "e_1 = t_1"'] \arrow[dr, "e_2 = t_2"] & \\
 & n_1 = M_1(1,0,1,1,0,0) \arrow[dl, "e_3 = t_3"'] \arrow[dr, "e_4 = t_2"] & & n_2 = M_2(1,1,0,0,1,0) \arrow[dl, "e_5 = t_1"] \\
n_3 = M_3(1,0,1,0,0,1) \arrow[dr, "e_6 = t_2"'] & & n_4 = M_4(1,0,0,1,1,0) \arrow[dl, "e_7 = t_3"] & \\
 & n_5 = M_5(1,0,0,0,1,1) \arrow[d, "e_8 = t_4"] & & \\
 & n_6 = M_6(0,0,0,0,0,0) & &
\end{tikzcd}$$

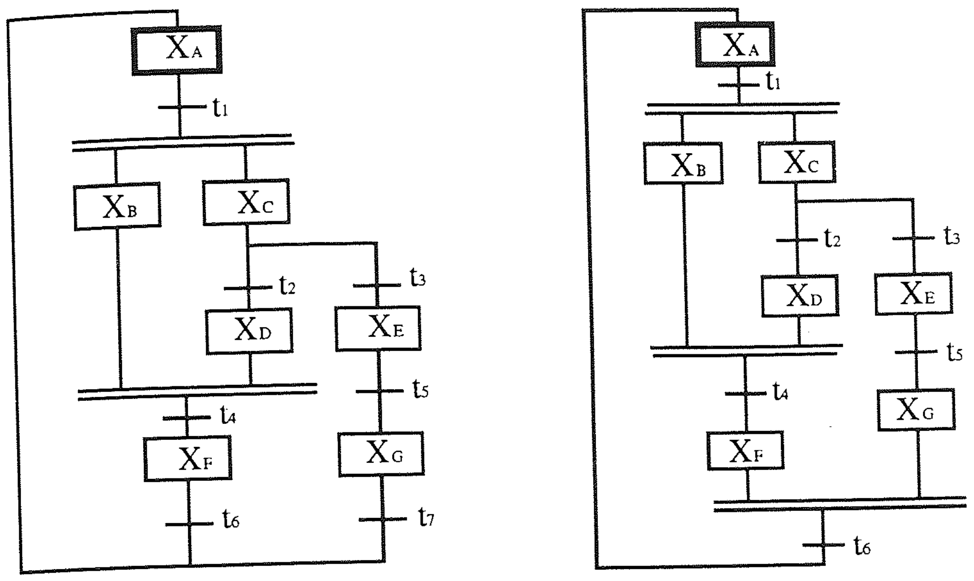
(a) An XTPTN (b) Reachability graph of (a)

Fig.5.1 An XTPTN for illustrating concurrent places and transitions

5.2 SFC State Analysis — Rechability-Based Analysis

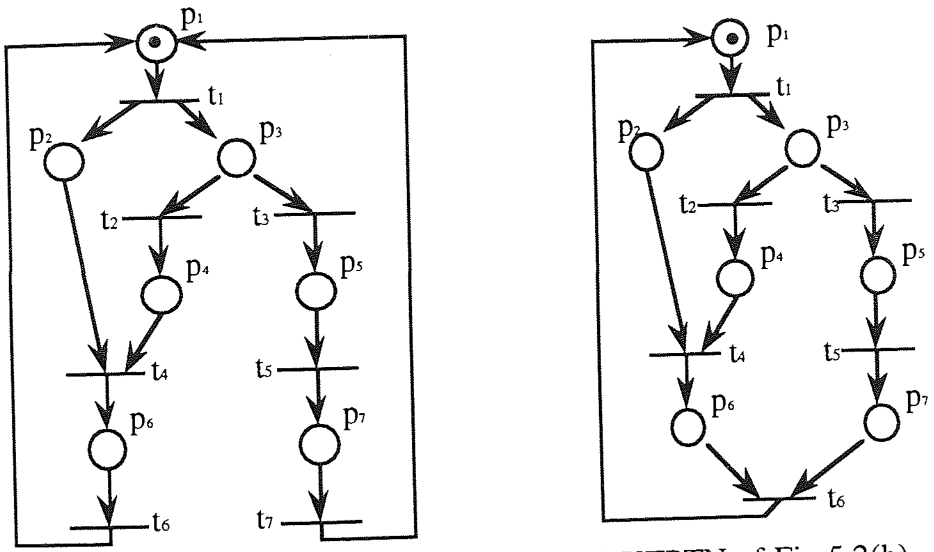
The initial marking (or system inputs) can be used to generate the reachability graph (i.e. state transformation sequences). Reachability graph analysis is very important to investigate the system properties. For example, if a marking in the reachability graph of a XTPTN model representing an SFC is unsafe (or unbounded) in terms of Petri nets (see page 31), then the SFC system may exhibit a reactivation or an uncontrolled proliferation of states in SFC. Also if a XTPTN model of a SFC is not live in terms of Petri nets (see page 31), then the SFC system will exhibit deadlock or an unreachable step due to control starvation. In IEC1131 it is stated that these behaviours shall be treated as errors during SFC preparation. Petri net theory provides the analysis method to identify these problems. From the reachability graph, it is easy to check whether these errors exist.

The nets shown in Fig.5.2(a) and Fig.5.2(b) are SFCs given in IEC1131 [IEC 93] as examples of structurally correct nets with problematic behaviours; the corresponding XTPTN's are shown in Fig.5.3(a) and Fig.5.3(b).



a) an "unsafe" SFC b) An "unreachable" SFC

Fig.5.2 'unsafe' and 'unreachable' SFCs



a) XTPTN of Fig.5.2(a) b) XTPTN of Fig.5.2(b)

Fig.5.3 XTPTN representations of SFCs shown in Fig.5.2

The reachability graphs of Fig.5.3(a) and Fig.5.3(b) are shown in Fig.5.4 and Fig.5.5, respectively. In Fig.5.4 ω, meaning state with potential for repeated and potentially infinite instantiation, appears in some nodes' labels for place p_2. This means that transformed net (Fig.5.3(a)) is "unsafe". This implies that step X_B in the SFC represented in Fig.5.2(a) may be a reactivated or unsafe step. Reactivation or unsafe nets are not suitable for use in SFC applications because it is difficult to guarantee the system to respond correctly and completely to each reactivation. (Reactivation of SFC has been discussed formally in Section 4.3 of Chapter 4). Hence, a design objective is to ensure that the XTPTN of an SFC is safe.

$$\begin{tikzcd}[column sep=small]
 & & n_0 = M_0(1\,0\,0\,0\,0\,0\,0) \arrow[d, "e_1 = t_1"] & & \\
 & & n_1 = M_1(0\,1\,1\,0\,0\,0\,0) \arrow[dl, "e_2 = t_2"'] \arrow[dr, "e_5 = t_3"] & & \\
 & n_2 = M_2(0\,1\,0\,1\,0\,0\,0) \arrow[d, "e_3 = t_4"] & & n_5 = M_5(0\,1\,0\,0\,1\,0\,0) \arrow[d, "e_6 = t_5"] & \\
 & n_3 = M_3(0\,0\,0\,0\,0\,1\,0) \arrow[d, "e_4 = t_6"] & & n_6 = M_6(0\,1\,0\,0\,0\,0\,1) \arrow[d, "e_7 = t_7"] & \\
 & \begin{array}{c} n_4 = M_4(1\,0\,0\,0\,0\,0\,0) \\ \text{"old"} \end{array} & & n_7 = M_7(1\,\omega\,0\,0\,0\,0\,0) \arrow[d, "e_8 = t_1"] & \\
 & & & n_8 = M_8(0\,\omega\,1\,0\,0\,0\,0) \arrow[dl, "e_9 = t_2"'] \arrow[dr, "e_{12} = t_3"] & \\
 & & n_9 = M_9(0\,\omega\,0\,1\,0\,0\,0) \arrow[d, "e_{10} = t_4"] & & n_{12} = M_{12}(0\,\omega\,0\,0\,1\,0\,0) \arrow[d, "e_{13} = t_5"] \\
 & & n_{10} = M_{10}(0\,\omega\,0\,0\,0\,1\,0) \arrow[d, "e_{11} = t_6"] & & n_{13} = M_{13}(0\,\omega\,0\,0\,0\,0\,1) \arrow[d, "e_{14} = t_7"] \\
 & & \begin{array}{c} n_{11} = M_{11}(1\,\omega\,0\,0\,0\,0\,0) \\ \text{"old"} \end{array} & & \begin{array}{c} n_{14} = M_{14}(1\,\omega\,0\,0\,0\,0\,0) \\ \text{"old"} \end{array}
\end{tikzcd}$$

Fig.5.4 Reachability graph of the XTPTN shown in Fig.5.3(a)

$$\begin{tikzcd}[column sep=small]
 & n_0 = M_0(1\,0\,0\,0\,0\,0\,0) \arrow[d, "e_1 = t_1"] & \\
 & n_1 = M_1(0\,1\,1\,0\,0\,0\,0) \arrow[dl, "e_2 = t_2"'] \arrow[dr, "e_4 = t_3"] & \\
n_2 = M_2(0\,1\,0\,1\,0\,0\,0) \arrow[d, "e_3 = t_4"] & & n_4 = M_4(0\,1\,0\,0\,1\,0\,0) \arrow[d, "e_5 = t_5"] \\
\begin{array}{c} n_3 = M_3(0\,0\,0\,0\,0\,1\,0) \\ \text{"dead-end"} \end{array} & & \begin{array}{c} n_5 = M_5(0\,1\,0\,0\,0\,0\,1) \\ \text{"dead-end"} \end{array}
\end{tikzcd}$$

Fig.5.5 Reachability graph of the XTPTN shown in Fig.5.3(b)

Examination of Fig.5.5 shows that this net will eventually deadlock, no matter what firing sequence is chosen. A system which exhibits deadlock is not suitable for use because once a system is in deadlock it is impossible to fire any further transition. Examination of Fig.5.5 shows that transition t_6 does not appear as an arc label in the graph, i.e. t_6 is dead in the net. This means that transition t_6 in the corresponding SFC representation, Fig.5.2(b), is unreachable. Hence, a design objective is to ensure that the XTPTN of an SFC is deadlock free (i.e. every transition is reachable by progressing through some further firing sequence).

The reachability graph can also be used for other purposes. For instance, it can provide the information of potential maximum execution concurrency about the SFC (i.e. how many states may be active at the same time) to the operating system.

place p_i	Concurrent set of Fig.5.4
p_1	$\{p_2\}$
p_2	$\{p_1, p_3, p_4, p_5, p_6, p_7\}$
p_3	$\{p_2\}$
p_4	$\{p_2\}$
p_5	$\{p_2\}$
p_6	$\{p_2\}$
p_7	$\{p_2\}$

place p_i	Concurrent set of Fig.5.5
p_1	$\{\}$
p_2	$\{p_3, p_4, p_5, p_7\}$
p_3	$\{p_2\}$
p_4	$\{p_2\}$
p_5	$\{p_2\}$
p_6	$\{\}$
p_7	$\{p_2\}$

Table 5.1 Concurrent sets derived from Fig.5.4 and Fig.5.5

place p_i	Set of sets of Fig.5.4
p_1	$\{<p_2>\}$
p_2	$\{<p_1>, <p_3>, <p_4>, <p_5>, <p_6>, <p_7>\}$
p_3	$\{<p_2>\}$
p_4	$\{<p_2>\}$
p_5	$\{<p_2>\}$
p_6	$\{<p_2>\}$
p_7	$\{<p_2>\}$

place p_i	Set of sets of Fig.5.5
p_1	$\{\}$
p_2	$\{<p_3>, <p_4>, <p_5>, <p_7>\}$
p_3	$\{<p_2>\}$
p_4	$\{<p_2>\}$
p_5	$\{<p_2>\}$
p_6	$\{\}$
p_7	$\{<p_2>\}$

Table 5.2 Set of sets of each place for Fig.5.4 and Fig.5.5

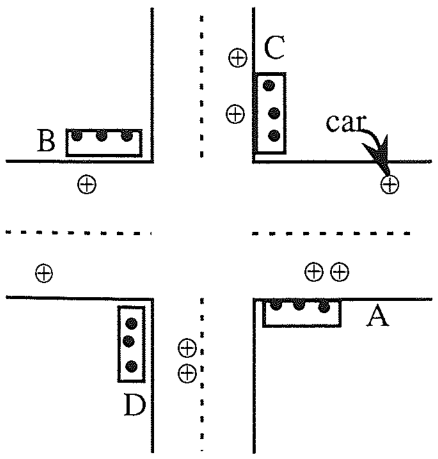
The concurrent set for each place as shown in table 5.1 can be obtained by inspecting the whole reachability graph. Concurrent set is useful in the analysis of the fault tolerance and system recoverable capability of a concurrent or distributed system [Skeen and Stonebraker 83, Hill and Holding 90] because, by definition, the information provided by the concurrent set determines what the potential local states of other components are for a given local state. (In most cases, concurrent set construction needs reachability graph. However, for a special class of Petri nets the concurrent set for each place can be calculated without generating the reachability graph first. The definition of the class of Petri nets and the algorithm of calculating the concurrent set are presented in appendix A).

Although concurrent set provides useful information on the potential concurrent behaviour, it does not reflect the relationships between elements of a concurrent set. To reflect this relationship, the concurrent set for each place *p* can be further classified into a set of sets as shown in table 5.2 in which each set is formed by all the places (local states) that can be tokenised at the same time (in a global state) as *p*. (For this particular example each set of places contains only one element: in general the sets will comprise more than one element). Thus, the meaning of a basic set plus *p* is the same as a marking in which *p* is tokenised.

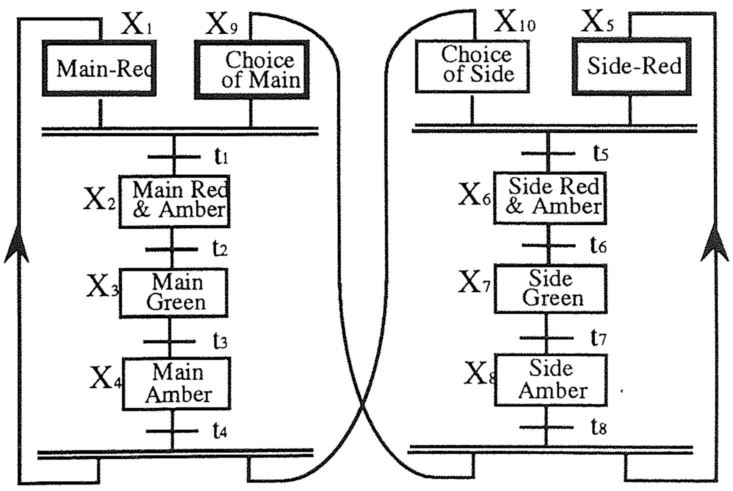
5.3 An Example

5.3.1 A Traffic Light Control System

In this section, the approach will be illustrated by considering the qualitative and quantitative behaviour of a simple traffic light system. The qualitative analysis analyses the system behaviour without considering of timing issues. The quantitative analysis includes timing analysis. Consider the traffic light control system shown in Fig.5.6(a). Let A and B be "main road lights" — Main and C and D be "side road lights" — Side. The SFC functional description for the traffic light control system is shown in Fig.5.6(b). The XTPTN model for the SFC is shown in Fig.5.7.



(a) The traffic light system

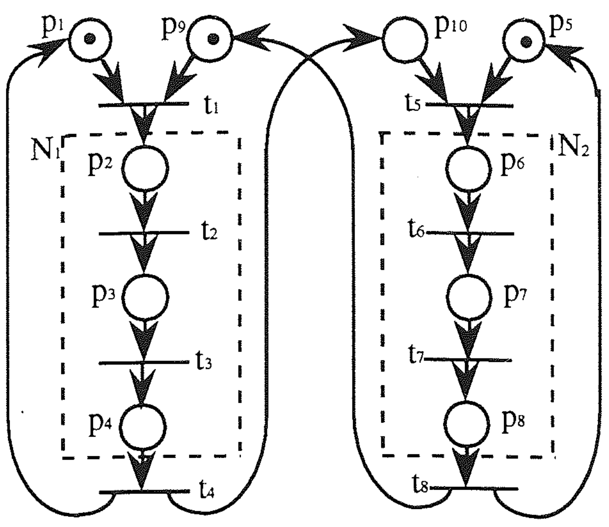


(b) SFC control system model

X_1: A and B are both red;
X_2: A and B are red and amber;
X_3: A and B are both green;
X_4: A and B are both amber;
X_5: C and D are both red;
X_6: C and D are red and amber;
X_7: C and D are both green;
X_8: C and D are both amber;
X_9: choice of Main road;
X_{10}: choice of Side road;
Initial states $\{X_1, X_5, X_9\}$

t_1: change A & B from red to red & amber;
t_2: change A & B from red & amber to green;
t_3: change A & B from green to amber;
t_4: change A & B from amber to red and give the choice to Side road;
t_5: change C & D from red to red & amber;
t_6: change C & D from red & amber to green
t_7: change C & D from green to amber;
t_8: change C & D from amber to red and give the choice to Main road;

Fig.5.6 Traffic light system and its SFC control system



$\upsilon(p_1, \tau) - \tau = z_1;$ $\Psi(t_1) = <0, 0>$

$\upsilon(p_2, \tau) - \tau = z_2;$ $\Psi(t_2) = <0, 0>$

$\upsilon(p_3, \tau) - \tau = z_3;$ $\Psi(t_3) = <0, 0>$

$\upsilon(p_4, \tau) - \tau = z_4;$ $\Psi(t_4) = <0, 0>$

$\upsilon(p_5, \tau) - \tau = z_5;$ $\Psi(t_5) = <0, 0>$

$\upsilon(p_6, \tau) - \tau = z_6;$ $\Psi(t_6) = <0, 0>$

$\upsilon(p_7, \tau) - \tau = z_7;$ $\Psi(t_7) = <0, 0>$

$\upsilon(p_8, \tau) - \tau = z_8;$ $\Psi(t_8) = <0, 0>$

$\upsilon(p_9, \tau) - \tau = \delta;$ $TURT_0(p_1) = 0;$

$\upsilon(p_{10}, \tau) - \tau = \delta;$ $TURT_0(p_5) = 0;$

$TURT_0(p_9) = 0;$

Fig.5.7 XTPTN representation of the SFC control system

In Fig.5.7, the unavailability times associated with places p_9 and p_{10} are assumed to be the same because they have the same functionalities; the enabled interval associated with each transition is *<0,0>* which means that each transition (traffic lights) will be fired (changed) immediately when it is enabled (can be changed); and all tokens in the initial marking are available (i.e. $TURT_0(p_1) = 0$, $TURT_0(p_9) = 0$, $TURT_0(p_5) = 0$).

5.3.2 Qualitative Analysis

Since the traffic light control system is a critical system, some safety and liveness requirements must be imposed on it such as the mutual exclusion problem (i.e. Main road and Side road must be mutual exclusive in the critical section represented by subnets N_1 and N_2), the reactivation problem (i.e. the net should be 1-bounded or safe), and the deadlock problem (i.e. the net should be live). The following presents these important properties imposed on the traffic control system and their proofs.

Property-1: The XTPTN shown in Fig.5.7 is a 1-bounded (or safe) net.

Proof: All places of the net shown in Fig.5.7 are included in following three loops:

$L_1 = \{p_2, t_2, p_3, t_3, p_4, t_4, p_{10}, t_5, p_5, t_6, p_6, t_7, p_7, p_8, t_8, p_9\}$

$L_2 = \{p_1, t_1, p_2, t_2, p_3, t_3, p_4, t_4\}$

$L_3 = \{p_5, t_5, p_6, t_6, p_7, t_7, p_8, t_8\}$

Based on the token invariant theorem of nets (presented in Chapter 2), the number of tokens in each loop remains invariant. Since only one token is in each loop in the initial marking, this means that 1-bounded or safe property of the net is guaranteed. □

Property-2: Main road and Side road must be mutually exclusive for following (local) states: {Main_Red&Amber, Main_Green, Main_Amber, Side_Red&Amber, Side_Green, Side_Amber}. In terms of XTPTN, this property can be stated as:

$$\forall p_i, p_j \in \{p_2, p_3, p_4, p_6, p_7, p_8\},\ p_i \neq p_j:\ \neg\exists M \in [M_0>\ [M(p_i) \neq 0 \wedge M(p_j) \neq 0] \qquad (5.7)$$

Proof: This property can be proved based on the evidence derived from the proof of Property-1. Since only one token can remain in L_1 at any instant, the mutually exclusion among places $\{p_2, p_3, p_4, p_6, p_7, p_8\}$ is guaranteed. Otherwise, there should be more than one initial token in L_1 which is contradicted with the initial marking. □

Property-2 can also be verified using concurrent set. Table 5.3 shows the concurrent set for each place, which is obtained by performing the algorithm presented in Appendix A. By examining Table 5.3, it can be seen that:

$$\neg\exists p_i, p_j \in \{p_2, p_3, p_4, p_6, p_7, p_8\}: p_i \neq p_j \wedge [p_i \in CP(p_j) \vee p_j \in CP(p_i)]$$

place	Concurrent set	place	Concurrent set
p_1	$\{p_5, p_6, p_7, p_8, p_9, p_{10}\}$	p_6	$\{p_1\}$
p_2	$\{p_5\}$	p_7	$\{p_1\}$
p_3	$\{p_5\}$	p_8	$\{p_1\}$
p_4	$\{p_5\}$	p_9	$\{p_1, p_5\}$
p_5	$\{p_1, p_2, p_3, p_4, p_9, p_{10}\}$	p_{10}	$\{p_1, p_5\}$

Table 5.3 Concurrent set for XTPTN shown in Fig.5.7

Property-3: The control system must be deadlock-free.

Property-3 is a liveness property which is much more difficult to prove compared to safety property. The details of proof for this property can be found in Appendix B.

Although the system was designed with these properties and satisfies the imposed requirements, it does not say anything about the system timing constraints.

5.3.3 Timing Analysis

System timing constraints can be derived by assigning variable time parameters to places and forming relational expressions of relative timing. Timing constraint has direct effect on system performance. For example, to increase the throughput of traffic, it is required that the Side road (C & D) changes from red light into red & amber lights as soon as the Main road (A & B) changes from amber light into red light and vice versa. This implies that the token in place p_5 should become available before the token in place p_{10} becomes available, otherwise, the throughput could be decreased due to the unnecessary delay. Thus, the token unavailable time in place p_5 and the token unavailable times in places $\{p_2, p_3, p_4, p_9, p_{10}\}$ should satisfy certain relationship. Similar relationship should exist between the token unavailable time in place p_1 and token unavailable times in places $\{p_6, p_7, p_8, p_9, p_{10}\}$.

In order to fire transition t_5 (i.e. to change the Side road from red light into red & amber lights) as early as possible after place p_{10} is tokenised (i.e. the Side road choice is released by the Main road light change from amber into red), the *up limit* of token unavailable time z_5 in place p_5 needs to be less than the cycle time of loop $p_9 \rightarrow p_2 \rightarrow p_3 \rightarrow p_4 \rightarrow p_{10}$. The relationship can be expressed in timing requirements as:

$$z_5 \le z_2 + z_3 + z_4 + \delta + \delta \tag{5.8}$$
where $z_i = \upsilon(p_i, \tau) - \tau$ and $\delta = \upsilon(p_9, \tau) - \tau = \upsilon(p_{10}, \tau) - \tau$

If (5.8) is imposed on z_5, then it is guaranteed that no unnecessary delay for Side road light change from red into red & amber. This is because that the token in p_5 always becomes available before the token in p_{10} becomes available such that transition t_5 can always be fired without delay.

For the Main road, symmetrically, the *up limit* of token unavailable time z_1 in place p_1 needs to be less than the cycle time of loop $p_{10} \rightarrow p_4 \rightarrow p_7 \rightarrow p_8 \rightarrow p_9$. The relationship can be expressed in timing requirements as:

$$z_1 \le z_6 + z_7 + z_8 + \delta + \delta \tag{5.9}$$
where $z_i = \upsilon(p_i, \tau) - \tau$ and $\delta = \upsilon(p_9, \tau) - \tau = \upsilon(p_{10}, \tau) - \tau$

Since states modelled by places p_9 and p_{10} have no external effect, the token unavailable time δ can be reduced to zero. If $C_1 = z_2 + z_3 + z_4$ is used to represent the total time of Main road in traffic lights {red & amber, green, amber} and $C_2 = z_6 + z_7 + z_8$ is used to represent the total time of Side road in traffic lights {red & amber, green, amber}, then we have:

$$\begin{cases} z_5 \le C_1 \quad \text{where } C_1 = z_2 + z_3 + z_4 & (5.10) \\ z_1 \le C_2 \quad \text{where } C_2 = z_6 + z_7 + z_8 & (5.11) \end{cases}$$

The above equations describe the timing constraints for the system. Note that C_1 is not necessary equal to C_2 because the system controls one Main road and one Side road.

Although a lot of information can be obtained from the reachability graph, concurrent set, and token invariants, they do not explicitly reflect any causal order among transitions. That is, it is difficult to see a potential concurrent set of transitions that may be fired concurrently (or simultaneously in SFC), since the system behaviour is observed sequentially using interleaving semantics. If the properties and behaviour of a system described in an SFC are interpreted using net theory, then it is desirable to find a means to observe the concurrent transitions of SFC in terms of nets.

Trace theory [Mazurkiewicz 87, 88] was introduced for describing the non-sequential behaviour of concurrent systems from its sequential observations. Since the reachability graph is generated based on sequential observation, trace theory can be used as a tool to reason about it. In the following trace theory will be used to derive the potentially concurrent behaviour of SFC transitions from the reachability graph of the XTPTN of the SFC.

5.4 SFC Transition Analysis — Trace Theory-Based Analysis

5.4.1 Introduction

In SFC transitions fire simultaneously if they can. Consider the SFC shown in Fig.5.8(a) in which $T_1 = c_1$ and $T_2 = c_2$, where c_1 and c_2 are two conditions. When steps X_{i1} and X_{j1} are active, if c_1 and c_2 both become true, then transitions T_1 and T_2 should fire simultaneously. Even if they are not fired simultaneously, T_1 is causally independent of T_2.

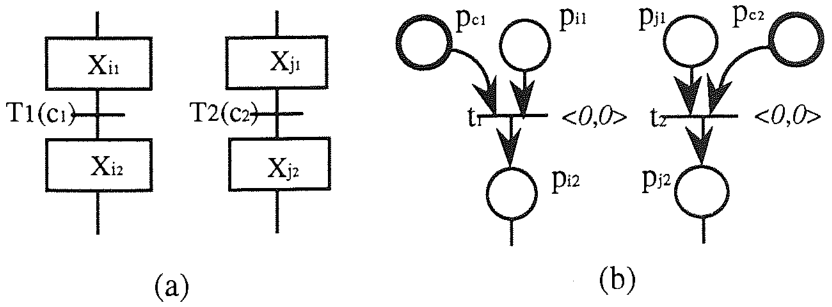


Fig.5.8 An SFC showing potential simultaneous firing and its XTPTN model

The firing rule defined for a XTPTN net describes a change of marking caused by a single transition firing. The reachability graph describes all firing sequences of the net. A directed path on the reachability graph can be viewed as a sequential observation of the behaviour of net. Such a firing sequence reflects only an observation made by a sequential observer capable of seeing only one transition occurrence at a time. Given a transition firing sequence $\Pi = \mathcal{H}(e_1)\mathcal{H}(e_2) \ldots \mathcal{H}(e_n)$ from a reachability graph $G = (N, E, \mathcal{F}, \mathcal{H})$ (see page 99), the following case may hold: $\exists \mathcal{H}(e_i), \mathcal{H}(e_j) \in \Pi, i < j$, and $\mathcal{H}(e_i) \neq \mathcal{H}(e_j) \Rightarrow$ $\mathcal{H}(e_i)$ and $\mathcal{H}(e_j)$ are both $\mathcal{F}(I(e_i))$-enabled and $\bullet\mathcal{H}(e_i) \cap \bullet\mathcal{H}(e_j) = \varnothing$., where $I(e_i)$ denotes the input node of e_i in G. That means although two or more transitions are fired sequentially in a firing sequence of the XTPTN reachability graph, they may actually be fired concurrently during the system execution. For example, the XTPTN model shown in Fig.5.8(b) is a case where the tokens from places $\{p_{i1}, p_{j1}, p_{c1}, p_{c2}\}$ could become available at the same time. In fact, the ordering of transitions in a firing sequence reflects not only the causal ordering of transition firings, but also an observational ordering resulting from a specific view over concurrent events. Therefore, the structure of a firing sequence alone does not allow us to decide whether the difference in ordering is caused by a choice resolution (a decision made in the system) or by different observations of concurrency. In order to extract the causal order of transition firing from the sequential observations of behaviour of a net, we must [illegible] the sequential behaviour with additional information — the dependency of transitions.

The following describes how the concurrent (or independent) behaviour of SFC can be generated from the sequential firing sequence of XTPTN using trace theory. Intuitively, if two adjacent transitions in a sequential firing sequence (a path) are independent of each other (i.e. the order of their firings is irrelevant), then we can say that these two transitions in SFC have the potential to be fired simultaneously if the conditions associated with them become true at the same time. The following sections present an overview of those aspects of trace theory needed in the analysis of XTPTN and SFC, (for more details of trace theory, the reader is referred to [Mazurkiewicz 87, 88, Rozenberg 87]).

5.4.2 Trace Theory

Definition *Reliance alphabet*
A *concurrent alphabet* is any ordered pair $\Sigma = (A, D)$ where A is a finite set (the alphabet of Σ) and D is a symmetric and reflexive binary relation defined on A (called the *dependency* in Σ). (A, A^2) is the concurrent alphabet with full dependency. $I = A^2 - D$ is a symmetric and irreflexive binary relation defined on A (called the *independency* in Σ). Triple (A, I, D), where A is an alphabet, D is a dependency relation over A and I is an independency relation over A, is called a *reliance alphabet*.

For instance, let us suppose A = {a, b, c, d} and D = {<a,a>,<b,b>,<c,c>, <d,d>,<a,b>, <b,a>,<b,c>,<c,b>,<b,d>,<d,b>}. Then Σ = (A, D) is a concurrent alphabet and the independency I in Σ is defined as I = A^2 - D = {<a,c>,<c,a>,<a,d>,<d,a>,<c,d>,<d,c>} and the reliance alphabet is the triple (A, I, D), where A^2 = {<a,a>,<b,b>,<c,c>,<d,d>, <a,b>,<b,a>,<a,c>,<c,a>,<a,d>,<d,a>,<b,c>,<c,b>,<b,d>,<d,b>,<c,d>,<d,c>}.

Definition *Trace equivalence relation*
Let Σ = (A, D) be a concurrent alphabet. The *trace equivalence relation* over Σ, denoted by $\equiv_\Sigma$, is defined as:

$$\forall\, a, b \in A: \quad <a,b> \in I_\Sigma \Rightarrow ab \equiv_\Sigma ba \tag{5.12}$$

where $I_\Sigma = A^2 - D$ is the independency relation in Σ

That is, $\forall \sigma_1, \sigma_2 \in A^*$: $\sigma_1 \equiv_\Sigma \sigma_2$ iff:

$$\exists \mu, \nu \in A^* \text{ and } \exists <a,b> \in I_\Sigma\ [\sigma_1 = \mu ab\nu \text{ and } \sigma_2 = \mu ba\nu] \tag{5.13}$$

where A^* denotes the finite sequences over A

Equivalence classes of relation $\equiv_\Sigma$ are called *traces* over Σ; a trace generated by σ is denoted by $[\sigma]_\Sigma$ and the set of all traces over Σ is denoted by $\Theta(\Sigma)$.

Let us consider again the concurrent alphabet Σ = (A, D) given above. Suppose σ_1 = bacd $\in A^*$ and σ_2 = bcad $\in A^*$, then we have $\sigma_1 \equiv_\Sigma \sigma_2$ because <a,c> $\in I_\Sigma$. For string σ = bacd $\in A^*$, trace $[\text{bacd}]_\Sigma$ = {badc, bdac, bdca, bcda, bcad}.

Given a concurrent alphabet Σ = (A, D), the trace equivalence relation over Σ is used to establish equivalencies between strings of alphabet symbols which differ only in the ordering of "adjacent independent symbols" (i.e., symbols belong to independency I in Σ). That is, a single trace contains all strings which differ only in the ordering of adjacent independent symbols. For example, symbol 'c' is independent to symbol 'd' and they are adjacent to each other in strings 'bacd' and 'badc' such that 'bacd' and 'badc' belong to a trace. If symbols in A are thought as atomic system events, then traces can be considered as composed events in which some elementary events occur independently (i.e. without any causal relationship) of each other. The notion of traces allows us to eliminate from strings the ordering between events which occur independently of each other. This view of trace theory is used to define the non-sequential behaviour of a sequentially observed system.

Each trace can be graphically represented as an abstract acyclic node-labelled directed graph such that the ordering of symbol occurrences within the trace can be explicitly viewed.

Let $\Sigma = (A, D)$ be a concurrent alphabet. The *dependency graph* over Σ (or *d-graph* for short) is a finite and acyclic node-labelled graph where the graph node is labelled with a symbol in A and any two nodes are connected with an arc iff they are labelled with two symbols of A in the relation D.

Definition *Dependency graph*
A dependency graph is defined as a triple $G = (N, E, \Phi)$, where:

N is a finite set of nodes;
$E \subseteq N \times N$;
$\Phi: N \rightarrow A$.

Definition *isomorphic*
Let $g_1 = (N_1, E_1, \Phi_1)$ and $g_2 = (N_2, E_2, \Phi_2)$ be two *d*-graphs. We say g_1 and g_2 are isomorphic, denoted as $g_1 \cong g_2$, iff there exists a bijection $\xi: N_1 \rightarrow N_2$ such that:

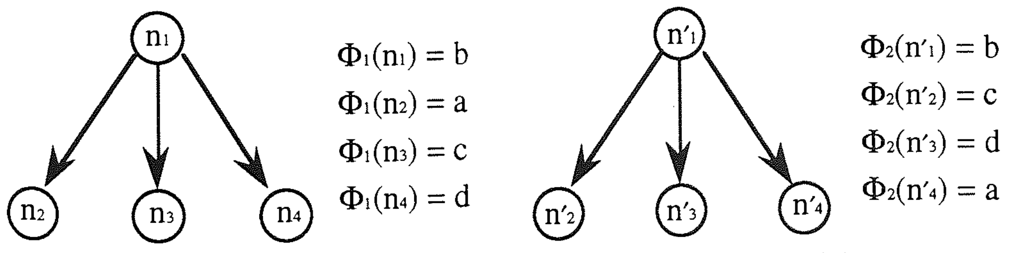
$$(\forall n \in N_1)\ [\Phi_2(\xi(n)) = \Phi_1(n)] \text{ and} \qquad (5.14)$$
$$\text{and} \quad (\forall n, n' \in N_1)\ [(n, n') \in E_1 \text{ iff } (\xi(n), \xi(n')) \in E_2] \qquad (5.15)$$

Given a concurrent alphabet $\Sigma = (A, D)$ and $\sigma \in A^*$, the node-labelled directed graph of σ denoted by $<\sigma>_\Sigma$ is constructed as:

i) σ is empty then $<\sigma>_\Sigma$ is the empty graph (no nodes and no arcs);
ii) $\sigma = \mu a$ where $\mu \in A^*$ and $a \in A$, then $<\sigma>_\Sigma$ is obtained by adding to $<\mu>_\Sigma$ a new node labelled with a and new arcs leading to the new node from all the nodes in $<\mu>_\Sigma$ labelled with symbols which are dependent upon a in Σ.

Clearly, for any string $\sigma \in A^*$ the $<\sigma>_\Sigma$ constructed as above is a *d*-graph. For example, the *d*-graph of σ_1 = bacd as shown in Fig.5.9(a) and *d*-graph of σ_2 = bcda as shown in fig.5.9(b) are isomorphic (i.e., Fig.5.9(a) $\cong$ Fig.5.9(b)) via the bijection ξ defined as:

$$\xi(n_1) = n'_1; \quad \xi(n_2) = n'_4; \quad \xi(n_3) = n'_2; \quad \xi(n_4) = n'_3; \qquad (5.16)$$



(a) *d*-graph of 'bacd' (b) *d*_graph of 'bcda'

Fig.5.9 Dependency graphs for strings of alphabet symbols

The relationship between traces and d-graphs over Σ is expressed by the following theorem whose proof has been given by Mazurkiewicz in [Mazurkiewicz 87] and can be found in Appendix C.

Theorem $\quad \forall\, \sigma', \sigma'' \in A^*: \quad <\sigma'>_\Sigma \cong <\sigma''>_\Sigma$ iff $[\sigma']_\Sigma = [\sigma'']_\Sigma$ (5.17)

Thus for a trace $[\sigma]_\Sigma \in \Theta(\Sigma)$, $<\sigma>_\Sigma$ is an "isomorphic invariant" of $[\sigma]_\Sigma$— it does not depend on the choice of a representative in $[\sigma]_\Sigma$.

Definition Let $\Sigma = (A, D)$ be a concurrent alphabet. Any ordered pair (Σ, T) where $T \subseteq \Theta(\Sigma)$ is called a trace system.

5.4.3 Nets and Traces

Definition *Dependence relation*
Let N = (PN, Υ, υ, ι, $\mathcal{C}$, Ψ, TURT$_0$) be a XTPTN with capacity $K(p) \leq 1$, where PN = (P, T, F, M_0) and $p \in P$, the dependence relation over T in N, denoted as D_N, is defined as:

$$\forall\, t_1,t_2 \in T: <t_1,t_2> \in D_N \Leftrightarrow (\bullet t_1 \bullet) \cap (\bullet t_2 \bullet) \neq \varnothing \qquad (5.18)$$

Definition *Independence relation*
Let N = (PN, Υ, υ, ι, $\mathcal{C}$, Ψ, TURT$_0$) be a XTPTN with capacity $K(p) \leq 1$, where PN = (P, T, F, M_0) and $p \in P$, the independence relation over T in N, denoted as I_N, is defined as:

$$\forall\, t_1,t_2 \in T: <t_1,t_2> \in I_N \Leftrightarrow (\bullet t_1 \bullet) \cap (\bullet t_2 \bullet) = \varnothing \qquad (5.19)$$

Clearly, I_N is irreflexive and symmetric and D_N is the complement of "relation I_N" which can be represented as $D_N = T \times T - I_N$. By definition of I_N, two transitions are said to be independent in N iff they have no common input and common output places. If two independent transitions occur next to each other in a firing sequence, the order of their occurrences is actually irrelevant since (according to the firing rule defined in SFC) they may occur concurrently in the execution (if the conditions associated with them become TRUE simultaneously in SFC). To detect which transitions are potential concurrent (or may be fired simultaneously in SFC), the ordered sequential firing sequence of XTPTN will be replaced by non-sequential one. That is, the behaviour of the XTPTN system will be expressed using traces rather than sequential sequences.

To model a net N = (PN, Υ, υ, ι, $\mathcal{C}$, Ψ, TURT$_0$) by a trace system, one has to define the corresponding concurrent alphabet over Σ_N.

If the sets of A and relation D in the concurrent alphabet Σ are replaced by the set of transitions T_N and the dependency relation D_N of the net respectively, then it can be easily verified that $\Sigma_N = (T_N, D_N)$ is a concurrent alphabet associated with N because D_N is a symmetric and reflexive. Respectively, we can use $\equiv_N$ to denote the trace equivalence relation over Σ_N and use $[\sigma]_N$ to denote the equivalence class of $\equiv_N$ containing σ. The *trace behaviour* of N is defined as a trace system $TS_N = (\Sigma_N, [\sigma]_N)$, where $\sigma \in T_N^*$ (a set of finite sequences over the set of transitions T in the net) is a firing sequence of N. Each element of TS_N is called a *firing trace* of N. Clearly, each firing sequence of N is contained in some equivalence class belonging to TS_N. In another words, each firing trace includes a firing sequence of N and all the other equivalent firing sequences which lead to the same marking of N. Here the equivalent firing sequences mean that they consist of the same set of transitions and their *d*-graphs are isomorphic.

After obtaining a trace system $TS_N = (\Sigma_N, [\sigma]_N)$, where $\Sigma_N = (T_N, D_N)$ and $\sigma \in T_N^*$, for a given net $N = (PN, \Upsilon, \upsilon, \iota, \mathcal{C}, \Psi, TURT_0)$ in which $PN = (P, T, F, M_0)$, the trace can be used to determine the non-sequential behaviour observation. If $[\sigma]_N$ is a firing trace, where $\sigma \in T_N^*$ is a sequential firing sequence, then the non-sequential behaviour of $[\sigma]_N$ can be observed by representing one sequential firing sequence in $[\sigma]_N$ as a *d*-graph.

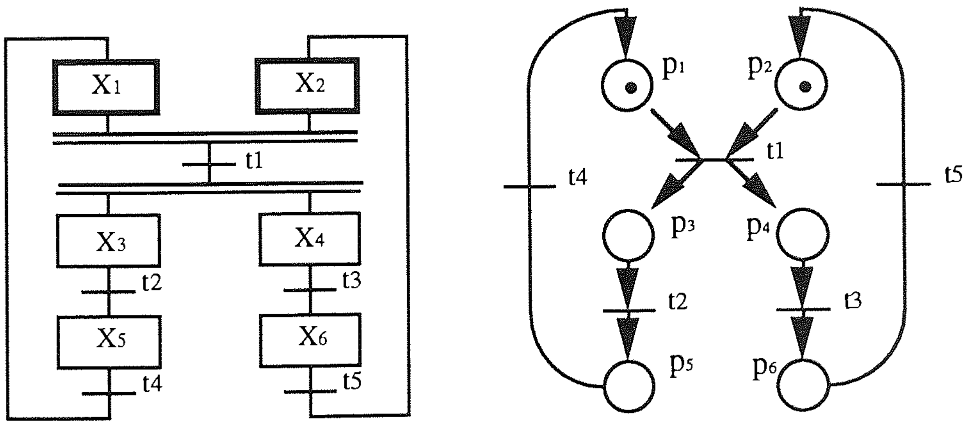


Fig.5.10 An SFC and its transformed XTPTN

By examining the transformed XTPTN shown in Fig.5.10 from SFC also shown in Fig.5.10, following dependence and independence relations can be constructed according to the XTPTN definitions:

$$T_N = \{t_1, t_2, t_3, t_4, t_5\};$$
$$D_N = \{<t_1,t_2>, <t_2,t_1>, <t_1,t_3>, <t_3,t_1>, <t_1,t_4>, <t_4,t_1>, <t_1,t_5>, <t_5,t_1>, <t_2,t_4>, <t_4,t_2>, <t_3,t_5>, <t_5,t_3>, <t_1,t_1>, <t_2,t_2>, <t_3,t_3>, <t_4,t_4>, <t_5,t_5>\};$$
$$I_N = T_N{}^2 - D_N = \{t_1, t_2, t_3, t_4, t_5\}\times\{t_1, t_2, t_3, t_4, t_5\} - D_N;$$
$$= \{<t_2,t_3>, <t_3,t_2>, <t_2,t_5>, <t_5,t_2>, <t_3,t_4>, <t_4,t_3>, <t_4,t_5>, <t_5,t_4>\}$$

Examination of reachability graph shown in Fig.5.11 shows that $\sigma = t_1t_2t_3t_4t_5$ is a firing sequence and its *d*-graph is shown in Fig.5.12.

$$\begin{tikzcd}[column sep=tiny]
 & & & n_0 = M_0\,(1\,1\,0\,0\,0\,0) \arrow[d, "e_1 = t_1"] & & & \\
 & & & n_1 = M_1\,(0\,0\,1\,1\,0\,0) \arrow[dl, "e_2 = t_2"'] \arrow[dr, "e_3 = t_3"] & & & \\
 & & n_2 = M_2\,(0\,0\,0\,1\,1\,0) \arrow[dl, "e_4 = t_4"'] \arrow[dr, "e_5 = t_3"] & & n_3 = M_3\,(0\,0\,1\,0\,0\,1) \arrow[dl, "e_6 = t_2"'] \arrow[dr, "e_7 = t_5"] & & \\
 & n_4 = M_4\,(1\,0\,0\,1\,0\,0) \arrow[dr, "e_8 = t_3"] & & n_5 = M_5\,(0\,0\,0\,0\,1\,1) \arrow[dl, "e_9 = t_4"'] \arrow[dr, "e_{10} = t_5"] & & n_6 = M_6\,(0\,1\,1\,0\,0\,0) \arrow[dl, "e_{11} = t_2"'] & \\
 & & n_7 = M_7\,(1\,0\,0\,0\,0\,1) \arrow[dr, "e_{12} = t_5"] & & n_8 = M_8\,(0\,1\,0\,0\,1\,0) \arrow[dl, "e_{13} = t_4"'] & & \\
 & & & n_9 = M_9\,(1\,1\,0\,0\,0\,0) \text{ "old"} & & &
\end{tikzcd}$$

Fig.5.11 Reachability graph of XTPTN shown in Fig.5.10

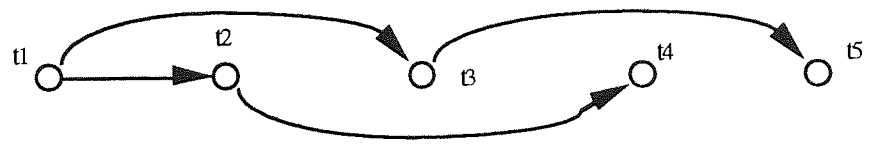


Fig.5.12 The *d*-graph for a firing sequence

Since the arc in *d*-graph represents a causal dependency between transitions, two elements are *concurrent* iff they are not linked by an arrow sequence. By inspecting the *d*-graph shown in Fig.5.12 for the sequence $\sigma = t_1t_2t_3t_4t_5$ we can derive that transition t_2 is concurrent with t_3 and may potentially be fired simultaneously in SFC if the conditions associated with them are changed into TRUE at the same time. Potential simultaneous firing is also applicable to the following pairs $<t_2,t_5>$, $<t_3,t_4>$, $<t_4,t_5>$ because of the causal independency between them. Also, from dependency point of view, we can derive that transition t_1 must be fired earlier than t_2 and t_3, and t_3 earlier than t_5 because they are linked by an arrow sequence.

Because the ordering of transitions with a firing trace is determined by mutual dependencies of transitions and the *d*-graphs of all firing sequences belonging to a trace are isomorphic, we can get the causal relationships and non-sequential behaviour of all the firing sequences in the trace by inspecting only one representative's *d*-graph. For example, the firing trace containing firing sequence $\sigma = t_1t_2t_3t_4t_5$ is as: $[t_1t_2t_3t_4t_5]_N = \{t_1t_2t_4t_3t_5, t_1t_2t_3t_5t_4, t_1t_3t_2t_4t_5, t_1t_3t_2t_5t_4, t_1t_3t_5t_2t_4\}$.

From the behavioural point of view, each firing trace *uniquely* determines the path (represented by the firing sequences in the trace) by which the specified marking (state) is reached from the initial marking (state). Different firing sequences which reflect different sequential observations may result from the same non-sequential behaviour. The non-sequential behaviour can be described by the trace and can be identified by the *d*-graph of any representitive of the trace.

Summary

This chapter discusses the analysis of SFC using existing net analysis techniques. The techniques used in the analysis include reachability tree and trace theory.

Chapter 6

Towards A Rule-based Approach for Real-Time Process Control Systems

6.1 Introduction

Chapter 4 and Chapter 5 presented a method of translating an SFC design into an extended Petri net (XTPTN) and a method of analysing the resulting XTPTN to determine the properties of the original SFC. Since SFC was defined in IEC1131 as a graphical description language, it lacks a systematic way of synthesising SFC from the real world systems. From this observation, a standardised approach to the development of real-time process control systems is required such that the system (software) functional requirements can be identified, captured, analysed, expressed in SFC and implemented using SFC and other programming languages as defined in IEC1131. In this chapter, a rule-based formalism will be investigated for the capture of (software) functional requirements of real-time process control systems.

The (software) functional requirement is the first document of a system's required behaviour. Errors in this document are difficult and expensive to correct if propagated to the design phase (or worse, to the implementation) [Boehm 80, Goldsack and Finkelstein 91]. Thus, the captured functional requirements must be analysed before system design begins. In this chapter, the following issues will be discussed:

a) the rule-based formalism;
b) a method to support the construction of rule-based descriptions;
c) the analysis of rule-based descriptions.

The analysis approach presented in this chapter is to convert the rule-based description into a Petri net model. Assertions which are enforced by the specification are then verified using the formal verification techniques of Petri nets [Reisig 85, Murata 89] and temporal Petri nets [Suzuki and Lu 89, He and Lee 90, Sagoo and Holding 91, Sagoo 92].

6.2. Rule-Based Approach

6.2.1 Rule-Based Formalism

Rule-based formalisms are a popular method for knowledge representation in expert systems because the experts find it easy to express methods for solving problems in their application areas using rules [Hayes-Roth 85]. However, in recent years, rule-based formalisms have been used by researchers in other areas such as real-time process-control systems as discussed in Chapter 2 [Pathak and Krogh 89, Wilson and Krogh 89, Etessami and Hura 91], Programmable Logic Controllers (PLC) [Barker *et al* 89, Barker and Song 92], and Information systems [Assche *et al* 88, Loucopoulos and Layzell 89].

Conventionally, a rule consists of two parts: a premise (IF) and an action (THEN) [Waterman 86]. When the IF portion of a rule is satisfied by the facts (data), the action specified by the THEN portion is performed. The rule's action may modify the set of facts (data) or may directly affect the real world. The advantage of rule formalism is that the rule-based description can be formally analysed [Hayes-Roth 85, Loucopoulos and Layzell 89]. Thus, rules provide a formal way of representing the system behaviour against its environment. This feature of rules provides a suitable way to describe the intended behaviour of real-time process control systems. This is possible because most real-time process control systems have a behaviour which can be described by a state space and a set of state transitions which specify the set of possible future states given a present state [Ostroff 89, Jaffe *et al* 91, Shaw 92]. Such a state transition can be described directly as a conventional IF-THEN rule. Thus, integrating a rule-based formalism into the functional requirements description of real-time process control systems looks very natural. Another important feature of the rules is that rule-based formalism supports the development of a system functional requirements in an incremental approach [Loucopoulos and Layzell 89, Willson and Krogh 90]. This indicates the potential of using a rule-based formalism to synthesise the software controller in a real-time process-control system development.

6.2.2 Rules and Events

The purpose of a software functional requirements is to describe a system's behaviour at an early stage of its software development life cycle. The functional requirements should describe *what* the system intend to do, not *how* it does it. The functional requirements should describe what the state transitions are and when to perform them. The state transition describes the observable behaviour of the system and the 'when' clause describes the time- and event-dependent *dynamic* and *reactive* nature of the real-time process-control system. For example, if a system is in a particular state, then only *when* an event (or a guard) occurs (is true) can this state be changed into another one. For a real-time process control system, "when to do" is very important since it provides a means to specify the time at which the state transition occurs.

Although the conventional IF-THEN rule structure provides a natural way of describing the state-based causal transitions comprising the behaviour of a real-time process control system (i.e., what to do) [Komoda *et al* 84, Tashiro *et al* 85, Wilson and Krogh 90, Etessami and Hura 91] (see Section 2.3.2 in Chapter 2), it cannot specify the dynamic and reactive features of the system very well (i.e. "when" features such as event- or time-related stimuli). This arises because of the differences between events and states and a corresponding deficiency of describing the effect of an event (a condition value change) on the transition in a rule formulation. If the rule-based formalism is applied to the functional requirement description of real-time process control systems, it must describe these dynamic functional requirements precisely. In order to describe the event-based (or time-based) constraints in the requirements, the conventional rule structure has to be extended.

The idea of adopting a, more dynamic, form of rule for real-time process control systems appears in an earlier independent case study [Marconi 86] in which an *event-based* rule was found to be useful for defining the software functional requirements of real-time process control systems, where an event is defined as a condition value change from true to false or vice versa [Heninger 80, Faulk and Parnas 88]. The case study concerned an industrial real-time process control system whose most important feature was that the system's behaviour was both state-based and event-driven. As discussed in Chapter 2, the conventional rule structure of IF-THEN form has been extended into a WHEN-IF-THEN form by introducing a trigger [Assche *et al* 88, Loucopoulos and Layzell 89]. A trigger can be interpreted as a guard on the IF-THEN rule. The extended rule is called dynamic rule and the conventional rule is called static rule in [Assche *et al* 88, Loucopoulos and Layzell 89]. Although the dynamic rule concept was originally used in information system developments and maintenance by researchers, it seems reasonable to apply this kind of idea to real-time process control systems because state transitions in real-time (process control) systems are often guarded by a condition change from false to true [Ostroff 89, Shaw 92, David and Alla 92].

For example, the following is a simplified event-based rule example taken from the case study by [Marconi 86] which can be construed to say 'if the computer system is in automatic control mode, then it must *immediately* change into manual control mode from automatic control mode *when* event X occurs':

IF Auto_Control
WHEN Event X
THEN Manu_Control

In the following research (see Chapter 7) event-based rules are shown to be suitable for specifying the behaviour of real-time process control systems because the behaviour of most industrial real-time process control systems are not only state-based but also event-driven, and an event-based rule provides a convenient way to describe both state-based and event-driven features together. The event-based rule clearly specifies when the state transition should be performed by distinguishing between state conditions using the IF clause and event occurrences using the WHEN clause. An event-based rule describes the instantaneous time point at which an event acts upon the state change. From the point of view of real-time control, it is natural to distinguish between an event and a state (condition) [Heninger 80, Faulk and Parnas 88, David and Alla 92, IEC848 88]. During this research it has also been found that the conventional IF-THEN rule is useful although it says nothing which implies the transition may or should occur. It is this feature of the conventional rule that provides a convenient way to describe the state-based behaviour in which an action is never forced to occur. For example, if one wants to describe a machine changing from its stationary state into its rotation state in which the time point at which the state transition occurs is not interesting, then it can simply be specified using the conventional IF-THEN rule construct. For this reason, two types of basic rule constructs will be defined here, one is called *static* (i.e. the conventional IF-THEN) rule, the other one is called *event-based* (i.e. IF-WHEN-THEN) rule.

6.2.3 Syntax and Semantics of Rule-Based Schemes

The static rule has the syntax form shown below:

IF L
THEN L' (6.1)

where L and L' are sets of (local) states of the system in terms of the SFC model, and each local state $l \in L \cup L'$ is defined as an interpretation of a control system assigning an action to a system component.

The set L provides the enabling condition prior to the state transition and set L' represents the states immediately after the state transition. The meaning of this rule construct can be stated as: "if all the local states L in the IF part are active, then the state transition can occur. The occurrence of the transition will cause the changes of states from the states in IF part, L, to the states in the THEN part, L'".

The event-based rule has the syntax form shown below:

$$\begin{array}{ll} \textbf{IF} & L \\ \textbf{WHEN} & \text{Event X} \\ \textbf{THEN} & L' \end{array} \qquad (6.2)$$

Where L and L' have the same forms as those defined for the static rule. L provides the enabling condition immediately prior to, and at the time as, the event occurrence. The meaning of this rule construct can be stated as: "if all the local states in L are active when event X occurs then the system shall immediately change into the states in L' from the states in L as a result of the occurrence of the event". An immediate problem concerning the event-based rule is to determine what will happen if some local state in L is not active when the event occurs since this rule says nothing about the effect of the transition in this case. In this situation, the state transition will not occur. It is the designer's responsibility to guarantee that the event described in an event-based rule occurs after all states in L are active. Ambiguity can be avoided by defining an alternative event-based rule which specifies all alternatives for the occurrence of an event.

A static rule only describes *what* the state transition should be (i.e., asserts that a transition will happen) without stressing *when* the transition occurs which may depend on the run time support system such as the scheduler. However, the event-based rule is quite different because it integrates the event occurrence with an enabling condition, the combination of causing the state change. Thus an event-based rule describes not only what the state transition should be but also precisely *when* the transition should occur. However, the event-based rule still does not say anything about the time of event occurrence which is usually determined by the controlled system (rather than the controller) and is provided to the controller (software) via the sensors [Jaffe *et al* 91].

6.2.4 Decomposition of Rules

To keep the (software) functional requirements description manageable, a real-time process control system may be considered as a group of sub-systems where each sub-system description consists of a set of rules. A number of methods accommodate hierarchical structures which reflect the layers of software functional requirement such as Statecharts [Harel 87], Petri nets [Zhou and DiCesare 93], and G++ [Brams *et al* 92]. By adopting the sequential and parallel decomposition techniques from G++ [Brams *et al* 92] and applying it to the states in a rule, then rules can be decomposed using: *sequential* decomposition and *parallel* decomposition.

Sequential decomposition of a state in a rule describes the division of one state l into two sub-states l_1 and l_2 acting in sequence (one at a time) by dividing the action a associated with l into a sequential execution, denoted by $a_1;a_2$, where each a_i (i = 1,2) is associated with one sub-state. The implied semantics of sequential decomposition is that when action a is being performed in state l, the system is really either performing a_1 in state l_1 *or* a_2 in state l_2 but not both.

To decompose a state sequentially, a new event may need to be defined to trigger the sub-state transition. For example, suppose state l represents a machine moving with which the action — "rotation" is associated. After dividing this action into two parts "initial acceleration" and "maximum acceleration" and defining the event as the change of condition — "machine has rotated α degree since initial acceleration starts", then state l can be sequentially decomposed into l_1 and l_2. The state change between l_1 and l_2 is triggered by the occurrence of the event.

Parallel decomposition of a state in a rule describes the division of one state l into two independent sub-states l_1 and l_2 acting in parallel by dividing the action a into a parallel execution, denoted by $a_1 \| a_2$, of a_1 and a_2. The semantics behind the parallel decomposition is that when action a is being performed in state l, the system is performing both a_1 in state l_1 and a_2 in state l_2 at the same time.

To decompose a state parallelly, it does not need to introduce a new event as in sequential decomposition. The decomposed sub-states l_1 and l_2 from l in parallel decomposition will be changed by the same transition as defined for state l. For example, suppose state l represents a swimming state with which the action — "butter fly stroke" is associated. After dividing this action into two parts "arm's moving" and "leg's moving", then state l can be parallelly decomposed into l_1 and l_2 which are caused by the same transition of l.

6.3 A Method — System Behaviour Driven Method (SBDM)

Rules provide a suitable means of representing, rather than a method of constructing, the software functional requirements. From this observation, it may be concluded that a method is required to support the construction of a rule-based functional requirements description. Specifically the method should facilitate the processes of elicitation and formalisation. An obvious first step is to attempt to associate an existing requirements engineering method with the rule-based formalism, e.g. SREM [Alford 77, 85], SA/RT [Ward and Mellor 85], Statecharts [Harel 87], and hybrid method for Petri nets [Zhou and DiCesare 89, 93]. However, this is not practical because the methods were not based on rule-based paradigms and the entities such as states and events which underlie rule-based formalisms are not necessarily interpreted in the sense used throughout this thesis. Therefore, this thesis adopts the alternative approach of conducting a detailed examination of the rule-based representation scheme, with a view to answering the following: what are the basic concepts which must be elicited and what is the best order in which to elicit them ? The answers to these questions can be used to select techniques from a variety of existing methods, to elicit just those concepts that are necessary to build the rule-based functional requirements.

Goldsack and Finkelstein [91] proposed a method, called *Structured Common Sense* (SCS), to support the construction of a formal requirements specification for real-time systems, in which a logic called MAL (Modal Action Logic) was defined as the formal representation scheme. SCS method provides a systematic way to guide and organise the elicitation process for MAL. Since the requirements specifications in MAL may be viewed as state-based [Goldsack and Finkelstein 91] and the notations agents and actions defined in MAL have corresponding interpretation in rule-based formalism, it seems reasonable based on the Structured Common Sense method to propose a method to support the construction of a rule-based functional requirements description. The proposed method is called System Behaviour Driven Method (SBDM) which provides a systematic way to elicit the functional requirements using rule-based formalism.

Following SCS, if we look at the concepts on which the rule-based schemes are centred, then the following features can be observed:

- system components;
- actions associated with system components;
- states created by actions for each component;
- events to trigger actions;
- conditions or predicates to define the events.

To construct rule-based (software) functional requirements, we must answer questions such as:

1. what system components comprise the system being specified ?
2. what actions are associated with these components ?
3. why do the components undertake these actions ?
4. what states can each action create for each components ?
5. what are the events to trigger these actions ?
6. what are the definitions of these events ?
7. what are the relationships among these states ?
8. what are the timing constraints ?

The SBDM, a rule-based requirements capture method derived from SCS [Goldsack and Finkelstein 91], is an attempt to guide and organise the process by which these questions are answered. It consists of a number of distinct steps; some of which are performed in parallel and some sequentially. Progress through SBDM is driven by the behaviour of the real-time process control system.

The steps in SBDM are as follows:

- Component identification:
  The process of identifying each autonomous component within the system.
- Action analysis:
  Determining, for each autonomous component, the actions the system has to perform.
- Component state analysis:
  Determining the states of each autonomous component. The states can be divided into layers using hierarchical approach.
- Event identification:
  Identifying the events that trigger state transition and trigger action to occur.
- Data-flow analysis:
  Determining the links and interactions between the states and constructing the rule representations based on the 'causal forces' exerted by actions.
- Action decomposition:
  Identifying the new rules by decomposing the action.
- Timing analysis:
  Determining the timing constraints associated with each state by analysing its associated action.

It is easy to understand SBDM by looking at a sample step. The important steps of SBDM are action analysis and event identification. The primary purpose of the action analysis step is to support the designer in identifying the actions associated with each component, from which the states can be subsequently derived. Actions are the operations carried out by system within a state, which change the system behaviour. Everything that happens in the system must be in principle traceable to some action. The primary purpose of event identification step is to support the designer in identifying the events that will trigger the state transitions and stimulate the actions to occur, from which the event-based rules can be derived.

On completion of these steps, the designer should have:

- a set of system components;
- a set of actions for each component;
- a set of states for each component;
- a set of events;
- a set of rules.

The rules obtained based on SBDM provide a formal representation of system functional requirements, which can be used to check consistency and to verify system properties. The set of rules provides the synchronisation and control logic for the software control system being developed. The declarations of actions and events provide the interfaces to next level development, which can be further described in more details at later stage of the software system development.

6.4 Analysis of Rule-Based Description

6.4.1 Introduction

Formal techniques have widely been used for specification, design, implementation, and verification of real-time systems by researchers working on time-related logic's [Jahanian and Mok 86, Ostroff 89, Goldsack and Finkestein 91, Manna and Pnueli 92], real-time process algebra [Milner 89, Reed and Roscoe 86], and Petri nets [Ghezzi *et al* 91]. The advantage of formal techniques is that formal description allows one to reason about the intended behaviour of the system. Among these techniques, temporal logic has been increasingly used in verification of real-time systems [Manna and Pnueli 88, 92, Ostroff 89] in which the system is portrayed as a logical model, and the specification is represented as logical formulas. If a formula is true in the model, then the system model is assumed to be correct.

In order to use a particular analysis technique, such as Petri nets, one needs to transform the functional requirement description in terms of rules into the appropriate representation that the technique will accept. Each rule describes only the intended behaviour of one component in the system. Since a system consists of many components and each of them may be dependent on others, most system behaviour can be described only via the co-operation of these individual rules. By integrating these individual rules together according to the causal dependency among them, properties of the composed system can be analysed. Thus, in the case of Petri nets, system properties such as mutual exclusion and deadlock can be verified. Also, after translating a rule-based functional requirement into a Petri net, the reachability graph of Petri nets can be explored using temporal operators [Suzuki and Lu 89, He and Lee 90, Sagoo and Holding 90, 91]. If the system properties expressed in temporal formulas are determined to be true, then the properties hold in the Petri net model and also in the (rule-based) functional requirements. Although it may be possible for rules to be transformed into a number of different state transition models, such as extended finite state machines (ESM) [Ostroff 89], CTL machine [Clarke *et al* 86], Statecharts [Harel 87] and synchronous model [Benveniste and Berry 91], a Petri net model is chosen as the analysis model because of following reasons:

i) There is a direct correspondence between rule structure and Petri nets [Atabakhche *et al* 86, Sahraoui *et al* 87, Wilson and Krogh 90, Etessami and Hura 91].
ii) Petri net framework provides a graphical representation in which the co-operation of all the individual rules can be represented explicitly.
iii) Petri net theory [Reisig 85, Rozenberg and Thiagarajan 87, Murata 89, Mazurkiewicz 87] provides the ability to analyse the graphical representation.
iv) The ability to translate Petri nets to temporal logic for formal proof of system properties has been shown by researchers [Reisig 88a, 88b, Suzuki and Lu 89, He and Lee 90, Uchihira and Honiden 90, Sagoo and Holding 90, 91, Sagoo 92].
v) After formally verifying the system properties, the Petri net model can easily be transformed to SFC and its associated programming environment for detailed design and implementation.

6.4.2 Modelling Rules as XTPTN

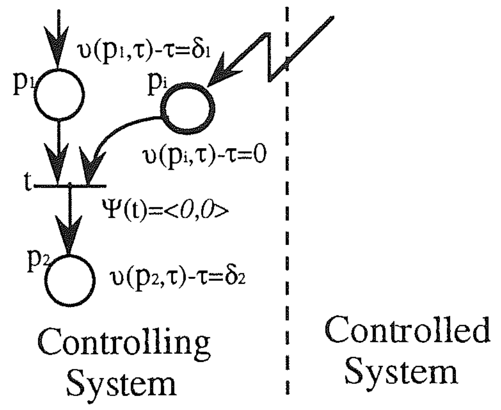
The type of XTPTN considered in this thesis for modelling rule-based descriptions is the XTPTN with capacity $K(p) = 1$ for each place p (see page 30). It is natural to consider the upper limit to the number of tokens that each place can hold to be 1 because each local state defined in rules will be modelled by a place in XTPTN and it may not make sense if a place holds more than one token.

It is assumed that all actions to be assigned to system components in rules have fixed durations. To represent the rule-based description explicitly using XTPTN, one has to map the elements of the rule structures onto the elements of the XTPTN structure and keep the mapping functional equivalent.

The static rule can be naturally modelled by XTPTN using following method. Each local state in the rule is modelled by a place $p \in P$ of XTPTN and the properties (prepositional values — *active* and *inactive*) defined for local states are modelled by a place with or without a token. The duration of the action to be assigned to the component by each local state in the rule is modelled by the place's unavailable time. Each state transition of a rule is modelled by an XTPTN transition with firable interval $<0, \infty>$. Such a mapping keeps the functional equivalence because the state transition of a static rule can occur at any instant after all the states in the IF part become active is preserved in XTPTN by the timing constraint $<0, \infty>$.

To model an event-based rule, the effect of the event on the state transition must be considered. The set of interface places $\mathcal{I}$ and the function Ψ in XTPTN provide a convenient way to model this special feature. Each event e in an event-based rules can be modelled by an interface place $p_e \in \mathcal{I}$. If place p_e is tokenised (which needs to be set externally), then it means that event e occurs, otherwise it means that event e has not occurred yet. States and state transitions of event-based rules can be modelled by XTPTN using the same method used for static rules. The instantaneous effect that event occurrence acts upon the state transition in event-based rules can be modelled by defining the firable interval of transition as $<0, 0>$ via function Ψ. The Fig.6.1 is an example of XTPTN model for an event-based rule. This technique will be used to systematically generate the XTPTN models for industrial manufacturing machineries in Chapter 7.

IF $\mathcal{L}_1<\boldsymbol{a}_1, comp_1>$
WHEN Event-*e*
THEN $\mathcal{L}'_1<\boldsymbol{a}'_1, comp'_1>$
where: $\mathcal{L}_1$, $\mathcal{L}'_1$ are local states;
$\boldsymbol{a}_1$, $\boldsymbol{a}'_1$ are actions;
$\boldsymbol{comp}_1$, $\boldsymbol{comp}'_1$ are components;
Duration($\boldsymbol{a}_1$) = δ_1;
Duration($\boldsymbol{a}_2$) = δ_2;



$\upsilon(p_1,\tau)-\tau=\delta_1$
p_1
p_i
$\upsilon(p_i,\tau)-\tau=0$
t
$\Psi(t)=<0,0>$
p_2
$\upsilon(p_2,\tau)-\tau=\delta_2$
Controlling System
Controlled System

Fig.6.1 Modelling an event-based rule in XTPTN

6.4.3 Verification

6.4.3.1 Verification of Rule-Based Description

In the requirement document, the user typically asserts a set of properties, such as functional and general safety assertions, that must be enforced. However, there is no guarantee that the rules representing the behaviour of the actual system satisfy these properties because the translation from the real world into a rule-based description is informal, intuitive and heuristic. Whether or not the rules accurately represent all the important facets of the system requirement depends upon the verification. Temporal Petri nets [Suzuki and Lu 89, He and Lee 90, Sagoo and Holding 90, 91, Sagoo 92] provides the ability to formally specify and verify the Petri net properties using temporal logic. In a temporal Petri net, system properties are specified as temporal logic formulas and the Petri net model is translated into a temporal logic system. The specified properties are then verified over the translated temporal logic systems using developed temporal logic proof techniques [He and Lee 90, Manna and Pnueli 92]. If a property, say φ, is true over all state sequences generated by the translated temporal logic system, then φ is called a valid property which is also valid over the Petri net model. It has been shown that rule-based descriptions can be transformed into a XTPTN with capacity $K(p)=1$ for each place p, which provides a Petri net based computational model. If the required system properties are qualitative rather than quantitative, then it seems reasonable to translate the XTPTN into temporal logic systems using the methods proposed by He and Lee [90], Sagoo and Holding [90, 91, Sagoo 92], and to verify these properties using the developed temporal Petri net proof techniques. The next section will discuss how the XTPTN obtained from rules can be translated into a temporal logic system using these methods.

6.4.3.2 Verification Technique — Temporal Petri Nets

In this section, the translation from XTPTN obtained from rules into a temporal logic system using the transform method proposed by He and Lee [90] and modified by Sagoo and Holding [91, Sagoo 92] will be presented. If this can be done, then the required system properties can be investigated by expressing them in formulas containing temporal logic operators and showing that a formula is logically valid over the XTPTN model using temporal Petri net proof techniques.

The temporal logic systems translated from Petri nets consists of two parts: system independent temporal logic axioms and inference rules such as the axioms and inference rules developed by Manna and Penuli [86, 88, 92], and system dependent axioms and inference rules [He and Lee 90, Sagoo and Holding 91]. The system independent temporal logic axioms and inference rules are independent of any specific Petri net systems. However, the system dependent axioms and inference rules differ from one Petri net to another. According to the method for low level C/E nets (Petri nets with 1 capacity for each place) [Sagoo and Holding 91] derived from the transform method of He and Lee [90], the initial marking of Petri nets is converted into system dependent axioms and each transition of Petri nets is converted into an system dependent inference rule which defines the firing of the transition in terms of pre- and post-conditions. The following is the transformation from an XTPTN to a temporal logic system.

Temporal Formulas over XTPTN:

Let N = (PN, Υ, υ, ι, C, Ψ, $TURT_0$) be a XTPTN, where PN = (P, T, F, M_0). Then the syntax of propositional temporal formulas over XTPTN is defined as:

1) A formula is built from
   i) atomic proposition: (p *has a token*), where $p \in$ P;
   ii) Boolean connectives: $\wedge$ (AND), $\vee$ (OR), $\neg$ (NOT), and $\rightarrow$ (implication);
   iii) temporal operators: $\bigcirc$ (next), $\Box$ (henceforth), $\Diamond$ (eventually), and $\mathcal{U}$ (until).

2) The formation rules are:
   i) an atomic proposition is a formula;
   ii) If g, h are formulas then so are $g \wedge h$, $g \vee h$, $\neg g$, $g \rightarrow h$, $\bigcirc g$, $\Box g$, $\Diamond g$, and $g \mathcal{U} h$

In the following discussion, atomic proposition (p *has a token*) will be abbreviated as p.

Semantics of Temporal Formulas:

Let $\Gamma = M_0 \ldots M_i M_{i+1} \ldots$ be an execution marking sequence which is a member of the relation $[M_0>^*$ (see the definition given in Section 5.1 of Chapter 5) and $\Gamma^{(k)}$ be a k-shifted marking sequence given by $M_k M_{k+1} \ldots$. Let f be a temporal formula defined above. The formal semantics of formula f is defined as follows:

a) For the propositional operators:

   i) for (p has a token), $\Gamma \models$ (p has a token) iff p has one token at M_0 (6.3)
   ii) $\Gamma \models \neg f$ iff not $\Gamma \models f$ (6.4)
   iii) $\Gamma \models f \wedge g$ iff $\Gamma \models f$ and $\Gamma \models g$ (6.5)
   iv) $f \vee g$ and $f \rightarrow g$ are the short hands for $\neg(\neg f \wedge \neg g)$ and $\neg f \vee g$ (6.6)

b) For the temporal operators:

   i) $\Gamma \models \Box f$ *iff* for every $0 \leq i \leq |\Gamma|$, $\Gamma^{(i)} \models f$
   where $|\Gamma|$ represents the length of execution marking sequence Γ (6.7)
   ii) $\Gamma \models \Diamond f$ *iff* for some $0 \leq i \leq |\Gamma|$, $\Gamma^{(i)} \models f$ (6.8)
   iii) $\Gamma \models \bigcirc f$ *iff* $|\Gamma| > 1$ and $\Gamma^{(1)} \models f$ (6.9)
   iv) $\Gamma \models f \mathcal{U} g$ *iff* $\Gamma^{(i)} \models g$ for some $0 \leq i \leq |\Gamma|$, and $\Gamma^{(j)} \models f$ for every $0 \leq j < i$ (6.10)

System Dependent Axioms:

Let N = (PN, Υ, υ, ι, $\mathfrak{C}$, Ψ, TURT$_0$) be a XTPTN, where PN = (P, T, F, M_0). The initial marking $M_0 = \{p_1, p_2, \ldots, p_n\}$ can be expressed as a temporal formula as $p_1 \wedge p_2 \wedge \ldots \wedge p_n$ which is referred to as the system dependent axiom.

System Dependent Inference Rules:

Each transition in the XTPTN will be converted into an inference rule which defines the firing of the transition in terms of system-dependent pre and post conditions [Sagoo and Holding 91]. For instance an inference rule for transition t has the form $\mathrm{U} \Rightarrow \bigcirc \mathrm{U}'$, where U contains a formula comprising the conjunction of all the places in $\bullet t$ and the conjunction of the negation of all the places in $(t\bullet - \bullet t)$ the U' contains a formula which is symmetric to U.

Summary

In this chapter, a rule-based formalism has been presented as a means of describing the functional requirements of a real-time process control system. Two different rule schemes (static and event-based) and an elicitation method for the functional requirements have been proposed. In order to formally analyse the rule-based descriptions, the method to translate rules into XTPTN and the transformation from XTPTN to temporal logic system have been presented.

Chapter 7 Examples and their Evaluations

7.1 Introduction

Chapter 6 presented a rule-based development method and an analysis technique. In this chapter, the application of the rule-based formalism together with the proposed SBDM method will be demonstrated through examples. These examples illustrate in a step by step manner how the functional requirements can be elicited by the SBDM method and represented in terms of a set of rules. In addition, it is shown how system properties can be verified with respect to the rule-based functional requirements via the Petri net theory and temporal Petri net techniques.

7.2 First Example — A Can Sorting Machine

7.2.1 Introduction

Consider the demonstration can sorting machine developed by Eurotherm Controls Ltd., shown in Fig.7.1, in which two different sizes of cans have to pass through two drums before being put on the conveyor. The two drums are driven by two independently controlled motors and need to be co-ordinated when the can is exchanged from drum_1 to drum_2. The two drums being controlled will provide external stimulus when the operations associated with them terminate. In this chapter, it is assumed that all the external stimulus will occur properly when the operations associated with the physical system terminate. In the following, the method presented in last chapter will be used to derive the rules representing the synchronisation and control logic of the two drums.

7.2.2 Eliciting the Rules Using SBDM

A. Identifying the autonomous components within the system:

1. drum1; 2. drum2; 3. conveyor; 4. can producer (including feeding);

B. Identifying the actions associated with each component:

1. drum1 = {can_insert, rotation1, can_exchange}
2. drum2 = {can_exchange, rotation2, can_eject}
3. conveyor = {running}   4. can producer = {producing}

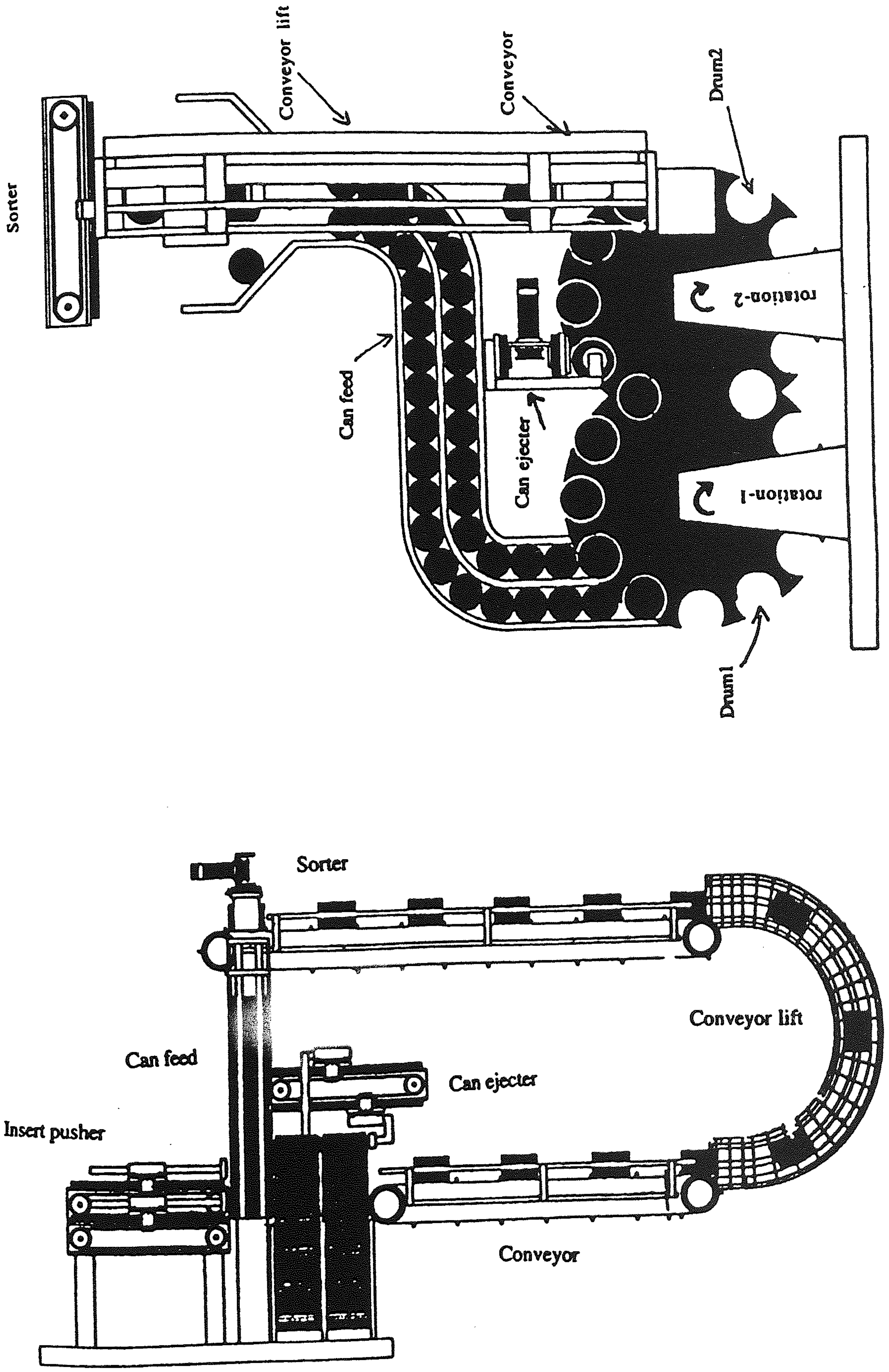


Fig.7.1 Can sorting physical system model

C. Identifying the states created by each action within each component:

Standardised mnemonics for drums:

S = space; C = can; P = position; INS = insert; EXC = exchange; EJE = eject;

1. Drum1:

   First layer

   drum1 = {C_INS_D1, D1_ROT, C_EXC, D1_STOP}

where	C_INS_D1	*insert* can from can feed into drum1
	D1_ROT	*rotate* drum1
	C_EXC	*exchange* can from drum1 to drum2
	D1_STOP	drum1 is stationary

   Second layer

   D1_STOP = {C_IN_D1, C_OUT_D1}

where	C_IN_D1	can is in drum1
	C_OUT_D1	can is not in drum1

   D1_ROT = {D1_C_IN_ROT, D1_C_OUT_ROT}

where	D1_C_IN_ROT	*rotate* drum1 AND can is in drum1
	D1_C_OUT_ROT	*rotate* drum1 AND can is not in drum1

   Third layer

   C_IN_D1 = {C_P_INS_D1, C_P_EXC_D1}

where	C_P_INS_D1	can is in drum1 AND drum1 is in insert position
	C_P_EXC_D1	can is in drum1 AND drum1 is in exchange position

   C_OUT_D1 = {S_P_INS_D1, S_P_EXC_D1}

where	S_P_INS_D1	drum1 is in insert position AND space is available in it
	S_P_EXC_D1	drum1 is in exchange position AND space is available in it

2. Drum2:

   First layer

   drum2 = {C_EXC, D2_ROT, D2_STOP, C_EJE}

where	C_EXC	*exchange* can from drum1 to drum2
	D2_ROT	*rotate* drum2
	C_EJE	*eject* can from drum2 onto conveyor
	D2_STOP	drum2 is stationary

   Second layer

   D2_STOP = {C_IN_D2, C_OUT_D2}

where	C_IN_D2	can is in drum2
	C_OUT_D2	can is not in drum2

   D2_ROT = {D2_C_IN_ROT, D2_C_OUT_ROT}

where	D2_C_IN_ROT	*rotate* drum2 AND can is in drum2
	D2_C_OUT_ROT	*rotate* drum2 AND can is not in drum2

Third layer

C_IN_D2 = {C_P_EXC_D2, C_P_EJE_D2}

where C_P_EXC_D2 can is in drum2 AND drum2 is in exchange position

C_P_EJE_D2 can is in drum2 AND drum2 is in eject position

C_OUT_D2 = {S_P_EXC_D2, S_P_EJE_D2}

where S_P_EXC_D2 drum2 is in exchange position AND space is available in it

S_P_EJE_D2 drum2 is in eject position AND space is available in it

3. Conveyor:

First layer

conveyor = {RUNNING}

where RUNNING conveyor is running

Second layer

RUNNING = {C_PRE_CO, S_AVA_CO}

where C_PRE_CO can is present on conveyor

S_AVA_CO space is available on conveyor

4. Can producer:

First layer

can producer = {C_PROD, P_STOP}

where C_PROD *produce* a can

P_STOP producer is rest

Second layer

P_STOP = {C_AVA_P, S_AVA_P}

where C_AVA_PRO can is available from producer

S_AVA_PRO space is available in producer

<u>Events which cause a state transformation:</u>

Drum1:	{C_inserted,drum1_has_rotated_with_can,drum1_has_rotated_without_can}
Drum2:	{C_ejected, drum2_has_rotated_with_can, drum2_has_rotated_without_can}
Drum1&2:	{C_exchanged} /* can has been exchanged from drum1 to drum2 */
Conveyor:	{C_conveyed} /* can has been conveyed */
Can_Producer:	{C_produced} /* can has been produced */

Following are the rule-based descriptions for the functional requirements based on he elements defined above:

Synchronisation between can producer and drum1

```
R1:     IF          C_AVA_PRO AND S_P_INS_D1
        THEN        C_INS_D1
text    IF "can is available from producer" AND
           "drum1 is in insert position AND space is available in drum1"
        THEN "insert can from producer into drum1"

R2:     IF          C_INS_D1
        WHEN        C_inserted
        THEN        C_P_INS_D1 AND S_AVA_PRO
text    IF "insert  can from producer into drum1"
        WHEN C_inserted
        THEN "drum1 is in insert position AND can is in drum1" AND
                "space is available in producer"
```

Synchronisation between conveyor and drum2

```
R3:     IF          C_P_EJE_D2 AND S_AVA_CO
        THEN        C_EJE
text    IF "drum2 is in eject position AND can is in drum2" AND "space is available on
        conveyor"
        THEN "eject can from drum2 onto conveyor"

R4:     IF          C_EJE
        WHEN        C_ejected
        THEN        S_P_EJE_D2 AND C_PRE_CO
text    IF "eject  can from drum2 onto conveyor"
        WHEN C_ejected
        THEN "space is available in drum2 AND drum2 is in eject position" AND
                "can is present on conveyor"
```

Synchronisation between drum1 and drum2:

```
R5:     IF          C_P_EXC_D1 AND S_P_EXC_D2
        THEN        C_EXC
text    IF "can is in drum1 AND drum1 is in exchange position" AND
           "space is available in drum2 AND drum2 is in exchange position"
        THEN "exchange  can from drum1 to drum2"

R6:     IF          C_EXC
        WHEN        C_exchanged
        THEN        S_P_EXC_D1 AND  C_P_EXC_D2
```

```
text    IF "exchange can from drum1 to drum2"
        WHEN C_exchanged
        THEN "space is available in drum1 AND drum1 is in exchange position" AND
                "can is in drum2 AND drum2 is in exchange position"

Can_producer (rules to account for full producer cycle):
R7:     IF          S_AVA_PRO
        THEN        C_PROD
text    IF "space is available in producer"
        THEN "produce a can"

R8:     IF          C_PROD
        WHEN        C_produced
        THEN        C_AVA_PRO
text    IF "produce a can"
        WHEN C_produced
        THEN "can is available from producer"

Drum1 (rules to account for full drum1 cycle):
R9:     IF          C_P_INS_D1
        THEN        D1_C_IN_ROT
text    IF "can is in drum1 AND drum1 is in insert position"
        THEN "rotate drum1 AND can is in drum1"

R10:    IF          D1_C_IN_ROT
        WHEN        drum1_has_rotated_with_can
        THEN        C_P_EXC_D1
text    IF "rotate drum1 AND can is in drum1"
        WHEN drum1_has_rotated_with_can
        THEN "can is in drum1 AND drum1 is in exchange position"

R11:    IF          S_P_EXC_D1
        THEN        D1_C_OUT_ROT
text    IF "space is available in drum1 AND drum1 is in exchange position"
        THEN "rotate drum1 AND can is not in drum1"

R12:    IF          D1_C_OUT_ROT
        WHEN        drum1_has_rotated_without_can
        THEN        S_P_INS_D1
```

text IF "*rotate* drum1 AND can is not in drum1"
WHEN drum1_has_rotated_without_can
THEN "space is available in drum1 AND drum1 is in insert position"

Conveyor (rules to account for full sorter and conveyor cycle):

R13: IF C_PRE_CO
WHEN C_conveyed
THEN S_AVA_CO

text IF "can is present on conveyor"
WHEN C_conveyed
THEN "space is available on conveyor"

Drum2 (rules to account for full drum2 cycle):

R14: IF C_P_EXC_D1
THEN D2_C_IN_ROT

text IF "can is in drum2 AND drum2 is in exchange position"
THEN "*rotate* drum2 AND can is in drum2"

R15: IF D2_C_IN_ROT
WHEN drum2_has_rotated_with_can
THEN C_P_EJE_D2

text IF "*rotate* drum2 AND can is in drum2"
WHEN drum2_has_rotated_with_can
THEN "can is in drum2 AND drum2 is in eject position"

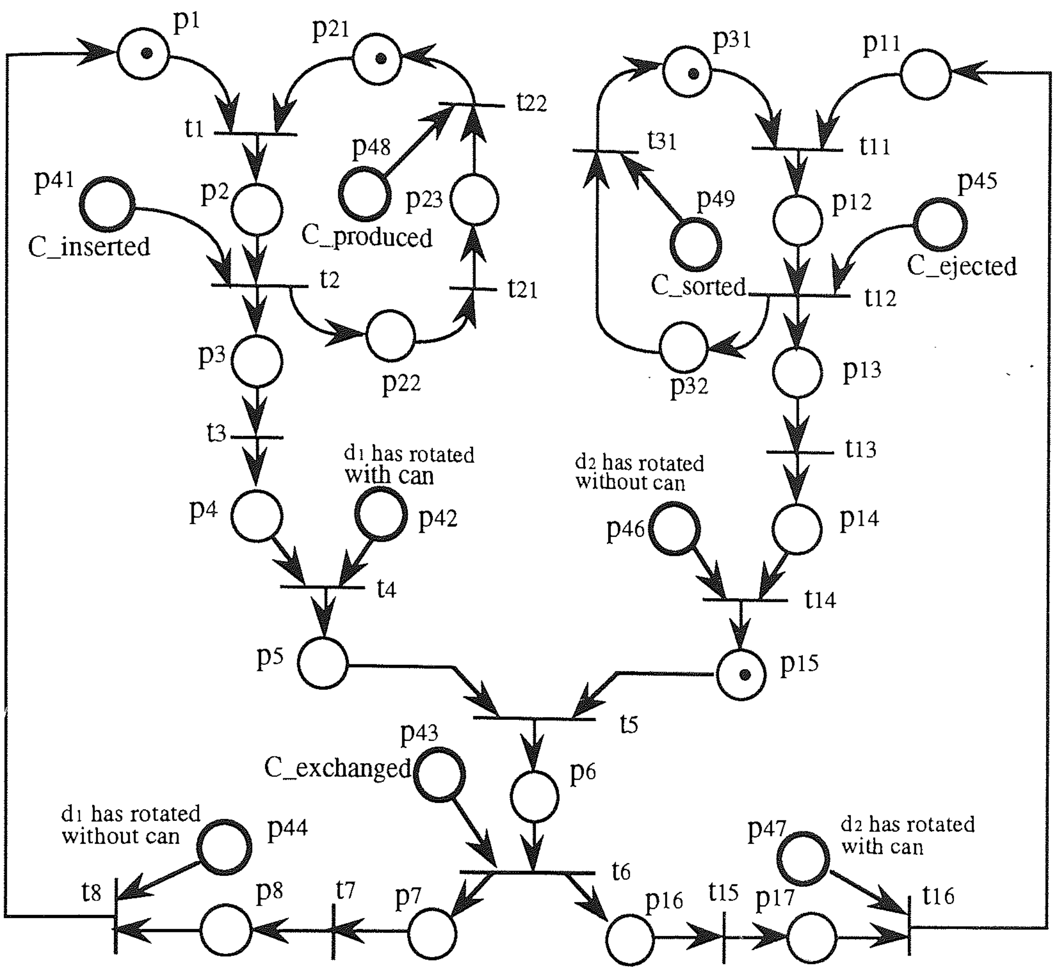
R16: IF S_P_EJE_D2
THEN D2_C_OUT_ROT

text IF "space is available in drum2 AND drum2 is in eject position"
THEN "*rotate* drum2 AND can is not in drum2"

R17: IF D2_C_OUT_ROT
WHEN drum2_has_rotated_without_can
THEN S_P_EXC_D2

text IF "*rotate* drum2 AND can is not in drum2"
WHEN drum2_has_rotated_without_can
THEN "space is available in drum2 AND drum2 is in exchange position"

An XTPTN model of the rule descriptions is given below. Note that the duration for each place, the enabling interval for each transition are not shown in the model because they are not relevant to the qualitative verification of properties.



p_1 — S_P_INS_D1;	p_{11} — C_P_EJE_D2;	p_{21} — C_AVA_PRO;
p_2 — C_INS_D1;	p_{12} — C_EJE;	p_{22} — S_AVA_PRO;
p_3 — C_P_INS_D1;	p_{13} — S_P_EJE_D2;	p_{23} — C_PROD;
p_4 — D1_C_IN_ROT;	p_{14} — D2_C_OUT_ROT;	p_{31} — S_AVA_CO;
p_5 — C_P_EXC_D1;	p_{15} — S_P_EXC_D2;	p_{32} — C_PRE_CO;
p_6 — C_EXC;	p_{16} — C_P_EXC_D2;	p_{45} — C_ejected;
p_7 — S_P_EXC_D1;	p_{17} — D2_C_IN_ROT;	p_{48} — C_produced;
p_8 — D1_C_OUT_ROT;	p_{43} — C_exchanged;	p_{49} — C_conveyed;
p_{41} — C_inserted;	p_{42} — drum1_has_rotated_with_can;	

p_{44} — drum1_has_rotated_without_can; p_{46} — drum2_has_rotated_without_can;

p_{47} — drum2_has_rotated_with_can;

Fig.7.2 XTPTN representation of rules for the can sorting system

7.2.3 Properties of the system

For any real-time process control systems, the essential concern is that the system should behave in a safe and acceptable fashion. Thus the system must be verified for correctness. An important corollary is that a specification of the system, which refers to the states, events and properties of the controlled systems, must be correct. The following are few typical properties for the system asserted by the user in the requirement documents:

i) The situation where a can is exchanging from drum1 to drum2 and meanwhile drum1 or drum2 is rotating must not happen. To verify this property it is necessary to show that the "rotate" states of drum1 or drum2 are mutually exclusive with "exchange" state. In terms of XTPTN, this can be expressed as invariances in temporal formulas as:

$$\Box\neg (p_4 \wedge p_6) \tag{7.1}$$
$$\Box\neg (p_8 \wedge p_6) \tag{7.2}$$
$$\Box\neg (p_{17} \wedge p_6) \tag{7.3}$$
$$\Box\neg (p_{14} \wedge p_6) \tag{7.4}$$

ii) Whenever a can is inserted into drum1 from producer then eventually it must be exchanged from drum1 into drum2. This can be expressed as a repeated eventuality property in temporal formula as:

$$\Box (p_3 \rightarrow \Diamond p_6) \tag{7.5}$$

iii) Whenever a can is exchanged from drum1 into drum2 then it must eventually be ejected onto conveyor. This property can be expressed as eventuality in temporal formula as:

$$\Box(p_6 \rightarrow \Diamond p_{11}) \tag{7.6}$$

iv) Whenever a can is inserted from producer then eventually a can will be produced again in producer. This property can be expressed as formula in temporal logic as:

$$\Box (p_3 \rightarrow \Diamond p_{21}) \tag{7.7}$$

v) The independent motion behaviour of both drums is permitted. In another words, the rotating states of drum1 and drum2 are not mutually exclusive. This property can be expressed as formulas in temporal logic as:

$$\neg\Box\neg (p_8 \wedge p_{17}) \tag{7.8}$$
$$\neg\Box\neg (p_8 \wedge p_{14}) \tag{7.9}$$
$$\neg\Box\neg (p_4 \wedge p_{17}) \tag{7.10}$$
$$\neg\Box\neg (p_4 \wedge p_{14}) \tag{7.11}$$

7.2.4 Verification of the system properties

Given a formal mathematical model and a specification of what the system should behave, the verification problem involves the formal demonstration by a mathematical proof that the model satisfies the specification.

For the net shown in Fig.7.2, the following system dependent axiom and inference rules can be obtained according to the combination between temporal logic and Petri nets discussed in last chapter:

System dependent axiom: $p_1 \wedge p_{15} \wedge p_{21} \wedge p_{31}$

System dependent inference rules:

1. $p_1 \wedge p_{21} \wedge \neg p_2 \Rightarrow \bigcirc (p_2 \wedge \neg p_1 \wedge \neg p_{21})$
2. $p_2 \wedge p_{41} \wedge \neg p_3 \wedge \neg p_{22} \Rightarrow \bigcirc (p_3 \wedge p_{22} \wedge \neg p_2 \wedge \neg p_{41})$
3. $p_3 \wedge \neg p_4 \Rightarrow \bigcirc (p_4 \wedge \neg p_3)$
4. $p_4 \wedge p_{42} \wedge \neg p_5 \Rightarrow \bigcirc (p_5 \wedge \neg p_4 \wedge \neg p_{42})$
5. $p_5 \wedge p_{15} \wedge \neg p_6 \Rightarrow \bigcirc (p_6 \wedge \neg p_5 \wedge \neg p_{15})$
6. $p_6 \wedge p_{43} \wedge \neg p_7 \wedge \neg p_{16} \Rightarrow \bigcirc (p_7 \wedge p_{16} \wedge \neg p_6 \wedge \neg p_{43})$
7. $p_7 \wedge \neg p_8 \Rightarrow \bigcirc (p_8 \wedge \neg p_7)$
8. $p_8 \wedge p_{44} \wedge \neg p_1 \Rightarrow \bigcirc (p_1 \wedge \neg p_8 \wedge \neg p_{44})$
9. $p_{11} \wedge p_{31} \wedge \neg p_{12} \Rightarrow \bigcirc (p_{12} \wedge \neg p_{11} \wedge \neg p_{31})$
10. $p_{12} \wedge p_{45} \wedge \neg p_{13} \wedge \neg p_{32} \Rightarrow \bigcirc (p_{13} \wedge p_{32} \wedge \neg p_{12} \wedge \neg p_{45})$
11. $p_{13} \wedge \neg p_{14} \Rightarrow \bigcirc (p_{14} \wedge \neg p_{13})$
12. $p_{14} \wedge p_{46} \wedge \neg p_{15} \Rightarrow \bigcirc (p_{15} \wedge \neg p_{14} \wedge \neg p_{46})$
13. $p_{16} \wedge \neg p_{17} \Rightarrow \bigcirc (p_{17} \wedge \neg p_{16})$
14. $p_{17} \wedge p_{47} \wedge \neg p_{11} \Rightarrow \bigcirc (p_{11} \wedge \neg p_{17} \wedge \neg p_{47})$
15. $p_{22} \wedge \neg p_{23} \Rightarrow \bigcirc (p_{23} \wedge \neg p_{22})$
16. $p_{23} \wedge p_{48} \wedge \neg p_{21} \Rightarrow \bigcirc (p_{21} \wedge \neg p_{23} \wedge \neg p_{48})$
17. $p_{32} \wedge p_{49} \wedge \neg p_{31} \Rightarrow \bigcirc (p_{31} \wedge \neg p_{32} \wedge \neg p_{49})$

Following are the proofs of those properties specified in last section. Refutation proof technique and the token invariance theorem presented in Chapter 2 are used in the proofs. A refutation proof of a formula p is a syntactical derivation of a sequence of formulas F_0, F_1,..., F_n such that $F_0 = \neg p$, F_n = false, and F_{i+1} is derived based on F_0 to F_i by one of the inference rules.

Proof of Property (7.1) $\Box\neg(p_4 \wedge p_6)$

(1)	$\neg\Box\neg(p_4 \wedge p_6)$	from property (7.1) by negation
(2)	$\Diamond(p_4 \wedge p_6)$	from (1) by axiom $\Diamond\varphi \Leftrightarrow \neg\Box\neg\varphi$

(3) The loop $C = \{p_1,t_1,p_2,t_2,p_3,t_3,p_4,t_4,p_5,t_5,p_6,t_6,p_7,t_7,p_8,t_8\}$ shown in Fig.7.2 has the property that the number of tokens in C remains invariant during all the executions of the net because C satisfies following requirements:

i) $\forall t \in C, |\bullet t \cap C| = |t\bullet \cap C|$.

ii) $\forall p \in C, (\bullet p \cup p\bullet) \subseteq C$.

Since there is only one token in C at the initial marking, this shows that mutual exclusion in C is guaranteed. In terms of temporal Petri nets, the mutual exclusion between places p_4 and p_6 can be expressed in temporal logic formula as [Suzuki and Lu 89]:

(4) $\Box(p_4 \rightarrow \neg p_6) \vee \Box(p_6 \rightarrow \neg p_4)$

Examining each disjunct of (4) gives:

(5)	$\Box(p_4 \rightarrow \neg p_6)$	first disjunct of (4)
(6)	$\Box\neg(p_4 \wedge p_6)$	by definitions of logical operators $\neg,\vee,\rightarrow$
(7)	$\Box(p_6 \rightarrow \neg p_4)$	second disjunct of (4)
(8)	$\Box\neg(p_4 \wedge p_6)$	by definitions of logical operators $\neg,\vee,\rightarrow$
(9)	$\neg\Diamond(p_4 \wedge p_6)$	from (6), (8) by axiom $\Diamond\varphi \Leftrightarrow \neg\Box\neg\varphi$

In both cases the conclusion (9) is contradicted with (2) obtained by negation, which shows the assumption false.

A similar procedure can be used to prove properties (7.2), (7.3), and (7.4).

Proof of property (7.5) $\Box(p_3 \rightarrow \Diamond p_6)$

(1)	$p_1 \wedge p_{15} \wedge p_{21} \wedge p_{31}$	by system dependent axiom
(2)	$p_1 \wedge p_{21} \wedge \neg p_2 \Rightarrow \bigcirc(p_2 \wedge \neg p_1 \wedge \neg p_{21})$	from (1) by inference rule-1

the occurrence of event C_inserted (i.e. p_{41} is tokenised) yields

(3)	$p_2 \wedge p_{41} \wedge \neg p_3 \wedge \neg p_{22} \Rightarrow \bigcirc(p_3 \wedge p_{22} \wedge \neg p_2 \wedge \neg p_{41})$	from (2) by inference rule-2
(4)	$p_3 \wedge \neg p_4 \Rightarrow \bigcirc(p_4 \wedge \neg p_3)$	from (3) by inference rule-3

the occurrence of event d_1_has_rotated_with_can (i.e. place p_{42} is tokenised) yields

(5)	$p_4 \wedge p_{42} \wedge \neg p_5 \Rightarrow \bigcirc(p_5 \wedge \neg p_4 \wedge \neg p_{42})$	from (4) by inference rule-4
(6)	$p_5 \wedge p_{15} \wedge \neg p_6 \Rightarrow \bigcirc(p_6 \wedge \neg p_5 \wedge \neg p_{15})$	from (1), (5) by inference rule-5
(7)	$p_3 \rightarrow \Diamond p_6$	from (3) to (5) by tautology
(8)	$\Box(p_3 \rightarrow \Diamond p_6)$	from (7) by temporal reasoning

A similar procedure can be used to prove properties (7.6) and (7.7).

Proof of property (7.8) $\neg\Box\neg(p_8 \wedge p_{17})$

To make the proof shorter, the immediate results of proof for property (7.5) will be used.

(1)	$\Box\neg(p_8 \wedge p_{17})$	by negating the property (7.8)
(2)	$\neg\Diamond(p_8 \wedge p_{17})$	from (1) by axiom $\Box\varphi \Leftrightarrow \neg\Diamond\neg\varphi$

(3) By examining step (1) to step (6) in the proof of property (7.5), we have:
$p_6 \wedge p_{22} \wedge p_{31}$
the occurrence of event C_exchanged (i.e. p_{43} is tokenised) yields

(4)	$p_6 \wedge p_{43} \wedge \neg p_7 \wedge \neg p_{16} \Rightarrow \bigcirc(p_7 \wedge p_{16} \wedge \neg p_6 \wedge \neg p_{43})$	from (3) by inference rule-6
(5)	$p_7 \wedge \neg p_8 \Rightarrow \bigcirc(p_8 \wedge \neg p_7)$	from (4) by inference rule-7
(6)	$p_{16} \wedge \neg p_{17} \Rightarrow \bigcirc(p_{17} \wedge \neg p_{16})$	from (4) by inference rule-13
(7)	$p_8 \wedge p_{17}$	from (5) and (6)
(8)	$\Diamond(p_8 \wedge p_{17})$	from (7) by temporal reasoning

This conclusion is contradicted with (2) which shows that the assumption by negation is false.

A similar procedure can be used to prove properties (7.9), (7.10), and (7.11).

7.3 Second Example — A Slider and Drum System

7.3.1 Introduction

The rule-based formalism and the design method proposed for the software functional requirement capture will be applied to a time-critical real-time process control system which consists of an incremental arbor drum and an intermittent transfer slider as shown in Fig.7.3 [Sagoo and Holding 90].

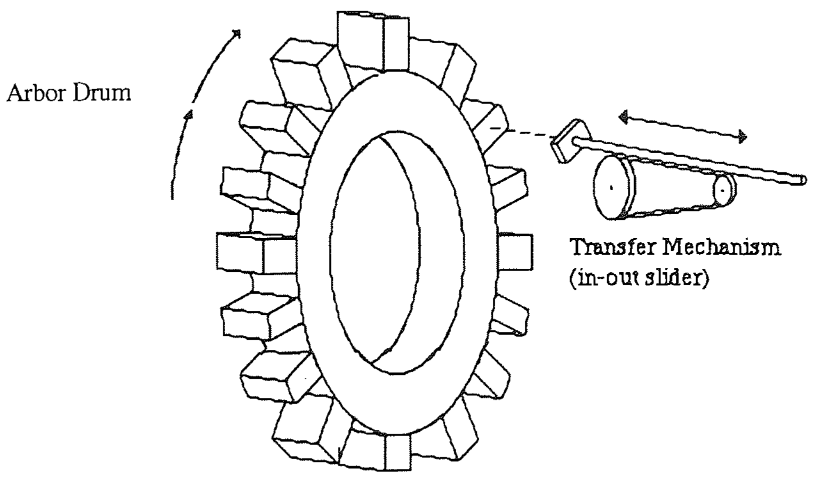


Fig.7.3 Arbor drum and transfer slider

The principal role of the controlling system (or controller) is to ensure the intermittent synchronisation of the slider and the drum. Asynchronous concurrent motion of the drum and slider is permitted. Clearly, the drum must be at rest and in position before the slider is inserted into it. Similarly, the slider must be withdrawn from the drum before the drum can rotate. The critical point in the slider's motion occurs when the slider is moving towards the drum and a decision has to be taken either to continue moving and insert in the drum (if it is stationary and is in position) or to decelerate (abort) and stop (if it is rotating or not in position)*. This safety-critical decision must be computed in a timely manner, as shown in Fig.7.4.

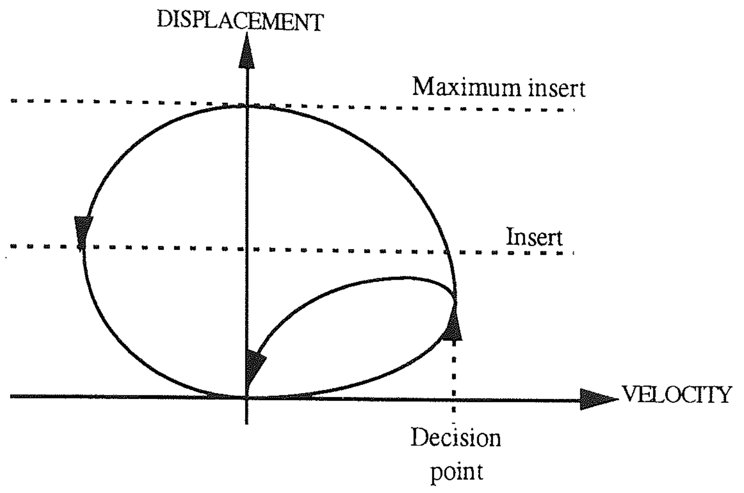


Fig.7.4 Time-critical decision point

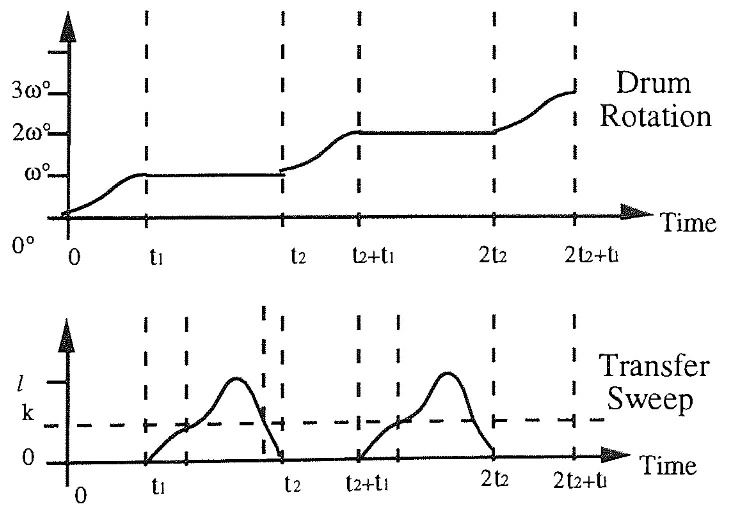


Fig.7.5 The motion profiles of drum and slider system

The diagram shown in Fig.7.5 is the motion profiles of the system described in a sequential form. However, to maximise performance, the case in which asynchronous concurrent motion is allowed will be considered. The motion of the drum and slider including

In [Sagoo and Holding 90], after abortion occurs, the slider stops immediately and is not allowed to move until the drum stops. In this thesis the slider is assumed to move back to its starting point after abortion occurs and is allowed to move towards the drum again when the slider reaches to its starting point.

synchronisation interlocks has been modelled using Petri nets by Sagoo and Holding [90, 91]; however, their Petri net model was constructed using an *ad-hoc* approach. In this section, the functional requirements of the system will be elicited based on the method SBDM and described by the rule-based formalism. It will be seen such an elicitation is natural and easy to understand. For this system, it is assumed that the decision whether to commit to insert or abort is made after the slider moves distance k from its starting point. Symmetrically, when the slider is withdrawn the drum is enabled to rotate after slider is withdrawn to distance k from the original start point. For this system, we also assume that an object such as a manufactured component or object located in an arbor on the drum will be ready for pushing by the slider after each rotation of drum (the loader of the system will be considered in section 7.4 when the flexibility of rule-based formalism is discussed).

7.3.2 Eliciting the Rules Using SBDM

A. Identifying the autonomous components within the system:

1. drum; 2. slider;

B. Identifying the action associated with each component:

1. drum = {rotation}
2. slider = {abortion, insertion, withdraw}

C. Identifying the states associated with each action within each component:

Standardised mnemonics for drums:

D = drum;	ROT = rotate;	S = slider;	INS = insert;
WW = withdraw;	ID = inside drum;	OD = outside drum;	

Drum:

First layer

drum = {D_STOP, D_ROT}

where	D_STOP	drum is stationary
	D_ROT	*rotate* drum

Slider:

First layer

slider = {S_STOP, S_ABORT, S_INS, S_WW}

where	S_STOP	slider is stationary
	S_ABORT	*abort* slider's motion and returns to its initial position
	S_INS	*insert* slider
	S_WW	*withdraw* slider

Second layer

S_INS = {S_OD_INS, S_ID_INS}

where	S_OD_INS	*insert* slider and slider is outside of drum
	S_ID_INS	*insert* slider and slider is inside of drum

S_WW = {S_ID_WW, S_OD_WW}

where	S_ID_WW	*withdraw* slider and slider is inside of drum
	S_OD_WW	*withdraw* slider and slider is outside of drum

Events cause state transformation:

1. Events associated with drum:

D_rotated	drum has rotated.

2. Events associated with slider:

S_aborted	slider has aborted.
S_decision	slider has reached the decision point.
S_inserted	slider has inserted (i.e. reaches maximum insertion point).
S_outed	slider has come out of drum.
S_extracted	slider has extracted.

One of main problem in co-ordination of concurrent processes is the management of communication between them. The general constructs that achieve such co-ordination between processes are referred to as *synchronisation constructs*. Two well known synchronisation constructs are message-passing-based synchronisation and shared resource-based synchronisation [Burns and Wellings 91, Manna and Pnueli 92]. Only message-passing based synchronisation construct will be considered here because there is no shared resource between slider and drum. As the insert and abort operations to be perform by the slider depend on different status of drum, two communication states may be used to convey the drum status to the slider. One informs slider that the drum has rotated (i.e. the object in drum is ready) and slider is allowed to insert into drum and the other informs slider that the drum has not rotated or is being rotating (i.e. the object in drum is not ready) and slider is not allowed to insert into drum. However, only one state is required to describe the slider status to the drum because the drum can rotate whenever the slider is out of drum. The following are the three communication states between slider and drum:

S_OUT	slide is out of drum and drum is allowed to rotate
D_OBJ	object in drum is ready and slider is allowed to insert into drum
D_N_OBJ	object in drum is not ready and slider is not allowed to insert into drum

Since slider has two potential operations (insertion and abortion) to perform when it reaches its decision point, two rules are required to specify each alternative. These two rules will have the same trigger event but have different prerequisite. The following are the functional requirement of drum and slider system described by rule-based formalism:

<u>Synchronisation between slider and drum</u>:

```
R1:     IF          S_OD_INS AND D_OBJ
        WHEN        S_decision
        THEN        S_ID_INS
text    IF "insert slider and slider is outside of drum" AND
           "object in drum is ready and slider is allowed to insert into drum"
        WHEN "slider reaches the decision point"
        THEN "slider is inside of drum and inserting"

R2:     IF          D_STOP AND S_OUT
        THEN        D_ROT
text    IF "drum is stationary" AND "slider is out of drum and drum is allowed to rotate"
        THEN "rotate drum"
```

Drum:

```
R3:     IF          D_ROT AND D_N_OBJ
        WHEN        D_rotated
        THEN        D_STOP AND D_OBJ
text    IF "rotate drum" AND
           "object in drum is not ready and slider is not allowed to insert into drum"
        WHEN "drum has rotated"
        THEN "drum is stationary" AND
                 "object is ready in drum and slider is allowed to insert into drum"
```

Slider:

```
R4:     IF          S_STOP
        THEN        S_OD_INS
text    IF "slider is stationary"
        THEN "insert slider and slider is outside of drum"

R5:     IF          S_OD_INS AND D_N_OBJ
        WHEN        S_decision
        THEN        S_ABORT AND D_N_OBJ
```

text IF "*insert* slider and slider is outside of drum" AND
"object in drum is not ready and slider is not allowed to insert into drum"
WHEN "slider reaches the decision point"
THEN "*abort* slider's motion and returns to its initial position" AND
"object in drum is not ready and slider is not allowed to insert into drum"

R6: IF S_ID_INS
WHEN S_inserted
THEN S_ID_WW AND D_N_OBJ
text IF "*insert* slider and slider is inside of drum"
WHEN "slider has inserted (i.e. reaches maximum insertion point"
THEN "*withdraw* slider and slider is inside of drum" AND
"object in drum is not ready and slider is not allowed to insert into drum"

R7: IF S_ID_WW
WHEN S_outed
THEN S_OD_WW AND S_OUT
text IF "*withdraw* slider and slider is inside of drum"
WHEN "slider has come out of drum"
THEN "*withdraw* slider and slider is outside of drum" AND
"slider is out of drum and drum is allowed to rotate"

R8: IF S_OD_WW
WHEN S_extracted
THEN S_STOP
text IF "*withdraw* slider and slider is outside of drum"
WHEN "slider has extracted"
THEN "slider is stationary"

R9: IF S_ABORT
WHEN S_aborted
THEN S_STOP
text IF "*abort* slider's motion and returns to its initial position"
WHEN "slider has aborted"
THEN "slider is stationary"

Fig.7.6 is the XTPTN model for the rule-based descriptions above. The system starts from states {D_STOP, S_STOP, D_N_OBJ, S_OUT} which corresponds to the initial marking (p_1, p_7, p_{11}, p_{14}). Following are the interpretations of all places of the XTPTN model shown in Fig.7.6:

p_1:	S_STOP;	p_6:	S_ABORT;	p_{13}:	D_OBJ;	p_{23}:	S_outed;
p_2:	S_OD_INS;	p_7:	S_OUT;	p_{14}:	D_N_OBJ;	p_{24}:	S_extracted;
p_3:	S_ID_INS	p_{11}:	D_STOP;	p_{21}:	S_decision;	p_{25}:	S_aborted;
p_4:	S_ID_WW;	p_{12}:	D_ROT;	p_{22}:	S_inserted;	p_{26}:	D_rotated;
p_5:	S_OD_WW;						

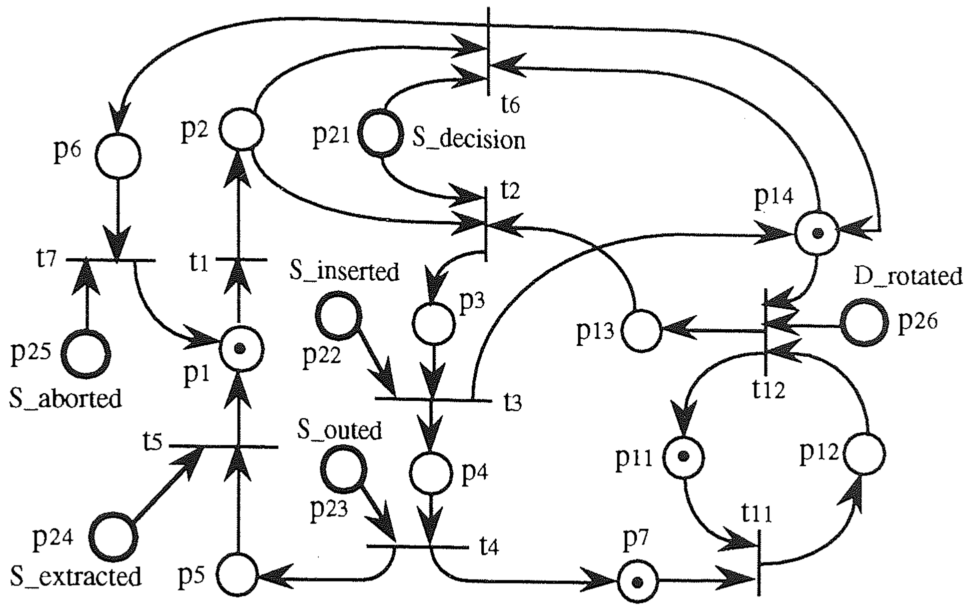


Fig.7.6 XTPTN representation of rules for the drum and slider system

7.3.3 Properties of the System

Safety assertions are invariant properties and liveness assertions are eventuality properties. The safety and liveness requirements for the drum and slider system include:

(a) The state that the slider is in the drum and the state that the drum is rotating must be mutual exclusive. This safety requirement can be expressed in temporal logic as:

$$\Box (p_3 \rightarrow \neg p_{12}) \vee \Box (p_{12} \rightarrow \neg p_3) \tag{7.12}$$

$$\Box (p_4 \rightarrow \neg p_{12}) \vee \Box (p_{12} \rightarrow \neg p_4) \tag{7.13}$$

(b) The state "the object in drum is ready and slider is allowed to insert into drum" and the state "slider is out of drum and drum is allowed to rotate" must be mutual exclusive. This safety requirement can be expressed in formula as:

$$\Box (p_7 \rightarrow \neg p_{13}) \vee \Box (p_{13} \rightarrow \neg p_7) \tag{7.14}$$

(c) Whenever event S_decision occurs, if drum is still rotating, then slider should abort its motion rather than insert into drum. This safety requirement can be expressed as:

$$\Box[(p_2 \wedge p_{12} \wedge p_{21}) \rightarrow \bigcirc p_6] \tag{7.15}$$

(d) If slider is in state S_OD_INS, then either the state (flag) D_OBJ (i.e. the object in drum is ready and slider is allowed to insert into drum) or the state (flag) D_N_OBJ (i.e. the object in drum is not ready and the slider is not allowed to insert into drum) must hold such that slider can decide to insert or abort its operation in order to avoid collision. This safety requirement can be expressed in temporal logic as:

$$\Box[p_2 \rightarrow (p_{13} \vee p_{14})] \tag{7.16}$$

(e) Whenever the object in drum is ready and slider is allowed to insert into drum, then eventually slider will insert into drum. This property can be expressed as:

$$\Box(p_{13} \rightarrow \Diamond p_3) \tag{7.17}$$

(f) Whenever the slider aborts its motion then eventually it will insert into the drum.

$$\Box(p_6 \rightarrow \Diamond p_3) \tag{7.18}$$

7.3.4 Verification of the System Properties

Safety properties must be hold during all the system executions. In the proof of the asserted safety properties, the important token invariance theorem presented in Chapter 2 will be used. The initial marking of the net shown in Fig.7.6 consists of one token each at slider and drum, representing the initial state of the two processes, and two tokens at communication states, representing the initial states for the communication. For the net shown in Fig.7.6, the following system dependent axiom and inference rules can be obtained:

System dependent axiom: $p_1 \wedge p_7 \wedge p_{11} \wedge p_{14}$

System dependent inference rules:

1. $p_1 \wedge \neg p_2 \Rightarrow \bigcirc (p_2 \wedge \neg p_1)$
2. $p_2 \wedge p_{13} \wedge p_{21} \wedge \neg p_3 \Rightarrow \bigcirc (p_3 \wedge \neg p_2 \wedge \neg p_{13} \wedge \neg p_{21})$
3. $p_3 \wedge p_{22} \wedge \neg p_4 \wedge \neg p_{14} \Rightarrow \bigcirc (p_4 \wedge p_{14} \wedge \neg p_3 \wedge \neg p_{22})$
4. $p_4 \wedge p_{23} \wedge \neg p_5 \wedge \neg p_7 \Rightarrow \bigcirc (p_5 \wedge p_7 \wedge \neg p_4 \wedge \neg p_{23})$
5. $p_5 \wedge p_{24} \wedge \neg p_1 \Rightarrow \bigcirc (p_1 \wedge \neg p_5 \wedge \neg p_{24})$
6. $p_2 \wedge p_{14} \wedge p_{21} \wedge \neg p_6 \Rightarrow \bigcirc (p_6 \wedge p_{14} \wedge \neg p_2 \wedge \neg p_{21})$
7. $p_6 \wedge p_{25} \wedge \neg p_1 \Rightarrow \bigcirc (p_1 \wedge \neg p_6 \wedge \neg p_{25})$
8. $p_{11} \wedge p_7 \wedge \neg p_{12} \Rightarrow \bigcirc (p_{12} \wedge \neg p_{11} \wedge \neg p_7)$
9. $p_{12} \wedge p_{14} \wedge p_{26} \wedge \neg p_{11} \wedge \neg p_{13} \Rightarrow (p_{11} \wedge p_{13} \wedge \neg p_{12} \wedge \neg p_{14} \wedge \neg p_{26})$

Proof of property (7.12) $\Box(p_3 \rightarrow \neg p_{12}) \vee \Box(p_{12} \rightarrow \neg p_3)$

(1) $\Box(\neg p_3 \vee \neg p_{12}) \vee \Box(\neg p_{12} \vee \neg p_3)$ from (7.12) by $\varphi \rightarrow \lambda \Leftrightarrow \neg\varphi \vee \lambda$

(2) $\Box \neg(p_3 \wedge p_{12})$ from (1) by logical operators $(\vee, \wedge)$ and tautology

(3) The loop $C = \{p_3, t_3, p_4, t_4, p_7, t_{11}, p_{12}, t_{12}, p_{13}, t_2\}$ shown in Fig.7.6 contains only one initial token in p_7 and satisfies the following requirements:

i) $\forall t \in C, |\bullet t \cap C| = |t\bullet \cap C|$.

ii) $\forall p \in C, (\bullet p \cup p\bullet) \subseteq C$.

According to the token invariance theorem presented in Chapter 2, this means that mutual exclusion between p_3 and p_{12} is guaranteed. This shows that (2) is true.

This proof also shows that properties (7.13) and (7.14) are true because:

i) all places in C are guaranteed to be mutual exclusive by the theorem;

ii) places p_4 and p_{12} required to be mutual exclusive in (7.13) are included in C;

iii) places p_7 and p_{13} required to be mutual exclusive in (7.14) are included in C.

Proof of property (7.15) $\Box[(p_2 \wedge p_{12} \wedge p_{21}) \rightarrow \bigcirc p_6]$

(1) $p_1 \wedge p_7 \wedge p_{11} \wedge p_{14}$ by system dependent axiom

(2) $p_{11} \wedge p_7 \wedge \neg p_{12} \Rightarrow \bigcirc(p_{12} \wedge \neg p_{11} \wedge \neg p_7)$ from (1) by inference rule-8

(3) $p_1 \wedge \neg p_2 \Rightarrow \bigcirc(p_2 \wedge \neg p_1)$ from (1) by inference rule-1

(4) $p_2 \wedge p_{12} \wedge p_{14}$ from (1), (2), and (3)

(5) no inference rule can be applied under (4) unless p_{21} or p_{26} is tokenised

(6) p_{21} by the occurrence of event S_decision

(7) $p_2 \wedge p_{12} \wedge p_{14} \wedge p_{21}$ from (4) and (6)

(8) the ***only*** applicable inference rule under (7) is the inference rule-6 which yields:
$p_2 \wedge p_{14} \wedge p_{21} \wedge \neg p_6 \Rightarrow \bigcirc(p_6 \wedge p_{14} \wedge \neg p_2 \wedge \neg p_{21})$ from (7) by inference rule-6

(9) $\Box[(p_2 \wedge p_{12} \wedge p_{21}) \rightarrow \bigcirc p_6]$ from (7), (8) by temporal reasoning

Proof of property (7.16) $\Box[p_2 \rightarrow (p_{13} \vee p_{14})]$

(1) Consider the following two sets of places and transitions:

$C_1 = \{p_1, t_1, p_2, t_2, p_3, t_3, p_4, t_4, p_5, t_5\}$

$C_2 = \{t_6, p_{14}, t_{12}, p_{13}, t_2, p_3, t_3\}$

C_1 and C_2 satisfy the following requirements:

i) $\forall t \in C_i, |\bullet t \cap C_i| = |t\bullet \cap C_i|$

ii) $\forall p \in C_i, (\bullet p \cup p\bullet) \subseteq C_i$ where i = 1, 2

(2) Since each set only contains one token initially, the mutual exclusion between any two places in C_1 or C_2 is guaranteed according to the token invariance theorem.

(3) $p_2 \rightarrow \neg p_3$ by mutual exclusive property of C_1

(4) $\neg p_3 \rightarrow (p_{13} \vee p_{14})$ by mutual exclusive property of C_2

(5) $p_2 \rightarrow (p_{13} \vee p_{14})$ by (3) and (4)

(6) $\Box[p_2 \rightarrow (p_{13} \vee p_{14})]$ by temporal reasoning

Proof of property (7.17) $\Box(p_{13} \rightarrow \Diamond p_3)$

(1) $p_1 \wedge p_7 \wedge p_{11} \wedge p_{14}$ by system dependent axiom

(2) $p_{11} \wedge p_7 \wedge \neg p_{12} \Rightarrow \bigcirc (p_{12} \wedge \neg p_{11} \wedge \neg p_7)$ from (1) by inference rule-8

(3) p_{26} by the occurrence of event D_rotated

(4) $p_{12} \wedge p_{14} \wedge p_{26} \wedge \neg p_{11} \wedge \neg p_{13} \Rightarrow (p_{11} \wedge p_{13} \wedge \neg p_{12} \wedge \neg p_{14} \wedge \neg p_{26})$

from (1) to (3) by inference rule-9

(5) the ***only*** applicable inference rule under (1)-(4) is the inference rule-1 which yields:

$p_1 \wedge \neg p_2 \Rightarrow \bigcirc (p_2 \wedge \neg p_1)$ from (1) by inference rule-1

(6) no inference rule can be applied under (1)-(5) unless p_{21} is tokenised

p_{21} by the occurrence of event S_decision

(7) the ***only*** applicable inference rule under (1)-(6) is the inference rule-2 which yields:

$p_2 \wedge p_{13} \wedge p_{21} \wedge \neg p_3 \Rightarrow \bigcirc(p_3 \wedge \neg p_2 \wedge \neg p_{13} \wedge \neg p_{21})$ from (4) to (6) by inference rule-2

(8) $\Box (p_{13} \rightarrow \Diamond p_3)$ from (1)-(7) by temporal reasoning

A similar procedure can be used to prove property (7.18).

7.3.5 Time Analysis of the System

Although the safety properties such as no collision between the slider and the drum have been guaranteed on the model, it is still an undesirable system behaviour if the abortion occurs. Is it possible to specify timing constraints for each state in order to avoid abortion? Based on the *assumption* that the drum always starts rotating immediately whenever it is enabled, we have the following timing analysis .

To ensure that the slider always inserts into the drum again rather than aborts its insert operation after it comes out of drum, the time durations associated with the states {S_OD_WW, S_STOP, S_OD_INS, D_ROT} modelled by places $\{p_5, p_1, p_2, p_{12}\}$ need to satisfy:

$$\begin{cases} \text{Time}(p_{12}) < \text{Time}(p_5) + \text{Time}(p_1) + \text{Time}(p_2) \\ \text{where Time}(p_i) \text{ stands for the time duration which} \\ \text{is consumed by state modelled by } p_i. \end{cases} \qquad (7.19)$$

By inspecting the net model shown in Fig.7.6, it is seen that abortion will occur from the initial marking (state) M_0 if the following relation (7.20) holds:

$$\text{Time}(p_1 + p_2) < \text{Time}(p_{12}) < \text{Time}(p_5 + p_1 + p_2) \qquad (7.20)$$

Note that (7.20) satisfies (7.19). To overcome this problem, the following two methods can be considered:

1) By introducing an extra place p and an extra transition t. Place p is put into the initial marking (state) and p_1 is taken out from the initial marking (state). For transition t, p is its input place and p_1 is its output place. Transition t is associated with timing constraint <Time(p), Time(p)>, e.g., the state modelled by place p is forced to wait for Time(p) time units before t occurs. The relation between Time(p) and other places are:

$$\text{Time}(p_{12}) < \text{Time}(p) + \text{Time}(p_1) + \text{Time}(p_2) \qquad (7.21)$$

2) By putting p_5 rather than p_1 into the initial marking such that the initial marking M_0 is defined as $M_0 = \{p_5, p_7, p_{11}, p_{14}\}$, and set the TURT$^{\tau_0}$ (p_5) = Time(p_5) (see Chapter 4). In this case, if timing constraint (7.19) is imposed, then abortion can be avoided because place p_{13} (i.e. state — "object in drum is ready and slider is allowed to insert into drum") has been tokenised (true) whenever transition t_2 is enabled after event S-decision occurs.

It is known that the cycle performance of slider is determined by timings associated with following places $\{p_1, p_2, p_3, p_4, p_5\}$. Timing constraint (7.19) shows that the timings associated with $\{p_1, p_2, p_5\}$ depend on the timing associated with p_{12}, which means the cycle performance of slider cannot be independently improved by only changing the slider behaviour associated with the states modelled by $\{p_1, p_2, p_5\}$. However, it can be shown that the cycle performance of slider can be improved independently by changing the (slider behaviour) timings associated with (states modelled by) p_3 and p_4 if we can prove following two properties:

(g) Whenever places p_3 and p_{22} are tokenised, no transitions except the transition t_3 (i.e. inference rule-3) can be fired (applied) whatever other places are tokenised. This property can be expressed in temporal logic as:

$$\Box[(p_3 \wedge p_{22}) \rightarrow \bigcirc\, p_4] \qquad (7.22)$$

(h) Whenever places p_4 and p_{23} are tokenised, no transitions except the transition t_4 (i.e. inference rule-4) can be fired (applied) whatever other places are tokenised. This property can be expressed in temporal logic as:

$$\Box[(p_4 \wedge p_{23}) \rightarrow \bigcirc (p_5 \wedge p_7)] \qquad (7.23)$$

Proof of property (7.22) $\Box[(p_3 \wedge p_{22}) \rightarrow \bigcirc p_4]$

(1) $p_1 \wedge p_7 \wedge p_{11} \wedge p_{14}$ by system dependent axiom
(2) $p_7 \wedge p_{11} \Rightarrow \bigcirc p_{12}$ from (1) by inference rule-8
(3) $p_1 \Rightarrow \bigcirc p_2$ from (1) by inference rule-1
(4) no inference rule can be applied under (1) to (3) unless p_{21} or p_{26} is tokenised by the occurrence of event D_rotated or S_decision. The occurrence of event D_rotated (i.e. p_{26} is tokenised) yields:
$p_{12} \wedge p_{14} \wedge p_{26} \Rightarrow \bigcirc p_{11} \wedge p_{13}$ from (1) to (3) by inference rule-9
(5) no inference rule can be applied under (1) to (4) unless p_{21} is tokenised by the occurrence of event S_decision. The occurrence of event S_decision (i.e. p_{21} is tokenised) yields:
$p_2 \wedge p_{13} \wedge p_{21} \Rightarrow \bigcirc p_3$ from (1) to (5) by inference rule-2
(6) no inference rule can be applied under (1) to (5) unless p_{22} is tokenised by the occurrence of event S_inserted. The occurrence of event S_inserted yields:
(7) $p_3 \wedge p_{11} \wedge p_{22}$ from (1) to (6)
(8) the ***only*** applicable inference rule under (7) is the inference rule-3 which yields:
$p_3 \wedge p_{22} \Rightarrow \bigcirc p_4 \wedge p_{14}$ from (7) by inference rule-3
(9) $\Box[(p_3 \wedge p_{22}) \rightarrow \bigcirc p_4]$ from (8) by temporal reasoning

A similar procedure can be used to prove property (7.23). By combining properties (g) and (h), we can have following property (i):

(i) $$\Box[p_3 \rightarrow (\neg p_5 \, \mathcal{U} \, p_7)] \qquad (7.24)$$

Properties (g) to (i) above imply that when slider is inside of drum (i.e. p_3 or p_4 is tokenised), no places (i.e. p_1, p_2, p_5, and p_{12}) specified by timing constraint (7.19) can be tokenised until slider comes out of drum. This means that the timings associated with p_3 and p_4 are irrelevant to the timing associated with p_{12}. Conversely, it indicates that the operations associated with states modelled by p_3 and p_4 (i.e. the *insertion* and *withdraw* within the drum) can be improved independently from the point of view of slider performance.

7.4 Flexibility of Rule-Based Formalism

7.4.1 Introduction

The generation of a functional requirements for a real-time control system is a continuing problem. Even if a system is designed to be totally reliable and well structured and completely meets the user requirements, it still likely requires change during its lifetime. Changes may be caused by a better understanding of the processes controlled by the system, the need for improved performance, changes in the behaviour of the external environment, and technical advances that result in changes in the physical system configuration [White and Lavi 85]. When changes are made in the system operating rules, the problem of maintaining complete, consistent, and correct system functional requirements will grow. In this section, it will show how the rule-based approach make the enhancement of the amended user requirements easier.

7.4.2 Extension of the Slider and Drum System

For the drum and slider system, suppose we want to consider a third mechanism, a "loader" component, whose function is to put or load an object into the drum, which can only be done when the drum is stationary. It is assuming that the drum has at least two slots on arbors for holding objects and at the beginning no objects are in any of the arbors. The rule-based approach provides an easy way for including such an extension because it has the important advantage of incrementabity in building the functional requirements. To put the "loader" component into the drum and slider system, it is not necessary to consider the whole system. For example, since the "loader" component is independent of the slider, it is not necessary to consider the slider when the functional requirements of "loader" is elicited. Consideration of the loader and the drum shows that it is necessary to introduce several rules to describe the functional requirement for the "loader" component and modify the relevant rules of the drum. Since there is no shared resource between the drum and the loader, the synchronisation between them can be described using the typical message-passing mechanism. That is, two communication states for the co-operation between the drum and the loader are needed comparable to those for co-operation between the drum and the slider. The following are the steps of elicitation for the extension.

A. Component:		Loader
B. Action associated with loader:		{loading}
C. States associated with loader:		{L_STOP, L_LOAD}
where	L_STOP	loader is stationary
	L_LOAD	*load* object into drum
D. Event	L_loaded	object has been loaded into drum

Communication states between drum and loader:

L_AVAI	drum is stationary and empty arbor is available for loading;
L_LOADED	drum is stationary and empty arbor has been loaded by loader;

The rule-based functional requirements for the loader and the changed part of drum are defined as:

Synchronisation between drum and loader:

```
R1′:    IF          L_STOP AND L_AVAI
        THEN        L_LOAD
text    IF "loader is stationary" AND
           "drum is stationary and empty arbor is available for loading"
        THEN "load object into drum"

R2′:    IF          D_STOP AND S_OUT AND L_LOADED
        THEN        D_ROT
text    IF "drum is stationary" AND
           "slider is out of drum and drum is allowed to rotate" AND
           "drum is stationary and empty arbor has been loaded by loader"
        THEN "rotate drum"
```

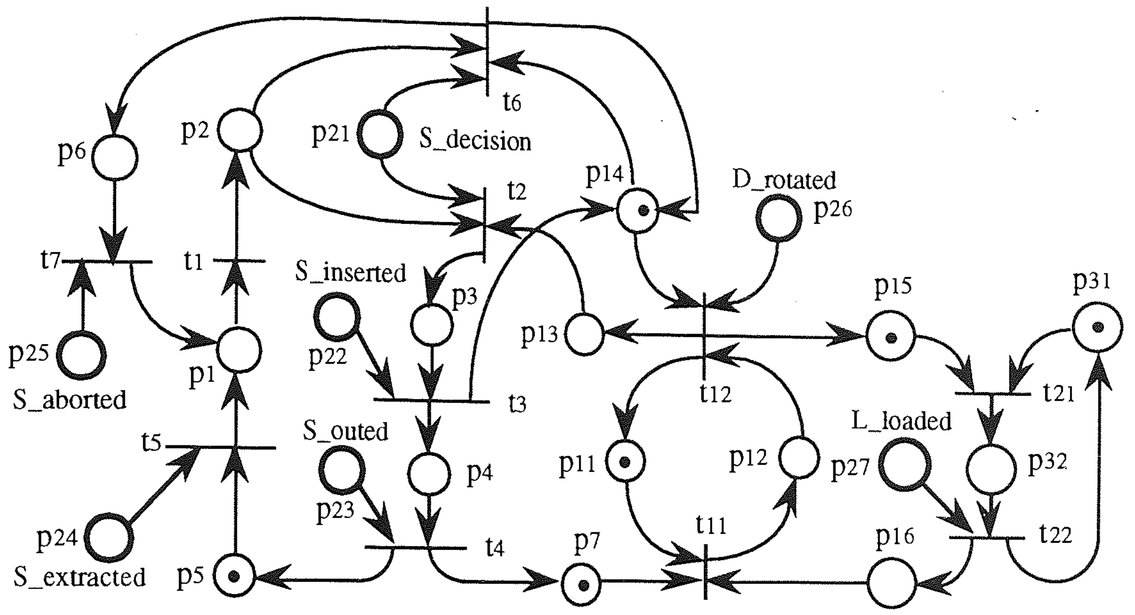
Loader:

```
R3′:    IF          L_LOAD
        WHEN        L_loaded
        THEN        L_LOADED AND L_STOP
text    IF "load object into drum"
        WHEN "object has been loaded into drum"
        THEN "drum is stationary and empty arbor has been loaded by loader" AND
                "loader is stationary"
```

Drum:

```
R4′:    IF          D_ROT AND D_N_OBJ
        WHEN        D_rotated;
        THEN        D_STOP AND D_OBJ AND L_AVAI
text    IF "rotate drum" AND
          "object in drum is not ready and slider is not allowed to insert into drum"
        WHEN "drum has rotated"
        THEN "drum is stationary" AND
              "object in drum is ready and slider is allowed to insert into drum" AND
              "drum is stationary and empty arbor is available for loading"
```

The XTPTN model for the extended system is shown in Fig.7.7 in which the definitions of duration for places and enabling intervals for transitions are omitted. It can be seen that the slider part in Fig.7.7 is still the same as that in Fig.7.6. The extended system starts from initial marking $\{p_5, p_7, p_{11}, p_{14}, p_{15}, p_{31}\}$.



p_{15}: L_AVAI; p_{16}: L_LOADED; p_{31}: L_STOP; p_{32}: L_LOAD

Fig.7.7 XTPTN model for extended drum and slider system

7.4.3 Properties of the Extended System

The properties for the extended system include:

(j) $\Box(p_{12} \rightarrow \neg p_{32}) \vee \Box(p_{32} \rightarrow \neg p_{12})$ (7.25)

(k) $\Box[p_{15} \rightarrow \Diamond p_{16}]$ (7.26)

(l) $\Box[p_{11} \rightarrow \Diamond p_{12}]$ (7.27)

Property (7.25) means that whenever the drum is in rotating states, the loader must not be in loading state and vice versa; (7.26) states that if an empty arbor is available for loading it will eventually be loaded, and (7.27) states that if the drum is stationary then it will eventually rotate. These new properties can be proved using the techniques discussed in this chapter; details of the proofs are included in the Appendix D. Again, significant use is made of the token invariance technique, particularly to prove the mutual exclusion; this technique is considered to be one of the powerful proof techniques associated with nets [Rozenberg and Thiagarajan 87, Murata 89, Manna and Pnueli 92].

Summary

This chapter demonstrated the application of a rule-based formalism and SBDM for eliciting the functional requirements of real-time process-control problems and their analysis. It has been shown that:

- the SBDM method can be applied to identify elements of a rule-based schema,
- the rule-based schema can be used as system behaviour description language,
- the relationship between rule-based schema and Petri net models of rules leads naturally to the synthesis of a Petri net solution,
- the formal method (Petri net theory and temporal logic) can be used to verify the properties of systems.

Chapter 8 Conclusion and Future Research

8.1 Conclusions and Contributions

The aim of this thesis is to contribute to the wider use of formal techniques in the modelling and analysis of real-time process-control systems, particularly of PLC systems, in their developments. SFC [IEC848 88], a variant of Grafcet which was developed in an industrial environment, has become one of the most widely used discrete event models in control industry. SFC is used in this thesis because it has achieved a level of maturity and acceptance and has been used in a wide variety of control applications as a software front end. Furthermore, the international standard IEC1131 [IEC 93] in which SFC has been chosen as the discrete mathematical model (with few changes) for real-time process-control system development has been one of the stimuli for this research.

In Chapter 2, a detailed discussion of today's practical methods within the category that focuses on both states and events was presented. From Chapter 3 to Chapter 5, this thesis has been primarily concerned with the investigation of Grafcet and its variant SFC. The Grafcet model was investigated first in Chapter 3. The scope of the investigation was from the definitions of basic elements such as events and conditions, to detailed consideration of its firing rules. Although Grafcet was shown to be a powerful model due to its peculiar firing rules, such as the simultaneous firing rule and the permitted implicit dependency, a problem arises due to implicit dependency. Methods to solve the implicit dependency was proposed. To make implicit dependency explicit in Grafcet, an inhibitor arc Grafcet was defined. With the help of inhibitor arcs, an algorithm to eliminate the implicit dependency was given and its correctness was proven. Within an inhibitor arc Grafcet, the simultaneous firing rule for a class of Grafcets has been shown to be a simplification rather than a fundamental restriction.

SFC models were investigated in Chapter 4. Firstly, SFC was discussed in various aspects such as timing, qualifiers, and evolution rules. Then, based on the discussion, a formal definition of a unified set of evolution rules for SFC was presented. This formal definition of evolution rules is necessary for both industrial implementation and SFC analysis. In order to formally analyse the design described in SFC using developed analysis techniques, an extended Petri net model was defined which forms the basis for the contents presented in Chapter 5. Finally, the transformation method from SFC to the extended Petri net model was defined. In Chapter 5, the problem of how to analyse the properties of an SFC design modelled using extended Petri nets was discussed. It was shown that reachability graph, trace theory, net proof, and timing analysis techniques could be used to evaluate system properties.

In view of the increasing importance in the industrial world of system development based on Grafcet and SFC, particularly in controller developments, the lack of a standardised software development formalism is a significant handicap. In Chapter 6, a rule-based approach to capture the functional requirements from the real-world systems has been proposed. The rule-based approach is natural, easy to use and can be integrated with the development of real-time process-control systems. The rule-based approach allows the system designer to easily elicit the functional requirements and to prove the correctness of the captured functional requirements with respect to the safety requirements. The rule-based approach also enjoys the important advantage of incrementality. By this we mean that if, after developing a functional description, a designer suddenly realises that the functional requirement is incomplete, he or she can always rectify the functional description by adding one or more missing functional requirements as additional rules in an incremental fashion. To assist the rule-based development approach, a method called SBDM (System Behaviour Driven Method) was proposed in Chapter 6. Via XTPTN and temporal logic, the formal verification techniques to prove the correctness of the rule-based functional requirements, instead of allowing only graphical editing and simulation, was also described in Chapter 6. The adopted approach draws on evidence of investigations to Grafcet and SFC models, the literature and a case study of a real-time control system in industry. The applications of this rule-based approach together with the method SBDM and the formal verification techniques were presented in Chapter 7 through two real-time process-control system examples.

8.1.1 Advantages of a Rule-Based Approach

A number of important characteristics were identified for real-time process-control systems in Chapter 1. These characteristics are now examined in a comprehensive fashion to summarise the strengths for rule-based approach and the SBDM method proposed in this thesis.

(1) Environment

Real-time process-control systems must keep up with their environments [Lin and Burke 92]. It is known that the static rule formalism is suitable to describe the state-based causal behaviour of a computer system but not suitable to describe the stimuli (such as events) coming from the behaviour of a physical system under control. This arises because of the difference between an event and a state and a corresponding deficiency of describing the effect of an event on the state transition in a static rule formalism. The event-based rule proposed in this thesis provides a precise means to describe the system behaviours which are both state-based and event-driven. Abstractly speaking, the static rule formalism can only describe "what" to do but not "when" to do. However, the event-based rule formalism enriches the static rule formalism such that "when" to do can be precisely described via the enhanced rule structure. "When" is very important in real-time process-control systems because it reflects the effects that the physical system under control (i.e., the environment) has on the computer system.

(2) Timing

By inspecting the definition of rule constructs, it can be seen that the timing is not quantitatively specified in the rule-based formalism. However, if each action is decomposed into a "software part" associated with the computer system and a "mechanical part" associated with the physical system under control, then timing can easily be embedded into the rule-based formalism by associating a duration with each computer system action and imposing a time interval on each event generated by the physical system. After such an enhancement, the rule-based description can still be modelled by the XTPTN because the time duration associated with each action in a rule can be modelled by the delay defined for places in XTPTN and the timing interval imposed on each event can be modelled by function ψ defined for transitions in XTPTN.

(3) Concurrency

The rule-based formalism provides a means to describe concurrent behaviour because each rule is defined based on local states rather than global states. Each local state describes the behaviour of one of the system components. The state transition specified by a rule is normally triggered by an event associated with one system component. It is the very nature of a real-time process control system that many components behave in a parallel manner.

(4) Safety

Safety properties are a set of invariant assertions which are expressed as temporal formulas in the propositional linear time logic (LTL) in this thesis. The rule-based functional requirement descriptions can be represented as a XTPTN model with capacity $K(p) = 1$ for each place p. The XTPTN can serve as a temporal logic model of a system. Thus, temporal Petri net verification techniques developed by Sagoo and Holding [90, 91] can be used to determine if the XTPTN is a model of the logic formulas (and by implication, that the safety properties hold in the functional requirements described by the rules). If the verification determines the formulas true, then the safety properties hold in the XTPTN model and also in the rule-based functional requirements.

(5) Size and Complexity

The rule-based formalism provides a hierarchical approach to elicit the functional requirements from the real world system by its sequential and parallel decompositions. Like Statecharts [Harel 87], the sequential and parallel decompositions defined for rules support top-down refinement and bottom-up clustering.

(6) Maintainability

As mentioned at the beginning of this chapter, rule-based approach has the important advantage of incrementality. That is, the rule-based approach supports incremental construction of functional requirements. Incrementality is a very important feature for system enhancement which has nicely been shown in Chapter 7 through the enhancement of the drum and slider system. The rule-based approach explicitly describes dependency and independency, this advantage supports maintenance as well because ripple effect analysis can be performed much easier based on explicit representation than based on implicit representation. Ripple effect analysis is one of important activities in maintenance after a change is made to a system.

The rule-based approach also has a number of other desirable properties: it has a uniform and complete mechanism for describing the behaviour of physical processes from the point of view of the computer system; it is executable and computationally universal; it is not only state-based but also event-driven with a natural Petri net representation; it is amenable to formal analysis; and it permits straightforward descriptions of given and required behaviours.

8.1.2 Disadvantages of a Rule-Based Approach

Rule-based formalism is not a complete modelling language, it only provides an approach. Many details about the action and event cannot be specified just by the proposed rule formalism. Although the details of most components do not need to be specified at functional requirement level, some components do need to be fully specified during this stage. Rule-based formalism lacks an appropriate means such as data variables and functions to describe them.

8.1.3 Major Contributions

The major contributions of this research described in this thesis are:

(1) An evaluation of Grafcet model and a method to make implicit communication in Grafcet explicit.

(2) A comparative study between SFC and Petri nets and a formal definition of the firing rules for SFC.

(3) An extended Petri net model and a mapping from SFC to the extended model.

(4) A rule-based approach and a system behaviour driven method (SBDM) for real-time process-control systems in functional requirement elicitation.

(5) Utilisation of these formal techniques (Petri net theory and temporal logic) to support the analysis for both SFC and rule-based descriptions.

(6) Illustration of the techniques by application to two examples from industrial demonstration machines.

8.2 Future Work

A number of areas for future research are identified as a result of the work developed in this thesis.

(1) An immediate extension to this work is that a high level description language is needed for describing the behaviour of system. Many operations associated with actions and variables involving conditions and events cannot be specified precisely by the rule-based formalism. Investigation of a formal production rule language is recommended.

(2) Further experimentation is required to determine whether the rule-based formalism and SBDM method proposed in this thesis could provide useful, cost effective support for functional requirements elicitation during early controller developments.

(3) A set of software tools supporting SFC analysis and rule-based formalism are needed. Based on such a software supporting environment, either SFC analysis or rule-based software development can be proceeded automatically or semi-automatically.

(4) Another extension of this work is that the further application of formal methods in the verifications of real-time process-control system designs when other languages such as FBD and ST are considered. In this aspect, the investigation of predicate logic rather than propositional logic may need to be considered because practical applications of temporal logic and of Petri nets require first order concepts in general [Reisig 88b].

(5) Finally, further research may need to explore the combination between partial order semantics, especially the Interleaving Set Temporal Logic (ISTL) [Katz and Peled 90], and Petri Nets. For this further research, the verification of properties of Petri nets may be considered to perform based on trace theory [Mazurkiewicz 87] rather than sequential firing sequences as in [Suzuki and Lu 89, He and Lee 90, Sagoo and Holding 90].

References:

[Alford 77] M.W. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements," *IEEE Trans. Soft. Eng.*, Vol.SE-3, No.1, Jan. 1977, pp.60-69.

[Alford 85] M.W. Alford, "SREM at the Age of Eight: The Distributed Computing Design System," *Computer*, April 1985, pp.36-46.

[Andre and Peraldi 93] C. Andre and M.A. Peraldi, "Grafcet and Synchronous Language," *Automatic Control Production Systems*, Vol.27, No.1, 1993, pp.95-105.

[Assche *et al* 88] F. Van Assche, P. Layzell, P. Loucopoulos and G. Speltincx, "Information systems development: a rule-based approach", *Knowledge-Based Systems*, Vol. 1, No. 4, Sept. 1988, pp.227-234.

[Atabakhche *et al* 86] H. Atabakhche, D. S. Barbalho, R. Valette, M. Courvoisier, "From Petri Net Based PLCs to Knowledge Based Control", *International Conference on Industrial Electronics, Control and Instrumentation, IECON-86*, Milwaukee, Wisconsin, 1986, pp.817-822.

[Aygaline and Denat 93] P. Aygaline and J-D Denat, "Validation of functional Grafcet models and performance evaluation of the associated system using Petri nets," *Automatic Control Production Systems*, Vol.27, No.1, 1993, pp.81-92.

[Aygaline and Denat 94] P. Aygaline and J-D Denat, Private communication, 1994.

[Azema *et al* 76] P. Azema, R. Valette, and M. Diaz, "Petri Nets as a Common Tool for Design Verification and Hardware Simulation," *Proc. Design Automation Conference*, San Francisco, California, June 1976, pp.109-116.

[Le Bail *et al* 91] Jean Le Bail, Hassane Alla, Rene David, "Hybrid Petri Nets," *ECC91, European Control Conference*, Grenoble, France, July 2-5 1991, pp.1472-1477.

[Barker and Song 92] H.A. Barker and J. Song, "A graphical simulation tool for programmable logic controllers," *IEE Colloquim on Discrete Event Dynamic Systems -- A New Generation of Modelling Simulation and Control Application*, 9, June 1992, London, pp.4/1-4/4.

[Barker *et al* 89] H.A. Barker, J. Song, and P. Townsend, "A Rule-Based Procedure for Generating Programmable Logic Controller Code from Graphical Input in the Form of Ladder Diagrams," *Engineering Applications of Artificial Intelligence*, Vol. 2, No. 4, 1989, pp.300-306.

[Benveniste and Berry 91] A. Benveniste and Gerard Berry, "The Synchronous Approach to Reactive and Real-Time Systems," *Proceedings of the IEEE*, Vol. 79, No.9, Sept. 1991, pp.1270-1282.

[Berthomieu and Diaz 91] B. Berthomieu and M. Diaz, "Modeling and Verification of Time Dependent Systems Using Time Petri Nets", *IEEE Trans. Soft. Eng.*, Vol. 17, No. 3, March 1991 pp.259-273.

[Best 87] E. Best, "Structural theory of Petri nets: The free choice hiatus," *LNCS*: , Vol. 254, Springer-Verlag, 1987, pp.168-206.

[Boehm 81] B.W.Boehm, *Software Engineering Economics*, Prentice-Hall, 1981.

[Bologna and Leveson 86] S. Bologna and N.G. Leveson (ed.) Special Issue on Reliability and Safety in Real-Time Process Control, *IEEE Trans. Soft. Eng.*, Vol.SE-12, No.9, Sept. 1986.

[Brams *et al* 93] H. Brams, M. Lobelle, G. Detroz, A. April, "G++: a Graphical Language to Specify Real-Time Parallel Applications", *Proceedings of Euromicro Workshop on Parallel and Distributed Processing*, Gran Canaria, Jan. 1993, pp.185-193.

[Burns and Wellings 91] A. Burns and A. Wellings, *Real-Time Systems and Their Programming Languages*, Addison-Wesley, 1991.

[Cao and Ho 90] Xi-Ren Cao and Yu-Chi Ho, "Models of Discrete Event Dynamic Systems," *IEEE Control Systems Magazine*, Vol.10, No.4, June 1990, pp.69-76.

[Cassandras and Ramadge 90] C.G. Cassandras and P.J. Ramadge, "Toward a Control Theory for Discrete Event Systems," *IEEE Control Systems Magazine*, Vol.10, No.4, June 1990, pp.66-68.

[Castellano *et al* 87] L. Castellano, G. De Michelis, and L. Pomello, "Concurrency vs Interleaving: an Instructive Example," *Bulletin, European Association for Theorectical Computer Science*, (31), 1987, pp.12-15.

[Chandrasekharan *et al* 85] M. Chandrasekharan, B. Dasarathy, and Z. Kishimoto, "Requirements-Based Testing of Real-Time Systems: Modeling for Testability", *IEEE Computer*, Vol. 18, April 1985, pp.71-80.

[Clarke *et al* 86] E.M.Clarke, E.A.Emerson and A.P.Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications", *ACM Trans. on Programming Languages and Systems*, Vol. 8, No.2, April 1986, pp.244-263.

[Cofrancesco *et al* 91a] P. Cofrancesco, A. Cristoforetti, and R. Scattolini, "Petri nets based approach to software development for real-time control," *IEE Proceedings-D*, Vol.138, No.5, Sept. 1991, pp.474-478.

[Cofrancesco *et al* 91b] P. Cofrancesco, A. Cristoforetti, M. Villa, R. Scattolini and D.W. Clarke, "A Workbench for Digital Control Systems," *IEEE Control Systems*, Vol.11, No.1, Jan. 1991, pp.102-106.

[Cohen *et al* 86] B. Cohen, W.T. Harwood and M.I. Jackson, *The Specification of Complex Systems*, Addison-Wesley, 1986.

[Coolahan and Roussooulos 83] J.E. Coolahan and N. Roussooulos, "Timing Requirements for Time-Driven Systems Using Augmented Petri Nets," *IEEE Trans. Soft. Eng.*, Vol. SE-9, Sept. 1983, pp.603-616.

[Courvoiser *et al* 83] M. Courvoiser, R. Valette, J. M. Bigou, and P. Esteban, "A programmable logic controller based on a high level specification tool," *Proc. of IECON*, 1983, pp.174-179.

[Crispin 90] A.J. Crispin, *Programmable Logic Controllers and their Engineering Applications*, McGraw-Hill, 1990

[Dasarathy 85] B. Dasarathy, "Timing Constraints of Real-Time Systems: Constructs for Expressing Them, Methods of Validating Them", *IEEE Trans.Soft. Eng.*, Vol. SE-11, No. 1, Jan. 1985, pp.80-86.

[David 91] Rene David, "Modeling of Dynamic Systems by Petri Nets", *ECC 91 European Control Conference*, Grenoble, France, July 2-5, 1991, pp.136-147

[David 93] Rene David, Private communication, 1993.

[David and Alla 92] Rene David and Hassane Alla, *Petri Nets & Grafcet, Tool for Modelling Discrete Event Systems*, Prentice Hall, 1992.

[der Beek 94] M. von der Beek, "A Comparison of Statecharts Variants," *LNCS,* Vol.863, H. Langmaack, E. -P. de Roever and J. Vytopil (Eds), Springer-Verlag, 1994.

[Emerson and Srinivasan 88] E. Allen Emerson and Jai Srinivasan, "Branching Time Temporal Logic," *LNCS 354, Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency 1988*, Springer-Verlag, pp.123-172.

[Etessami and Hura 91] Farhad S. Etessami and Gurdeep S. Hura, "Rule-Based Design Methodolody for Solving Control Problems", *IEEE Trans. Soft. Eng.*, Vol. 17, No. 3, March 1991, pp.274-282.

[Falcione and Krogh 93] A. Falcione and B. Krogh, "Design Recovery for Relay Ladder Logic," *IEEE Control Systems*, Vol. 13, No.2, April 1993, pp.90-98.

[Faulk and Parnas 88] S.R. Faulk and D.L. Parnas, "On Synchronization in Hard-Real-Time Systems," *Communications of the ACM*, Vol. 31, No.3, March 1988, pp.274-287.

[Ferrarini and Maffezzoni 91] L. Ferrarini and C. Maffezzoni, "Designing Logic Controllers with Petri Nets", *Computer Aided Design in Control System, IFAC Symposium*, Swansea, U.K., 15-17, July 1991, pp.403-408.

[Fleming 88] W.H. Fleming, Chair, "Future Directions in Control Theory -- A Mathematical Perspective," *Report of the Panel on Future Directions in Control Theory*, SERC, UK 1988.

[Frachet and Colombari 93] J-P Frachet and G. Colombari, "Elements for a semantics of the time in GRAFCET and dynamic systems using non-standard analysis," *Automatic Control Production Systems*, Vol.27, No.1, 1993, pp.107-124.

[Genrich and Lautenbach 81] H.J. Genrich and K. Lautenbach, "System Modelling with High-Level Petri Nets," *Theoretical Computer Science* 13, 1981, pp.109-136.

[Ghezzi *et al* 91] C. Ghezzi, D. Mandrioli, s. Morasca, and M. Pezze, "A Unified Hige-Level Petri Net Formalism for Time-Critical Systems," *IEEE Trans. Soft. Eng.*, Vol.17, No.2, Feb. 1991, pp.160-171.

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

[Grossman *et al* 93] *Hybrid Systems, LNCS, Vol. 736,* R.L. Grossman, A. Nerode, A.P. Ravn and H. Rischel (Eds), Springer-Verlag, 1993.

[Halang and Kramer 92] W. Halang and B. Kramer, "Achieving High Integrity of Process Control Software by Graphical Design and Formal Verification," *Software Enginering Journal*, Vol.7, No.1, Jan. 1992, pp.53-64.

[Halang and Kramer 94] W. Halang and B. Kramer, "Safety Assurance in Process Control," *IEEE Software*, Vol.11, No.1, Jan. 1994, pp.61-67.

[Harel 87] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, 8, 1987, pp.231-274.

[Hayes-Roth 85] Frederick Hayes-Roth, "Rule-Based Systems", *Communications of the ACM*, Vol. 28, No.9, Sept. 1985, pp.921-932.

[He and Lee 90] X. He and John A.N. Lee, "Integrating Predicate Transition Nets with First Order Temporal Logic in the Specification and Verification of Concurrent Systems," *Formal Aspects of Computing*, (2), 1990, BCS, pp.226-246.

[Heninger 80] K.L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," *IEEE Trans. Soft. Eng.*, Vol. SE-6, No.1, Jan. 1980, pp.2-13.

[Heymann 90] M. Heymann, "Concurrency and Discrete Event Control," *IEEE Control Systems Magazine*, Vol.10, No.4, June 1990, pp.103-112.

[Hill and Holding 90] M. R. Hill and d. J. Holding, "The modelling, simulatin, and analysis of comit protocols in distributed computing systems," in *Proceedings of the 1990 UKSC Conference on Computer Simulation*, Burgess Hill, UK, Sept. 5-7, 1990, pp.207-212.

[Ho 89] Y.C. Ho, (Ed.) Special Issue on Dynamics of Discrete Event Systems, *Proc. of the IEEE*, Vol. 77, Jan. 1989.

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

[Holiday and Venon 87] M.A. Holiday and M.K. Venon, "A Generalized Timed Petri Net Model for Performance Analysis," *IEEE Trans. Soft. Eng.*, Vol. SE-13, No. 12, Dec. 1987, pp.1297-1310.

[Holloway and Krogh 90] L.E. Holloway and B.H. Krogh, "Synthesis of Feedback Control Logic for a Class of Controlled Petri Nets," *IEEE Trans. automatic Control*, Vol.35, No.5, May 1990, pp.514-523.

[Hopcroft and Ullman 79] J.E. Hopcroft and J.D. Ullman, *Formal Languages And Their Relation To Automata*, Addison-Wesley, 1979.

[IEC848 88] International Electrotechnical Commission Technical Committee No.3, 848 Preparation of Function Charts for Control Systems, 1988.

[IEC 93] International Standard IEC1131 Part 3, *Programmable Controllers, Programming Languages*, International Electrotechnical Commission, Geneva, March 1993.

[IEE 93] *Proceedings of IEE Colloquium on "Advances in software engineering for PLC (programmable logic controller) systems,"* London, 14 October 1993.

[Inan and Varaiya 88] Kemal Inan and Pravin Varaiya, "Finitely Recursive Processes," *Lecture Notes in Control and Information Sciences, Discrete Event Sstems: Models and Applications, IIASA Conference*, Hungary, August 3-7, 1987, Springer-Verlag, 1988.

[Jaffe *et al* 91] M.S.Jaffe, N.G.Leveson, M.P.E.Heimdahl and .E.Melhart, "Software Requirements Analysis for Real-Time Process-Control Systems,", *IEEE Trans. Soft. Eng.*, Vol. 17, No.3, March 1991, pp.241-258.

[Jahanian and Mok 86] F. Jahanian and A.K. Mok, "Safety Analysis of Timing Properties in Real-Time Systems,"*IEEE Trans. Soft. Eng.*, SE-12(9), Sept. 1986, pp.890-904.

[Jensen 81a] K. Jensen, "Coloured Petri Nets and the Invariant Method," *Theoretical Computer Science*, 1981, Vol. 14, pp.317-336.

[Jensen 81b] K. Jensen, "How to Find Invariants for Coloured Petri Nets," *LNCS*, Vol. 118, 1981, pp.327-338.

[Jiang *et al* 91] J. Jiang, X. Zhou, and D. Robson, "Program slicing for C — the problems in implementation," *Proceedings of Conference on Software Maintenance 1991*, 15-17 Oct. 1991 Sorrento Italy, pp.182-190.

[Jones 86] C.B. Jones, *Systematic Software Development Using VDM.* Prentice-Hall, 1986.

[Joseph and Goswami 89] M. Joseph and A. Goswami, "Formal description of realtime systems: a review," *Information and Software Technology*, Vol.31, No.2, March 1989, pp.67-76.

[Joshi and Supinski 92] S.B.Joshi and M.R.Supinski, "The development of a generic PC-based programmable logic controller simulator," *International Journal of Production Research*, Vo.30, No.1, 1992, pp.151-168.

[Karam and Buhr 91] G.M. Karam and R.J.A. Buhr, "Temporal Logic-Based Deadlock Analysis For Ada," *IEEE Trans. Soft. Eng.*, Vol.17, No.10, Oct. 1991, pp.1109-1125.

[Katz and Peled 90] S. Katz and D. Peled, "Interleaving Set Temporal Logic," *Theoretical Computer Science*, Vol. 75, 1990, pp.263-287.

[Kemmerer and Ghezzi 92] R.A. Kemmerer and C. Ghezzi (ed.), Special Issue: Specification and Analysis of Real-Time Systems, *IEEE Trans. Soft. Eng.*, Vol.18, No.9, Sept. 1992.

[Komoda *et al* 84] N. Komoda, K. Kera, T. Kubo, "An Autonomus, Decentralized Control System for Factory Automation", *IEEE Computer*, Dec. 1984, pp.73-83.

[Lamport 83] L. Lamport, "Specifying concurrent program modules," *ACM Trans. Programming Languages and Systems*, vol. 5, No. 2, April 1983, pp.190-222.

[Lamport 89] L. Lamport, "A Simple Approach to Specifying Concurrent Systems," *Communications of the ACM*, Vol. 32, No. 1, Jan. 1989, pp.32-45.

[Langmaack *et al* 94] *Formal Techniques in Real-Time and Fault-Tolerant Systems*, *LNCS* Vol. 863, H. Langmaack, E. -P. de Roever and J. Vytopil (Eds), Springer-Verlag, 1994.

[Lautenbach 87] Kurt Lautenbach, "Linear Algrbraic Technieues for Place/Transition Nets," *LNCS 255, Petri Nets: Applications and Relationships to other Models of Concurrency*, Springer-Verlag, 1987, pp.143-167.

[Le Guernic *et al* 91] P. Le Guernic, T. Gautier, M. Le Borgne and C. Le Maire, "Programming Real-Time Applications with SIGNAL," *Proceedings of IEEE*, Vol.79, No.9, Sept. 1991, pp.1321-1336.

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

[Leveson and Stolzy 87] N.G. Leveson and J.L. Stolzy, "Safety Analysis Using Petri Nets," *IEEE Trans. Soft. Eng.*, Vol. SE-13, No.3, March 1987, pp.386-397.

[Leveson and Nenmann 93] N.G. Leveson and P.G. Nenmann (ed.) Special Issue on Software for Critical Systems, *IEEE Trans. Soft. Eng.*, Vol.19, No.1, Jan. 1993.

[Levi and Agrawala 90] S. T. Levi and A. K. Agrawala, *Real Time System Design*, Mcgraw-Hill, 1990.

[Lewis 92] R.W. Lewis, private communications, Sept. 1992.

[Lin and Burke 92] Kwei-Jay Lin and E.J. Burke (ed.) Special Issue on Real-Time Realities, *IEEE Software*, Vol.9, No.5, Sept. 1992.

[Loucopoulos and Champion 88] P. Loucopoulos and R. Champion, "Knowledge-based approach to requirements engineering using method and domain knowledge", *Knowledge-Based Systems*, Vol.1, No.3, June 1988, pp.179-187.

[Loucopoulos and Layzell 89] P. Loucopoulos and P.J. Layzell, "Improving information system development and evolution using a rule-based paradigm" *Software Engineering Journal*, Sept. 1989, pp.259-267.

[Mallaband 91] S. Mallaband, "Specification of Real Time Control Systems y Means of Sequential Function Charts", *International Conference on Software Engineering for Real-Time Systems*, 16-18, Sept. 1991, U.K., pp.57-61.

[Manna and Pnueli 88] Z. Manna and A. Pnueli, "The Anchored Version of the Temporal Framework," *LNCS 354, Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, 1988*, Springer-Verlag, pp.201-283.

[Manna and Pnueli 92] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems,* Springer-Verlag, 1992.

[Marce and Le Parc 92] L. Marce and P. Le Parc, "Defining the Semantics of Languages for Programmable Controllers with Synchronous Processes", *International Workshop on Real-Time Programming WRPT'92*, Bruges, Belgium, 23-26, June 1992, pp.113-118.

[Marconi 86] The Functional Specification Definition of Haifield Tunnel Software Control System, Marconi Command and Control Systems, 1986.

[Mazurkiewicz 87] Antoni Mazurkiewicz, "Trace Theory", *LNCS 255, Petri Nets: Applications and Relationships to other Models of Concurrency*, Springer-Verlag, 1987, pp.279-324.

[Mazurkiewicz 88] Antoni Mazurkiewicz, "Basic Notions of Trace Theory," *LNCS 354, Linear Time, Branching Time and Partial Order in Logics and Models of Concurrency* 1988, Spinger-Verlag, pp.285-363.

[McDermid 90] J.A. McDermid, "Issues in Developing Software for Safety Critical Systems", Dept. of Computer Science, University of York, YCS 138, July 1990.

[McMermid and Thewlis 91] J.A. McDermid and D.J. Thewlis (ed.) Special Issue on Safety-Critical Systems, *Software Engineering Journal*, Vol.6, No.2, March 1991.

[Merlin and Farber 76] P. M. Merlin and D. J. Farber, "Recoverability of Communication Protocols -- Implications of a Theoretical Study", *IEEE Trans. on Communication* Vol. COM-24, Sept. 1976, pp.1036-1043.

[Michel 90] G. Michel, *Programmable Logic Controllers—Architecture and Applications*, New York: Wiley, 1990.

[Milner 89] R. Milner, *Communication And Concurrency*, Prentice Hall, 1989.

[Mok 83] A.Ka-Lau Mok, "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment," *PhD Thesis*, Department of Electrical Engineering and Computer Science, MIT, May 1983.

[Moller and Tofts 90] F. Moller and C. Tofts, "A temporal calculus of communicating systems," In *CONCUR 90, LNCS 458*, Springer-Verlag, 1990, pp.401-415.

[Moon 94] Il Moon, "Modelling Programmable Logic Controllers for Logic Verification," *IEEE Control Systems*, Vol.14, No.2, April 1994, pp.53-59.

[Murata *et al* 86] T. Murata, N. Komoda, K. Matsumoto, and K. Haruna, "a Petri Net-Based Controller for Flexible and Maintainable Sequence Control and its Applications in Factory Automation," *IEE Trans. Industrial Electronic*, Vol.IE-33, No.1, Feb. 1986, pp.1-8.

[Murata 89] Tadao Murata, "Petri Nets: Properties, Analysis and Applications", *Proceedings of the IEEE*, Vol. 77, No. 4, April 1989, pp.541-580.

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

[Ostroff 90] J.S. Ostroff, "A Logic for Real-Time Discrete Event Processes," *IEEE Control Systems Magazine*, Vol.10, No.4, June 1990, pp.95-102.

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

[Ostroff and Wonham 90] J.S. Ostroff and W.M. Wonham, "A Framework for Real-Time Discrete Event Control," *IEEE Trans. on automatic Control*, Vol.35, No.4, April, 1990, pp.386-397.

[Pathak and Krogh 89] Dhiraj K. Pathak and Bruce H. Krogh "concurrent Operation Specification Language, COSL, for Low-Level Manufacturing Control", *Computer Industry*, Vol. 12, No. 2, May 1989, pp.107-122.

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

[Pnueli 86] A. Pnueli, "Applications of Temporal Logic to the Specification and Verification of Reactive Sstems: a Servey of Current Trends," *Current Trends in Concurrency, LNCS* 244, Springer-Verlag, 1986, pp.510-584.

[Poo and Layzell 90] C-C D Poo and P J Layzell, "Enchancing software maintenance through explicit system representation", *Information and Software Technology*, Vol.32, No.3, April 1990, pp.176-186.

[Queille and Sifakis 82] J.P. Queille and J. Sifakis, "specification and Verification of Concurrent systems in CESAR," *LNCS 224*, 1982, Springer-Verlag, pp.510-584.

[Ramamoorthy and Ho 80] C. V. Ramamoorthy and G. S. Ho, "Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets", *IEEE Trans. Soft. Eng.*, Vol. SE-6, No. 5, Sept. 1980, pp.440-449.

[Ramchandani 74] C. Ramchandani, "Analysis of Asynchronous Concurrent Systems by Timed Petri Nets", *Technical Report* MAC TR120, MIT Feb. 1974.

[Razouk 84] R. R. Razouk, "The Derivation of Performance Expressions for Communication Protocols from TImed Petri Net Models", *Computer Comm. Review* (USA), Vol. 14, No. 2, 1984, pp.210-217.

[Reed and Roscoe 86] G. Reed and a. Roscoe, "A timed model for communicating sequential processes," in *Proc. ICALP'86, LNCS* 226, Springer-Verlag, pp.314-323.

[Reisig 85] W. Reisig, *Petri Nets: An Introduction, Volume 4 of EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, 1985.

[Reisig 88a] W. Reisig, "Temporal Logic and Causality in Concurrent Systems," *International Conference on Formal Methods for Disctributed Systems*, Oct. 1988, Hamburg Germany, Concurrency' 88, Springer-Verlag, Vol. 335, 1988, pp.121-139.

[Reisig 88b] W. Reisig, "Towards a Temporal Logic for Causality and Choice in Distributed Systems," *LNCS 354, Linear Time, BranchingTime and Partial Order in Logics and Models for Concurrency 1988*, Springer-Verlag, pp.603-627.

[Reisig 92] W. Reisig, "Combining Petri Nets and Other Formal Methods," *LNCS 616, Application and Theory of Petri Nets 1992,* Springer-Verlag, pp.24-44.

[Reutenauer 88] Christophe Reutenauer, *The Mathematics of Petri Nets*, Prentice Hall International, 1988.

[Rozenberg 87] G. Rozenberg, "Behaviour of Elementary Net Systems", *LNCS 254, Advances in Petri Nets 1986, Part I, Proceedings of an Advanced Course* , Springer-Verlag, 1987, pp.60-94.

[Rozenberg 88] G. Rozenberg, *LNCS 340, Advances in Petri Nets 1988*, Rozenberg (Ed.) Springer-Verlag, 1988.

[Rozeenberg 89] G. Rozenberg, *LNCS 424, Advances in Petri Nets 1989*, Rozenberg (Ed.) Pringer-Verlag, 1989.

[Rozenberg 90] G. Rozenberg, *LNCS 483, Advances in Petri Nets 1990*, Rozenberg (Ed.) Springer-Verlag, 1991.

[Rozenberg 91] G. Rozenberg, *LNCS 524, Advances in Petri Nets 1991*, Rozenberg (Ed.) Springer-Verlag, 1991.

[Rozenberg 92] G. Rozenberg, *LNCS 609, Advances in Petri Nets 1992*, Rozenberg (Ed.) Springer-Verlag, 1992.

[Rozenberg and Thiagarajan 87] G. Rozenberg and P.S. Thiagarajan, "Petri Nets: Basic Notions, Structure, Behaviour," *LNCS, Current Trends in Concurrency*, Springer-Verlag, pp.585-668.

[Sahraoui *et al* 87] A. Sahraoui, H. Atabakhche, M. Courvoisier, R. Valette, "Joining Petri

Nets and Knowledge Based Systems", *Proceedings 1987 IEEE Int. Conf. Robotics and Automation, IEEE Council Robotic and Automation*, March 1987, pp.1160-1165.

[Sagoo 92] J.S. Sagoo, The Development of Hard Real-Time Systems Using A Formal Approach, *Ph.D Thesis*, Department of Electronic Engineering and Applied Physics, Aston University in Birmingham, UK, Sept. 1992.

[Sagoo and Holding 90] J.S.Sagoo and D.J.Holding, "The Specification and Design of Hard Real-Time System using Timed and Temporal petri Nets", *Microprocessing and Microprogramming*, (30), 1990, pp.389-396.

[Sagoo and Holding 91] J.S.Sagoo and D.J.Holding, "A Comparison of Temporal Petri Net Techniques in the Specification and Design of Hard Real-Time Systems", *Microprocessing and Microprogramming*, (32), 1991, pp.111-118.

[Sayat and Ladet 93] B. Sayat and P. Ladet, "Control specification of a production system using GRAFCET and Petri nets," *Automatic Control Production Systems*, Vol.27, No.1, 1993, pp.53-64.

[Scholefield 90] D.J. Scholefield, "The Formal Development of Real-Time Systems: A Review," Department of Computer Science, University of York, YCS 145, November, 1990.

[Shaw 92] A.C. Shaw, "Communicating Real-Time State Machines," *IEEE Trans. Soft. Eng.*, Vol.18, No.9, Sept. 1992, pp.805-816.

[Shin 87] K.G. Shin, "Introduction to the Special Issue of Real-Time Systems," *IEEE Trans. on Computer*, Vol.C-36, No.8, August 1987, pp.901-902.

[Sifakis 80] J. Sifakis, "Performance Evaluation of Systems Using Nets", *LNCS, 84, Net Theory and Applications*, pp.307-319.

[Silva 89] M. Silva, "Logical Controllers," *IFAC Symp. Low Cost Automation 1989*, Milan, Italy, Nov.8-10, 1989, pp.249-259.

[Silva and Valette 89] M. Silva and R. Valette, "Petri Nets and Flexible Manfacturing," *LNCS 424, Advances in Petri Nets 1989*, G. Rozenberg (Ed.) Springer-Verlag, pp.374-417.

[Silva and Vililla 82] M. Silva and S. Vililla, "Programmable Logic Controllers and Petri Nets: A Comparative Study," *IFAC Software for Computer Control*, Madrid, Spain 1982, pp.83-88.

[Skeen and Stonebraker 83] D. Skeen and M. Stonebraker, "A Formal Modelof Crash Recovery in a Distributed System," *IEEE Trans. Soft. Eng.*, Vol. SE-9, No.3, 1983, pp.219-228.

[Sommerville 92] I. Sommerville, *Software Engineering*, Addison-Wesley, 1992.

[Spivey 92] J.M. Spivey, *The Z notation: a reference manual*, Prentice-Hall, 1992.

[Stankovic 88] J.A. Stankovic, "A Serious Problem for Next-Generation Systems," *Computer*, Vol. 21, No.10, October 1988, pp.10-19.

[Strutt 89] N. Strutt, "A Survey of Formal Methods," Report From The Ministry Of Defence, ARETM (AXC) 89001, Jan. 1989.

[Suzuki and Lu 89] I. Suzuki and H. Lu, "Temporal Petri Nets and Their Applications to Modelling and Analysis of a Handshake Daisy chain Arbiter," *IEEE Trans. Computers*, Vol. 38, No. 5, May 1989, pp.696-704.

[Suzuki and Murata 83] I. Suzuki and T. Murata, "A Method for Stemwise Refinements and Abstractions of Petri Nets," *J. of Comp. and Syst. Sci.*, Vol. 27, 1983, pp.51-76.

[Swainston 91] F. Swainston, *A Systems Approach to Programmable Controllers*, Thomas Nelson Australia, 1991.

[Tashiro *et al* 85] T. Tashiro, N. Komoda, I. Tsushima and K. Matsumoto, "Advanced Software for Constraint Combinational Control of Discrete Event Systems -- Rule-Based Control Software for Factory Automation --", *Proceedings COMPINT 85 Computer Aided Technologies, IEEE*, Montreal, Canada Sept. 1985, pp.132-137.

[Tyrrell and Holding 86] A.M. Tyrrell and D.J. Holding, "Design of Reliable Software in Distributed Systems Using the Conversation Scheme," *IEEE Trans. Soft. Eng.*, Vol. SE-12, No.9, Sept. 1986, pp.921-928.

[Uchihira and Honiden 90] N. Uchihira and S. Honiden, "verification and Synthesis of Concurrent Programs Using Petri Nets and Temporal Logic," *The Transactions of The IEICE*, vol. E-73, No.12, Dec. 1990, pp2001-2010.

[Valette 79] R. Valette, "Analysis of Petri Nets by Stepwise Refinements," *J. of Comp. and Syst. Sci.*, Vol. 18, 1979, pp.35-46.

[Valette *et al* 85] R. Valette, M. Courvoisier, H. Demmou, JM. Bigou, and C. Desclaus, "Putting Petri Nets to Work for Controlling Flexible Manufacturing Systems," *Proc. of ISCAS 85*, pp.929-932.

[Velilla and Silva 88] S. Velilla and M. Silva, "The SPY: A Mechanism for Safe Implementation of Highly Concurrent Systems," *IFAC Real Time Programming*, Valencia, Spain, 1988, pp.75-81.

[Vogler 91] W. Vogler, *Modular Construction and Partial Order Semantics of Petri Nets*, *LNCS 625*, Springer-Verlag, 1991.

[Ward and Mellor 85] P.T. Ward and S.J. Mellor, *Structured Development for Real-Time Systems*, New York: Yourdon, 1985.

[Warnock 88] I.G. Warnock, *Programmable Controllers: Operation and Application*, Prentice Hall, 1988.

[Waterman 86] Donald A. Waterman, *A Guide to Expert Systems*, Addison-Wesley Pusblishin Company, 1986.

[Waters 82] R.C. Waters, "The Programmer's Apprentice: Knowledge Based Program Editing," *IEEE Trans. Soft. Eng.*, Vol. SE-8, No.1, Jan. 1982, pp.1-12.

[Weiser 84] M. Weiser, "Program Slicing", *IEEE Trans. Soft. Eng.*, Vol. SE-10, No. 4, July 1984, pp.352-357.

[Wellings 91] A. Wellings (ed.) Special Issue on Real-Time Software, *Software Engineering Journal*, Vol.6, No.3, May 1991.

[White and Lavi 85] S.M. White and Jonah Z. Lavi, "Embedded Computer System

Requirements Workshop", *Computer*, Vol. 18, April 1985, pp.67-70.

[Wilson and Krogh 90] G. Wilson, B.H. Krogh, "Petri Net Tools for the Specification and Analysis of Discrete Controllers", *IEEE Trans. Soft. Eng.*, Vol. 16, No. 1, Jan. 1990, pp.39-50.

[Wing 90] J.M. Wing, "A Specifier's Introduction to Formal Methods," *Computer*, Vol.23 , No.9 , Sept. 1990, pp.8-24.

[Zave 82] P. Zave, "An Operational Approach to Requirements Specification for Embedded Systems," *IEEE Trans. Soft., Eng.*, Vol. SE-8, No. 3, May 1982, pp.250-269.

[Zhou and DiCesare 89] M. Zhou and F. DiCesare, "Adaptive Design of Petri Net Controllers for Error Recovery in Automated Manufacturing Systems," *IEEE Trans. Systems, Man, and Cybernetics*, vol.19, no.5, Spet./Oct. 1989, pp.963-973.

[Zhou and DiCesare 93] M. Zhou and F. DiCesare, *Petri Net Synthesis For Discrete Event Control of Manufacturing Systems*, Klumer Academic 1993.

[Zuberek 85] W.M. Zuberek, "Performance Evaluation Using Extended Timed Petri Nets", *International Workshop on TImed Petri Nets*, Torino, Italy, July 1-3, 1985, pp.272-278.

[Zurawski and Dillon 91] R. Zurawski and T.S. Dillon, "A Method for Systematic Construction of Functional Abstractions of a Class of Petri Nets Models used to Represent Primary Components of Flexible Manufacturing Systems," *Proc. IECON'91*, 28 Oct. - 1 Nov. 1991, Kobe, Japan, pp.872-877.

Appendix A

Concurrent Set Algorithm for a Class of Petri Nets

After generating the reachability graph, the concurrent set for each place can be obtained by following formula:

$$\forall p \in \mathrm{P}: \mathrm{CS}(p) = [\bigcup_{\mathrm{M_i} \in \mathrm{R}(\mathrm{M_0}) \wedge \mathrm{M_i}(p) \neq 0} \mathrm{M_i}] - \{p\} \qquad \text{(A.1)}$$

However, for a special class of Petri nets the concurrent set for each place can be calculated directly. This class of Petri nets satisfy following two requirements:

i) they are safe (1-bounded) nets, i.e. the number of tokens in each place does not exceed 1 for any marking reachable from $\mathrm{M_0}$;
ii) they are marked graphs, i.e., each place has exactly one input transition and exactly one output transition.

That is, the Petri nets belonging to this class are safe (1-bounded) marked graphs.

Definition *Marked graph* (Murata 89]
A marked graph (MG) is an ordinary Petri net such that each place p has exactly one input transition and exactly one output transition, i.e.,

$$(\forall p \in \mathrm{P})\ [|\bullet p| = |p\bullet| = 1], \text{ where P is the set of places} \qquad \text{(A,2)}$$

An developed algorithm to calculate the concurrent set for each place of this kind of Petri nets will be presented in this appendix. Following is an important theorem related to concurrent sets of places for safe marked graphs.

TheoremA.1: Inheritance Property
Let PN = (P, T, F, M_0) be a safe marked graph, $t \in$ T be a transition and CS(p) be the concurrent set of place $p \in \bullet t$. Then,

$$\forall\, p' \in [\bigcap_{p \in \bullet t} \mathrm{CS}(p)] \Rightarrow p' \in \mathrm{CS}(p''), \text{ where } p'' \in t\bullet \qquad \text{(A.3)}$$

To prove this property, it is necessary to prove that:

$$\forall p' \in [\bigcap_{p \in \bullet t} CS(p)] \Rightarrow \exists M \in R(M_0): [\prod_{p \in \bullet t} M(p)] \neq 0 \wedge M(p') \neq 0 \quad (1)$$

Proof:

Let us assume that:

$$\neg \exists M \in R(M_0): [\prod_{p \in \bullet t} M(p)] \neq 0 \wedge M(p') \neq 0 \quad (2)$$

then, without losing generality, we have:

$$\exists p_1, p_2 \in \bullet t: \neg \exists M \in R(M_0): M(p_1) \neq 0 \wedge M(p_2) \neq 0 \wedge M(p') \neq 0 \quad (3)$$

That is,

$$\forall M', M'' \in R(M_0):$$

$$M'(p_1) \neq 0 \wedge M'(p') \neq 0 \Rightarrow M'(p_2) = 0 \quad (4)$$

and $$M''(p_2) \neq 0 \wedge M''(p') \neq 0 \Rightarrow M''(p_1) = 0 \quad (5)$$

The relationship between M′ and M″ can be one of following cases:

i) $M'' \in R(M') \vee M' \in R(M'')$ (6)

ii) $M'' \notin R(M') \vee M' \notin R(M'')$ (7)

Let us consider disjunct $M'' \in R(M')$ of (6) first. When M′ is reached from the initial marking M_0, clearly the transition t, where $p_1, p_2 \in \bullet t$:, cannot be fired under marking M′ because there is no token in p_2. Since each place $p \in P$ has exactly one input transition and one output transition, p_1 is still tokenised when M″ is reached from M′. This conclusion is contradicted with (5) derived based on the assumption, which shows (1) true. A similar procedure can be used to show that disjunct $M' \in R(M'')$ of (6) is contradicted with (4) as well.

Now, let us consider disjunct $M'' \notin R(M')$ of (7). From (4) and (5) we have:

$$\exists\ \delta_1, \delta_2 \in T^*: M_0[\delta_1 > M', M_0[\delta_2 > M'' \quad (8)$$

where, $\delta_1 = t_{11}t_{12}...t_{1p}$, $\delta_2 = t_{21}t_{22}...t_{2q}$

Let Ord(t) represent the ordinal number of transition t in δ_1 or δ_2. For example, Ord(t_{27})=7 and Ord(t_{15})=5. Note that different firings of the same transition are labelled differently in firing sequences.

To prove the assumption (2) false, it is necessary to prove that:

$\exists M''' \in R(M') \wedge \exists M'''' \in R(M'')$:

$M'''(p_1) \neq 0 \wedge M'''(p_2) \neq 0 \wedge M'''(p') \neq 0$ (9)

and $M''''(p_1) \neq 0 \wedge M''''(p_2) \neq 0 \wedge M''''(p') \neq 0$ (10)

Only (9) will be proved here (similar procedure can be used to prove (10)). After M′ is reached from M_0, if $p' \notin M_0$, then based (A.2), (4), (5), and (8), we can find a $t_{2x} \in \delta_2$ and a t_{2y} which satisfy:

i) $p_2 \in t_{2x}\bullet \wedge \neg\exists\, k > x\ [t_{2x} \in \delta_2 \wedge p_2 \in t_{2k}\bullet]$ (11)

ii) $\exists p \in \bullet t_{2x}\ [M'(p) = 0]$ (12)

iii) $p' \in t_{2y}\bullet \wedge \neg\exists\, m > y\ [t_{2m} \in \delta_2 \wedge t_{2m} = t_{1i}]$ (13)

Note that $\mathrm{Ord}(t_{2y}) \neq \mathrm{Ord}(t_{2x})$, otherwise, (7) will not be true because, based on (A.2), the marking M″ has been reached from M′ when p' is tokenised. Also, a place p satisfying ii) will exist, otherwise (7) will not be true as well because, by firing t_{2x} under M′, a marking M″ in which p_2 is tokenised can be obtained.

Case-1: $\mathrm{Ord}(t_{2y}) > \mathrm{Ord}(t_{2x})$

Let us construct a new sequence δ_3, denoted as $\delta_3 = t_2^{(1)}t_2^{(2)}\ldots t_2^{(k)}$, based on the sub-sequence $\delta = t_{21}t_{22}\ldots t_{2x}\ldots t_{2y}$ of δ_2 in such a way such that each $t_2^{(i)} \in \delta_3$ satisfies the following requirements:

i) there exists a transition $t_{2j} \in \delta$ which is the *(i)*th transition in δ (from left to right) whose occurrence is not covered by δ_1;

ii) For each transition $t_{2x} \in \delta$, if the occurrence of t_{2x} is not covered by δ_1 and $\mathrm{Ord}(t_{2x}) < \mathrm{Ord}(t_{2j})$, then there exists a $k < j$ such that $t_2^{(k)} \in \delta_3 \wedge t_2^{(k)} = t_{2x}$.

Based on following conditions:

i) $\forall t \in \delta_3 \Rightarrow t \notin \delta_1$;

ii) $|\bullet p| = |p\bullet| = 1$;

iii) $M_0[\delta_2 > M''$.

we can get the conclusion that δ_3 is a feasible firing sequence under marking M′. Otherwise, we can derive following contradictions:

either i) $\exists t_2^{(i)} \in \delta_3, \exists p \in \bullet t_2^{(i)}\ [|p\bullet| > 1]$;

or ii) δ_2 is an infeasible firing sequence from the initial marking.

Let $M'[\delta_3>M$, where $\delta_3 = t_2^{(1)}t_2^{(2)}...t_2^{(k)}$. According to the properties of t_{2x} and t_{2y} specified by (12) and (13), we have:

$$M(p_1) \neq 0 \wedge M(p_2) \neq 0 \wedge M(p') \neq 0 \qquad (14)$$

The similar procedure can be applied for the case-2, where $\mathrm{Ord}(t_{2y}) < \mathrm{Ord}(t_{2x})$. Thus, a contradiction is derived from (14). Similar result can be obtained for the case of $p' \in M_0$, by constructing δ_3 in following way:

```
if    $\forall t_{2i} \in \delta_3 : p' \notin t_{2i}\bullet$
then  $\delta = t_{21}t_{22}...t_{2x}$
else  $\delta = t_{21}t_{22}...t_{2x}...t_{2y}$
```

All the contradictions show (1) is true.

Algorithm of Concurrent Set for 1-Bounded Marked Graphs

```
$\forall t \in T, \forall p \in P \quad R^0_t := \emptyset, \quad CS^0(p) := \emptyset$
$\forall p \in M_0 \quad CS^0(p) := M_0 - \{p\}$
i := 0;
Repeat
    i := i + 1;
    $\forall t \in T, \forall p \in P$
        $R^i_t := R^{i-1}_t$;     $CS^i(p) := CS^{i-1}(p)$;
    For each $t \in T$
        IF $CS^i(p) \neq \emptyset$, for each $p \in \bullet t$     /* to consider enabled transition only */
        then $R^i_t := R^i_t \cup [\bigcap_{p \in \bullet t} CS^i(p)]$     /* to consider the inheritance property */
        For each $p \in (t\bullet)$
            $CS^i(p) := CS^i(p) \cup R^i_t \cup t\bullet - \{p\}$
            /* if $|t\bullet| \leq 1$, then $t\bullet - \{p\} = \emptyset$, otherwise $t\bullet - \{p\} \neq \emptyset$. */
            /* Note '$t\bullet - \{p\}$' is used to deal with the splitting situation */
            $\forall p' \in CS^i(p)$
                If $p \notin CS^i(p')$     /* to consider the symmetric property */
                then $CS^i(p') := CS^i(p') \cup \{p\}$
Until $R^i_t = R^{i-1}_t$ and $CS^i(p) = CS^{i-1}(p)$, for each $t \in T$ and each $p \in P$.
```

Appendix B

Proof of the Deadlock-Free Property of the Traffic Control System

The following is the proof of Property-3 for the traffic light control system in Chapter 5. To show Property-3 is true, it is necessary to prove that the net shown in Fig.5.7 is live (i.e., for any transition *t* of the net, it is possible to ultimately fire *t*, no matter what marking has been reached from the initial marking M_0).

The net shown in Fig.5.7 (see Fig.B.1) is a marked graph according to the definition given in Appendix A. A marked graph can be viewed as a marked directed graph, where arcs correspond to places, nodes to transitions, and tokens are placed on arcs [Rozenberg and Thiagarajan 87, Murata 89]. For example, the net shown in Fig.B.1 (Fig.5.7) can be drawn as the marked directed graph shown in Fig.B.2:

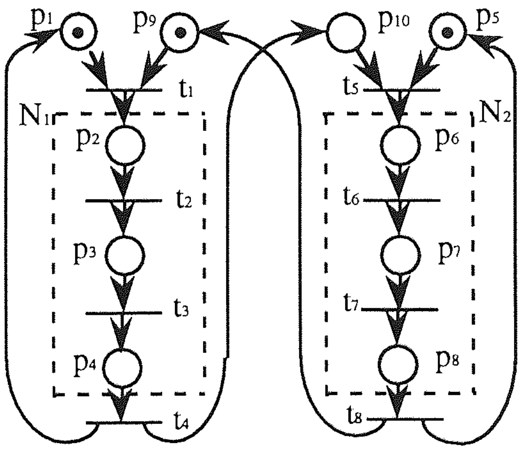


Fig.B.1 Copy of Fig.5.7

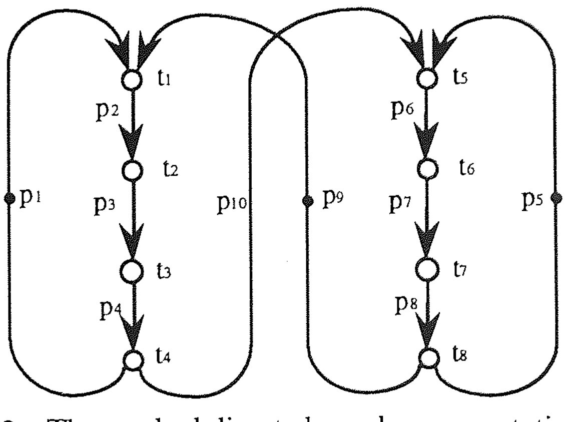


Fig.B.2 The marked directed graph representation of Fig.5.7

The firing of a node (transition) in a marked graph consists of removing one token from each incoming arc (input place) and adding one token to each outgoing arc (output place). If a node is on a directed circuit (or loop), then exactly one of its incoming arcs and one of its outgoing arcs belong to the directed circuit. This means that if there are no tokens on a directed circuit at the initial marking, then this directed circuit remains *token-free*. Conversely, if a node is never enabled by any firing sequence, then by back-tracking token-free arcs, one can find a token-free directed circuit. Based on these analysis, the liveness of a marked graph can be characterised as follow theorem [Rozenberg and Thiagarajan 87, Murata89].

Theorem A marked graph is live iff the initial marking M_0 places at least one token on every directed circuit. □

Now, let us look at the marked graph shown in Fig.B.1 in which three directed circuits exist:

$C_1 = \{p_1, p_2, p_3, p_4\}$
$C_2 = \{p_2, p_3, p_4, p_{10}, p_6, p_7, p_8, p_9\}$
$C_3 = \{p_5, p_6, p_7, p_8\}$

Each directed circuit C_i satisfies $\#(C_i, M_0) \geq 1$ ($i = 1, 2, 3$), where $\#(C_i, M)$ denotes the total number of tokens on C_i. Thus, the net shown in Fig.5.12 is guaranteed to be live (i.e., there is no deadlock in it).

Appendix C

The Theorem of Mazurkiewicz about Traces and its Proof

Following is the proof of the theorem about traces used in Chapter 5. Before presenting the proof, two propositions are needed to present first.

Let $\Sigma = (A, D)$ be a concurrent alphabet.

Proposition-1 If $\omega \equiv_\Sigma \mu$, where ω and μ denote strings over Σ, then ω is a permutation of μ.

Proof: In the definition of $\equiv_\Sigma$ relation, ab is a permutation of ba; if μ_1, ω_1 are permutation of μ_2, ω_2 respectively, then $\mu_1\omega_1$ is a permutation of $\mu_2\omega_2$; finally, composition of permutation is a permutation. □

Proposition-2 Let ω_1, ω_2 be strings, e_1, e_2 be symbols, $e_1 \neq e_2$. If $\omega_1 e_1 \equiv_\Sigma \omega_2 e_2$, then e_1, e_2 are independent and there is a string ω such that $\omega_1 \equiv_\Sigma \omega e_2$, $\omega_2 \equiv_\Sigma \omega e_1$.

Proof: Assume $\omega_1 e_1 \equiv_\Sigma \omega_2 e_2$, $e_1 \neq e_2$. Applying twice the cancellation rule we get $\omega_1 \backslash e_2 \equiv_\Sigma \omega_2 \backslash e_1$; denote $\omega_1 \backslash e_2$ by ω. By the same rule we get $\omega_1 \equiv_\Sigma (\omega_2 \backslash e_1) e_2 \equiv_\Sigma \omega e_2$, and $\omega_2 \equiv_\Sigma (\omega_1 \backslash e_2) e_1 \equiv_\Sigma \omega e_1$. Thus, since $\omega_1 e_1 \equiv_\Sigma \omega_2 e_2$, $\omega e_2 e_1 \equiv_\Sigma \omega e_1 e_2$. By the cancellation rule $e_2 e_1 \equiv_\Sigma e_1 e_2$, which proves independency of e_1, e_2. □

Theorem $\forall\, \omega_1, \omega_2 \in A^*$: $\langle\omega_1\rangle_\Sigma \cong \langle\omega_2\rangle_\Sigma$ iff $[\omega_1]_\Sigma = [\omega_2]_\Sigma$

Proof by induction w.r. to the length of ω_1. If $\omega_1 = \varepsilon$ the assertion of the theorem is obviously true, since the only trace equal to the empty trace is empty and the only *d*-graph isomorphic to the empty *d*-graph is also empty. Let now $\omega_1 = \mu_1 e_1$ for a string μ_1 and symbol e_1. Assume $\omega_1 \equiv_\Sigma \omega_2$. Thus, $\mu_1 e_1 \equiv_\Sigma \omega_2$; ω_2 cannot be empty, hence $\omega_2 = \mu_2 e_2$ for a string μ_2 and symbol e_2. We have then $\mu_1 e_1 \equiv_\Sigma \mu_2 e_2$ and by Proposition-2 either $e_1 = e_2$ and $\mu_1 \equiv_\Sigma \mu_2$ or there is a string μ such that $\mu_1 = \mu e_2$, $\mu_2 \equiv_\Sigma \mu e_1$ and e_1, e_2 are independent. In the first case $\langle\mu_1\rangle_\Sigma \cong \langle\mu_2\rangle_\Sigma$ by induction hypothesis and $\langle\mu_1 e_1\rangle_\Sigma \cong \langle\mu_2 e_2\rangle_\Sigma$ because $e_1 = e_2$. In the second case e_1, e_2 are independent , hence $\langle\mu e_1 e_2\rangle_\Sigma \cong \langle\mu e_2 e_1\rangle_\Sigma$ by the composition definition of *d*-graphs. Since by induction hypothesis $\langle\mu_1\rangle_\Sigma$

$\cong <\mu e_2>_\Sigma$ and $<\mu_2>_\Sigma \cong <\mu e_1>_\Sigma$, we have $<\mu_1 e_1>_\Sigma \cong <\mu_2 e_2>_\Sigma$. Thus, in both cases $<\mu_1 e_1>_\Sigma \cong <\mu_2 e_2>_\Sigma$, i.e. $<\omega_1>_\Sigma \cong <\omega_2>_\Sigma$.

Assume now $<\omega_1>_\Sigma \cong <\omega_2>_\Sigma \cong \zeta$. Since ζ is not empty, strings ω_1, ω_2 are also not empty and there are strings μ_1, μ_2, and symbols e_1, e_2 such that $\omega_1 = \mu_1 e_1$, $\omega_2 = \mu_2 e_2$. Thus, $<\mu_1 e_1>_\Sigma \cong <\mu_2 e_2>_\Sigma$. If $e_1 = e_2$, $<\mu_1>_\Sigma \cong <\mu_2>_\Sigma$ and, by induction hypothesis, $\mu_1 \equiv_\Sigma \mu_2$. It implies that $\mu_1 e_1 \equiv_\Sigma \mu_2 e_2$. If $e_1 \neq e_2$, *d*-graph ζ contains two maximal nodes labelled with e_1 and e_2. Remove both these nodes from ζ; let μ be a string such that $<\mu>_\Sigma$ is isomorphic to the resulting *d*-graph. Thus, $<\mu e_1 e_2> \cong \zeta \cong <\mu e_2 e_1>$. Clearly we have $<\mu e_1>_\Sigma \cong <\mu_2>_\Sigma$ and $<\mu e_2>_\Sigma \cong <\mu_1>_\Sigma$. By induction hypothesis $\mu e_1 \equiv_\Sigma \mu_2$ and $\mu e_2 \equiv_\Sigma \mu_1$, hence $\mu e_1 e_2 \equiv_\Sigma \mu_2 e_2$ and $\mu e_2 e_1 \equiv_\Sigma \mu_1 e_1$. Since by definition of trace congruence $\mu e_1 e_2 \equiv_\Sigma \mu e_2 e_1$, we have $\mu_2 e_2 \equiv_\Sigma \mu_1 e_1$, i.e., $\omega_1 \equiv_\Sigma \omega_2$, which completes the proof. □

Appendix D

Proof of the Properties for Extended Drum and Slider System

The following are the proofs of properties required by the extended drum and slider system which was shown in Chapter 7.

For the net shown in Fig.7.7, the following system dependent axiom and inference rules can be obtained:

System dependent axiom: $p_5 \wedge p_7 \wedge p_{11} \wedge p_{14} \wedge p_{15} \wedge p_{31}$

System dependent inference rules:

1. $p_1 \wedge \neg p_2 \Rightarrow \bigcirc (p_2 \wedge \neg p_1)$
2. $p_2 \wedge p_{13} \wedge p_{21} \wedge \neg p_3 \Rightarrow \bigcirc (p_3 \wedge \neg p_2 \wedge \neg p_{13} \wedge \neg p_{21})$
3. $p_3 \wedge p_{22} \wedge \neg p_4 \wedge \neg p_{14} \Rightarrow \bigcirc (p_4 \wedge p_{14} \wedge \neg p_3 \wedge \neg p_{22})$
4. $p_4 \wedge p_{23} \wedge \neg p_5 \wedge \neg p_7 \Rightarrow \bigcirc (p_5 \wedge p_7 \wedge \neg p_4 \wedge \neg p_{23})$
5. $p_5 \wedge p_{24} \wedge \neg p_1 \Rightarrow \bigcirc (p_1 \wedge \neg p_5 \wedge \neg p_{24})$
6. $p_2 \wedge p_{14} \wedge p_{21} \wedge \neg p_6 \Rightarrow \bigcirc (p_6 \wedge p_{14} \wedge \neg p_2 \wedge \neg p_{21})$
7. $p_6 \wedge p_{25} \wedge \neg p_1 \Rightarrow \bigcirc (p_1 \wedge \neg p_6 \wedge \neg p_{25})$
8. $p_{11} \wedge p_7 \wedge p_{16} \wedge \neg p_{12} \Rightarrow \bigcirc (p_{12} \wedge \neg p_{11} \wedge \neg p_7 \wedge \neg p_{16})$
9. $p_{12} \wedge p_{14} \wedge p_{26} \wedge \neg p_{11} \wedge \neg p_{13} \wedge \neg p_{15} \Rightarrow (p_{11} \wedge p_{13} \wedge p_{15} \wedge \neg p_{12} \wedge \neg p_{14} \wedge \neg p_{26})$
10. $p_{15} \wedge p_{31} \wedge \neg p_{32} \Rightarrow \bigcirc (p_{32} \wedge \neg p_{15} \wedge \neg p_{31})$
11. $p_{32} \wedge p_{27} \wedge \neg p_{16} \wedge \neg p_{31} \Rightarrow \bigcirc (p_{16} \wedge p_{31} \wedge \neg p_{27} \wedge \neg p_{32})$

Proof of property (7.25) $\Box (p_{12} \rightarrow \neg p_{32}) \vee \Box (p_{32} \rightarrow \neg p_{12})$

(1) $\Box (\neg p_{12} \vee \neg p_{32}) \vee \Box (\neg p_{32} \vee \neg p_{12})$ from (7.25) by $\varphi \rightarrow \lambda \Leftrightarrow \neg\varphi \vee \lambda$

(2) $\Box \neg (p_{12} \wedge p_{32})$ from (1) by logical operators $(\vee, \wedge)$ and tautology

(3) The loop $C = \{p_{12}, t_{12}, p_{15}, t_{21}, p_{32}, t_{22}, p_{16}, t_{11}\}$ shown in Fig.7.7 contains only one initial token in p_{15} and satisfies the following requirements:

i) $\forall t \in C, |{\bullet t} \cap C| = |t{\bullet} \cap C|$ (D.1)

ii) $\forall p \in C, ({\bullet p} \cup p{\bullet}) \subseteq C.$ (D.2)

According to the token invariance theorem presented in Chapter 2, this means that mutual exclusion between p_{12} and p_{32} is guaranteed. This shows that (2) is true.

Proof of property (7.26) $\Box(p_{15} \rightarrow \Diamond p_{16})$

For this property, we need consider the following two sets of places and transitions:

$C_1 = \{p_{12}, t_{12}, p_{15}, t_{21}, p_{32}, t_{22}, p_{16}, t_{11}\}$; $C_2 = \{p_{31}, t_{21}, p_{32}, t_{22}\}$

Since C_1 and C_2 satisfy the requirements (D.1) and (D.2) described above and each set only contains one token initially, the mutual exclusion between any two places in C_1 or C_2 is guaranteed.

(1)	$p_{15} \rightarrow \neg p_{32} \wedge \neg p_{16}$	from C_1
(2)	$\neg p_{32} \rightarrow p_{31}$	from C_2
(3)	$p_{15} \rightarrow (p_{15} \wedge p_{31} \wedge \neg p_{32} \wedge \neg p_{16})$	from (1) and (2)
(4)	$p_{15} \wedge p_{31} \wedge \neg p_{32} \Rightarrow \bigcirc (p_{32} \wedge \neg p_{15} \wedge \neg p_{31})$	from (3) by inference rule-10
(5)	p_{27}	by the occurrence of event L_loaded
(6)	$p_{32} \wedge p_{27} \wedge \neg p_{16} \wedge \neg p_{31} \Rightarrow \bigcirc (p_{16} \wedge p_{31} \wedge \neg p_{27} \wedge \neg p_{32})$	from (3)-(5) by inference rule-11
(7)	$\Box (p_{15} \rightarrow \Diamond p_{16})$	from (1)-(6) by temporal reasoning

Proof of property (7.27) $\Box(p_{11} \rightarrow \Diamond p_{12})$

For this property, we need consider the following four sets of places and transitions:

$C_1 = \{p_{12}, t_{12}, p_{15}, t_{21}, p_{32}, t_{22}, p_{16}, t_{11}\}$; $C_2 = \{p_{11}, t_{11}, p_{12}, t_{12}\}$

$C_3 = \{p_3, t_3, p_4, t_4, p_7, t_{11}, p_{12}, t_{12}, p_{13}, t_2\}$

$C_4 = \{p_5, t_5, p_1, t_1, p_2, t_2, p_3, t_3, p_4, t_4, p_6, t_7, t_6\}$

Since each $C_{i,}$ (i=1, 2, 3, 4) satisfies the requirements (D.1) and (D.2) described above and each set only contains one token initially, the mutual exclusion between any two places in each C_1 is guaranteed.

(1)	$p_{11} \rightarrow \neg p_{12}$	from C_2
(2)	$\neg p_{12} \rightarrow (p_{15} \vee p_{32} \vee p_{16})$	from C_1
(3)	$((p_{15} \vee p_{32}) \rightarrow \Diamond p_{16})$	from property (7.26)
(4)	$\neg p_{12} \rightarrow (p_{13} \vee p_3 \vee p_4 \vee p_7)$	from C_3
(5)	$p_{13} \rightarrow \Diamond p_3$	from property (7.17) (see Chapter 7)
(6)	p_{22}	from (5) by occurrence of event S_inserted
(7)	$p_3 \wedge p_{22} \wedge \neg p_4 \Rightarrow \bigcirc (p_4 \wedge \neg p_3 \wedge \neg p_{22})$	from (4)-(5) and C_3 by inference rule-3
(8)	p_{23}	from (7) by occurrence of event S_outed
(9)	$p_4 \wedge p_{23} \wedge \neg p_5 \wedge \neg p_7 \Rightarrow \bigcirc (p_5 \wedge p_7 \wedge \neg p_4 \wedge \neg p_{23})$	from (7)-(8) and C_4 by inference rule-4
(10)	$p_{11} \rightarrow \Diamond (p_{11} \wedge p_{16} \wedge p_7)$	from (1)-(9) by temporal reasoning
(11)	$p_{11} \wedge p_7 \wedge p_{16} \wedge \neg p_{12} \Rightarrow \bigcirc (p_{12} \wedge \neg p_7 \wedge \neg p_{11} \wedge \neg p_{16})$	from (10) by inference rule-8
(12)	$\Box (p_{11} \rightarrow \Diamond p_{12})$	from (10)-(11) by temporal reasoning