TRANSFORMING UML DIAGRAMS TO CP-NETS

TRANSFORMING UML DIAGRAMS TO COLOURED PETRI NETS

By
WENXIANG YAO, B.SC.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

M. A. Sc
Department of Computing and Software
McMaster University

MASTER OF APPLIED SCIENCE (2005) McMaster University
(Computing and Software) Hamilton, Ontario

TITLE:
TRANSFORMING UML DIAGRAMS TO COLOURED PETRI NETS

AUTHOR: Wenxiang Yao, B.Sc. (MCMASTER UNIVERSITY, CANADA)

SUPERVISOR: Dr. Ryszard Janicki, Dr. N. S. Nedialkov

NUMBER OF PAGES: ix, 91

Abstract

UML is a graphically based language to specify, visualize, construct, and document the requirements of software systems. UML is defined informally. We need a well-defined semantic base for better analysis and application.

Place/Transition nets (P/T-nets) and Coloured Petri Nets are two of the most useful languages for modelling of systems containing concurrent processes. P/T-nets are low-level Petri nets, whose tokens contain very simple information.

Coloured Petri Nets (CP-nets) are high-level Petri nets. Each token of CP-nets can carry many attributes. CP-nets have a well-defined semantics allowing formal description.

The main purpose of this thesis is to proposed a set of transitions rules, which can transform UML graphs into the Coloured Petri Nets. These rules can be used to describe and analyse concurrent systems. The verification scheme and some examples to support the method are provided.

Acknowledgements

I would like to express my sincere appreciations to my supervisors, Dr. Ryszard Janicki and Dr. N. S. Nedialkov, for their inspirations, invaluable and patient guidances through out the research of this thesis.

I am grateful to Dr. Emil Sekerinski and Dr. Michael Soltys for their careful review of this thesis and for their valuable comments.

Also I want to thank all staffs who work in the department office. I gratefully acknowledge their support and help.

Finally, I appreciate very much the support from my parent and my sister.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter provides a brief introduction to the background, purpose and outline of this thesis.

1.1 Background

1.1.1 Petri Nets and Coloured Petri Nets

Petri Nets were developed in 1962 by Carl Adam Petri, as a tool for modeling and analyzing processes [23]. In principle, Petri Nets are graphic tools with strong mathematical basis.

There are various types of Petri nets. In this thesis, a kind of Petri nets called Place/Transition nets (P/T-nets) [23] will be used. P/T-nets are one of the most used and well-known Petri Nets.

P/T-nets are graphical and mathematical modelling tools, which include places, transitions and arcs. P/T-nets are low-level Petri nets, whose tokens contain very simple information. In a large concurrent system, the solution of P/T-nets may

become very large and inaccessible.

One of the extensions of P/T-nets is Coloured Petri Nets (CP-nets) [18]. CP-nets are high-level Petri nets. Each token of CP-nets can carry many attributes. CP-nets allow a large system to be specified in a compact way.

1.1.2 UML

This section uses [11] as reference.

Identified object-oriented modeling languages appeared in the 70's and are developed in the 80's and 90's. In 1994, there were more than 50 kinds of identified modeling languages. It is difficult to choose a language which can completely satisfy most of the users.

In the middle of the 90's, incorporating different techniques becomes new trend. In 1994, Grady Booch and Jim Rumbaugh of Rational Software Corporation began to develop the Unified Modeling Language (UML).

UML is a graphically based language to specify, visualize, construct, and document the requirements of software systems. It uses mostly graphical notations to specify the design of software projects.

The development of UML was a co-operative effort, which combined the work of many UML partners. UML is organized by OMG (Object Management Group, Inc.). OMG provides the framework by which different opinions can come together to form a consensus.

UML 1.0 was published by OMG in 1996. It incorporates contributions from many partners such as Digital Equipment Corp., HP and IBM. This thesis uses the version 1.4 of UML, which was published in September 2001.

1.1.3 Formalized UML

UML is defined informally. We need a well-defined semantic base for better analysis and application. This can be done in two ways.

First way is to define a precise UML semantics using general tools, as for instance, Z [9, 10], CASL [14], Linear Temporal Logic [20], etc.

The second way is to transform UML to some specific model as automata, Petri Nets, etc [3, 25]. One possibility is to use Object Petri Net Model (OPN). Related papers on this approach are [4, 5, 25]. Another possibility is to use Coloured Petri Nets (CPN). The well-defined syntax and semantics of CPNs can be used to specify UML solutions.

1.2 Purpose

UML includes nine kinds of diagrams.

For concurrent systems, choosing suitable diagrams and building sets of corresponding transition rules are very challenging and non-trivial jobs. The most popular approaches are the *activity* diagrams [19] and *statechart* diagrams [6, 13]. In this thesis, we try a new approach and use the *class* diagram, *object* diagram, *collaboration* diagram and *statechart* diagram to describe a concurrent system.

The purpose of this thesis is to propose a set of transition rules to transform UML graphs to Coloured Petri Nets. These rules will be used to describe and analyse concurrent systems. The verification scheme and some examples to support the method are provided.

It should be emphasized that it is not necessarily true that our method is the best method for specifying concurrent systems. Some may argue that we should not start

with UML after all. However, UML specification are very popular, and any result that helps to verify their properties seems to be worth to explore.

1.3 Outline

Chapter 2 introduces concurrency, Petri Nets and Coloured Petri Nets by the classic *Dining Philosopher* (DP) problem .

Chapter 3 introduces some UML concepts and definitions. It uses an example of the DP problem to explain how to specify a concurrent problem by UML.

Chapter 4 is the main part of this thesis. This chapter proposes a set of transition rules, which transform UML diagrams to CPN diagrams. It also gives some examples to illustrate them.

Chapter 5 discusses another classic concurrent problem — the Reader and Writer problem.

Chapter 6 summarizes the contribution and future work of this thesis.

Chapter 2

Concurrency, Petri Nets and Coloured Petri Nets

This chapter provides a brief introduction to concurrent systems, Petri Nets and Coloured Petri Nets.

2.1 Concurrent Systems

In this section, some definitions are introduced first. All these definitions come from [16].

Sequential Systems: Set of actions/events are executed in a sequential manner.

Concurrent Systems: Actions/events are executed in a non-sequential manner.

Simultaneity: Two actions are executed simultaneously.

A possible definition of simultaneity is: There are two actions a and b.

$\alpha, \beta : Events \rightarrow Time$

$\alpha(a)$: the beginning of a

$\beta(a)$: the end of a

a and b are **simultaneous** $\Leftrightarrow \alpha(a) < \beta(b)$ or $\alpha(b) < \beta(a)$, i.e, if a and b are "overlap". The general axiomatic definition of simultaneity is problematic, however in most of specific applications, it is rather obvious, what simultaneous means.

Concurrency: Two actions a and b are concurrent if the orders of executions: ab, ba, $a \mid b$ lead to the same result, and their choice is non-deterministic. "|" means "simultaneous execution if allowed".

Many tools are used to solve the concurrent problems, such as Calculus of Communicating System (CCS) [24], Communicating Sequential Processes (CSP) [8], Concurrent Systems (COSY) [15] and others. In this thesis, we use Petri Nets and Coloured Petri Nets to analyse a concurrent system.

2.2 Petri Nets

Petri Nets and Coloured Petri Nets are two of the most useful languages for modelling of systems containing concurrent processes. There are many books that introduce Petri Nets and Coloured Petri Nets. In this chapter, [16, 17, 18, 23, 26] are used as references.

2.2.1 Informal Introduction to P/T-Nets

Ordinary Petri nets, which are Petri nets without colours, are low-level nets. In this section we introduce a kind of Petri nets called Place/Transition nets (P/T-nets).

P/T-nets are one of the most used and well-known Petri nets.

P/T-nets are graphical and mathematical modelling tools, which include places, transitions and arcs. A P/T-net can be represented as a directed graph.

2.2.2 Formal Introduction to P/T-nets

A P/T-net can be formally defined as follows [17].

Let $\mathbb{Z}, \mathbb{N}$ and $[A \rightarrow B]$ denote integers, nonnegative integers, and total functions from A to B respectively.

A P/T-net is a 4-tuple $PTN = (P, T, W, M_0)$. Here,

- P is a set of places. Places are indicated by circles or ellipses.
- T is a set of transitions.

  Transitions are represented by rectangles, and usually, but not always, represent actions.
- $P \cap T = \varnothing, P \cup T \neq \varnothing$.
- $W \in [P \times T \rightarrow \mathbb{Z}]$ is a weight function. The number of tokens removed or added is specified by W.
- M is a marking function, where $M : P \longrightarrow \mathbb{N}$. M_0 is the *initial marking.*

A **marking** of PTN is a function in $[P \rightarrow \mathbb{N}]$. A **place** p is a condition for a transition t if and only if $W(p,t) \neq 0$. It is a precondition if and only if $W(p,t) < 0$ and a postcondition if and only if $W(p,t) > 0$.

Token Each place contains a dynamically changing number of small black dots, which are called *tokens* [18].

Places contain tokens. Transitions may fire thereby removing tokens from their preconditions and adding tokens to their postconditions. The number of tokens removed or added are specified by W. Transitions may fire concurrent (simultaneous) if and only if they involve disjoint sets of tokens [17].

2.2.3 Example of P/T-nets: the DP Problem

In this chapter, a widely known synchronization problem: the Dining Philosopher problem will be used as an example to illustrate some basic concepts of P/T-nets and CP-nets.

Dining Philosopher Problem There are five philosophers sitting around a circular table. They alternately think and eat. To eat, a philosopher needs two forks. However, there are only five forks at this table, and each philosopher is only allowed to use the two forks nearest to him. Obviously two neighbours can not eat at the same time.

The philosopher system can be described by a P/T-net. In our solution, we assume each philosopher takes and puts down two forks *simultaneously.*

Why we need simultaneity? Suppose philosophers are allowed to take left fork first, right fork second. If five philosophers want to eat at the same time, everyone takes left fork. Then all five forks are taken, and there is no free fork left. Everyone takes a fork and is waiting for another, and nobody can eat. Obviously this is a deadlock.

The definition of *simultaneously* is given in Section 2.1. In our solution, we use

simultaneity to avoid deadlock. For more details, see reference [16].

Figure 2.1 is a general solution for the DP problem using P/T-net [16].

Explaining this solution will give more details.

- *Tokens:* Each token has two attributes: type and number. Hence $philosopher_1$ and $philosopher_2$ are different tokens. They should be described separately. These tokens are represented by black dots in places $think1$, $think2$, ... $think5$. The number of each token is 1.

- *Places:* From Figure 2.1, we can see that each philosopher has two places: *think* and *eat.* Each fork has one place $free\_fork$.

- *Transitions:* Each philosopher has two actions: *take forks* and *put down forks.*

- *Arcs*: Arcs are associated with transitions and places. The arcs with arrowheads indicate the start and end points of all processes.

- *Marking:* The *initial markings* are all places with black dots in Figure 2.1, and the token number for each of the places is 1.

- *Weight:* No number appeares on arcs. This means that the weight function is 1.

At the beginning five forks are free, and five philosophers are in the *think* place. This is called *initial marking* M_0. When a philosopher wants to eat, he must be sure that both left and right forks closest to him are available. Then he takes two forks and goes to the *eat* place. After he finishes eating, he puts down the two forks synchronously and goes to the *think* state.

From Figure 2.1, we can see that every philosopher and every fork is a token different from the others. It is necessary to describe the status of each philosopher and

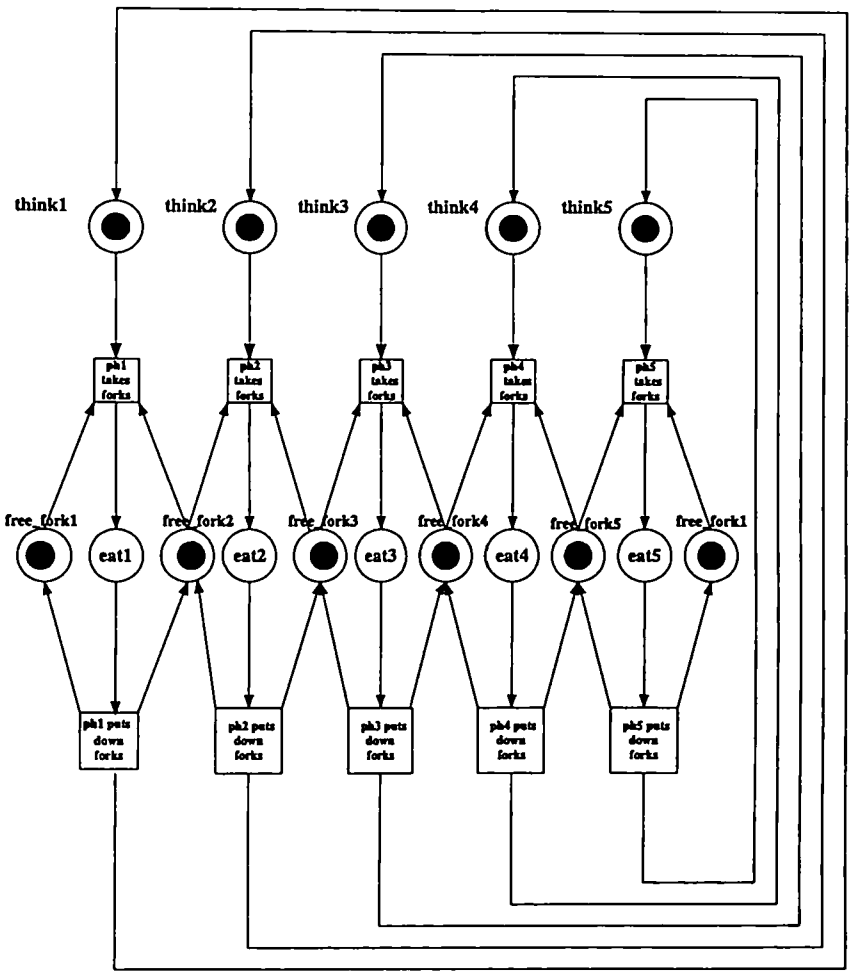


Figure 2.1: The DP problem: P/T-net solution

each fork since each is a different token, even though many processes are similar. Since the status of each fork and philosopher must be described, the diagram becomes very large and contains redundant information. This kind of problem can cause trouble for a small system, and may be catastrophic in a large system, such as a case with ten philosophers and forks.

Coloured Petri Nets can be used to solve this problem.

2.3 Coloured Petri Nets

A data value — called the *token colour* can be attached to each token. The data value may be of arbitrarily complex type. This kind of Petri nets are called **Coloured Petri Nets**. All definitions in this section are taken from [18].

2.3.1 Hierarchical CP-net and Non-hierarchical CP-net

The basic idea behind **hierarchical CP-nets** is to allow the modeller to construct a large model by combining many small CP-nets into a large one. It is same as that a large program consists of a set of modules and subroutines. After we have a set of non-hierarchical CP-nets, the next task is to combine them into a large hierarchical CP-net. The relationship between hierarchical CP-net and non-hierarchical CP-net is that each hierarchical CP-net can be translated into a behaviorally equivalent non-hierarchical CP-net and vice versa [18].

This thesis considers the **Non-hierarchical CP-net** only.

2.3.2 Informal Introduction to CP-nets

Coloured Petri Nets (CP-nets) are high-level nets. They have a graphical representation and well-defined semantics allowing formal description.

A CP-net consists of three parts: the *net structure*, the *declarations* and the *net inscriptions.*

1. **Net Structure** It includes *places*, *transitions* and *arcs.*

2. **Declaration** It includes *colour sets*, *functions*, *variables* and *constants.* It can be used in net inscriptions. In this thesis, as in [18], *CPN ML* is used to describe the *declarations.* The declaration is surrounded by a box with dashed lines.

   The declaration includes:

   - **colour sets** Each colour set declaration indicates a new colour set, whose elements are called colours.

     As an example, the following colour sets can be defined by:

     colour A = int with 1..10 (all integers between 1 and 10)

     colour Season = with Spring | Summer | Winter | Autumn

   - **functions** Each function declaration defines a function. The function takes a list of arguments and returns a result. The types for these arguments and returns are either a colour defined in colour sets or other types recognized by *CPN ML*, such as *bool.*

     For examples,

     $fun\ Fac(n) = if\ n > 1\ then\ n * Fac(n-1)\ else\ 1$

- **variables** Each variable declaration defines a list of variables, with a type which must be defined in colour set.

  For example the variables can be defined like this:

  $var\ x,y\ :\ Season$

- **constants** Each constant declaration defines a constant, with a type which is either already defined in colour sets or in another type recognized by *CPN ML*.

  For example,

  $val\ n = 4$

  declares a constant with type *int*.

3. **Net Inscription** contains various text strings which are attached to the elements of the net structure. Each net expression consists of variables, constants and functions.

In addition to the basic concepts above, the following concept is also needed.

We see the simple expression:

$S_1 = 2T + 3P$.

T and P are different transitions. The coefficients 2 and 3 indicate that how many times T and P are executed. The sign "+" is a *binding* sign, which means that we can have a step where both T and P occur synchronously. In other words, two transitions can be *concurrently enabled* if there exist bindings for the variables.

2.3.3 Formal Definition of CP-nets

All definitions in this part are taken from [18].

Before Coloured Petri nets are formally defined, we need a well-defined semantics first.

- The *elements of a type*, T. The set of all elements in T is denoted by the type name T itself.
- The *type of a variable* v is denoted by $Type(v)$.
- The *type of an expression* expr is denoted by $Type(expr)$.
- The *set of variables in an expression* expr is denoted by $Var(expr)$.
- A *binding of a set of variables* V, associating with each variable $v \in V$ an element $b(v) \in Type(v)$.
- The *value obtained by evaluating an expression*, *expr*, in *a binding*, *b* - denoted by $expr\langle b \rangle$. $Var(expr)$ is required to be a subset of the variables of b, and the evaluation is performed by substituting for each variable $v \in Var(expr)$ the value $b(v) \in Type(v)$ determined by the binding.
- An expression without variables is said to be a **closed** expression. It can be evaluated in all bindings, and all evaluations give the same value, which we often shall denote by the expression itself. We simply write "*expr*" instead of the more pedantic "$expr\langle b \rangle$".
- $\mathbb{B}$ denotes the boolean type: $\{false, true\}$
- When Vars is a set of variables, we use $Type(Vars)$ to denote the set of types $\{Type(v) \mid v \in Vars\}$.

A non-hierarchical CP-net is a tuple $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ satisfying the requirements below.

1. Σ is a finite set of non-empty types, called **colour sets.**

2. P is a finite set of **places.**

3. T is a finite set of **transitions.**

4. A is a finite set of **arcs** such that:

   $P \cap T = P \cap A = T \cap A = \varnothing$.

5. N is a **node** function, $N : A \rightarrow (P \times T) \cup (T \times P)$. N is a total function.

6. C is a **colour** function, $C : P \rightarrow \Sigma$. C is a total function.

7. G is a **guard** function. It is defined from T into expressions such that:

   $\forall t \in T$: $[Type(G(t)) = \mathbb{B} \wedge Type(Var(G(t))) \subseteq \Sigma]$.

8. E is an **arc expression** function. It is defined from A into expressions such that:

   $\forall a \in A : [Type(E(a)) = C(p(a))_{MS} \wedge Type(Var(E(a))) \subseteq \Sigma]$,
   where $p(a)$ is the place of $N(a)$.

9. I is an **initialization** function. It is defined from P into closed expressions such that

   $\forall p \in P : [Type(I(p)) = C(p)_{MS}]$.

To describe the behaviour of non-hierarchical CP-nets, we introduce some definitions.

$\forall t \in T : Var(t) = \{v | v \in Var(G(t)) \vee \exists a \in A(t) : v \in Var(E(a))\}$.

$\forall (x_1, x_2) \in (P \times T \cup T \times P) : E(x_1, x_2) = \sum_{a \in A(x_1, x_2)} E(a)$.

$Var(t)$ is called the set of **variables** of t while $E(x_1, x_2)$ is called the **expression** of (x_1, x_2).

Definition 2.1: A **binding** of a transition t is a function b defined on $Var(t)$, such as:

1. $\forall v \in Var(t) : b(v) \in Type(v)$.

2. $G(t) < b >$

By $B(t)$ we denote the set of all bindings for t.

A binding of a transition t is a substitution that replaces each variable of t with a colour. It is required that each colour is of the correct type and that the guard evaluates to true.

Definition 2.2: A **token element** is a pair (p, c) where $p \in P$ and $c \in C(p)$, while a **binding element** is a pair (t, b) where $t \in T$ and $b \in B(t)$. The set of all token elements is denoted by TE, while the set of all binding elements is denoted by BE.

A **marking** is a multi-set over TE while a **step** is a non-empty and finite multi-set over BE. The **initial marking** M_0 is the marking which is obtained by evaluating the initialization expressions:

$\forall (p, c) \in TE : M_0(p, c) = (I(p))(c)$.

The sets of all marking and steps are denoted by $\mathbb{M}$ and $\mathbb{Y}$, respectively.

Definition 2.3: A step Y is **enabled** in a marking M if and only if the following property is satisfied:

$\forall p \in P : \sum_{(t,b)\in Y} E(p,t)\langle b\rangle \leq M(p)$.

Let the step Y be enabled in the marking M. When $(t,b) \in Y$, we say that t is **enabled** in M for the binding b. We also say that (t,b) is enabled in M, and so is t. When (t_1,b_1), $(t_2,b_2) \in Y$ and $(t_1,b_1) \neq (t_2,b_2)$ we say that (t_1,b_1) and (t_2,b_2) are **concurrently enabled**, and so are t_1 and t_2. When $|Y(t)| \geq 2$ we say that t is concurrently enabled with itself. When $Y(t,b) \geq 2$ we say that (t,b) is concurrently enabled with itself.

When a step is enabled it may occur, and this means that tokens are removed from the input places and added to the output places of the occurring transitions. The number and colours of the tokens are determined by the arc expressions.

Definition 2.4: When a step Y is enabled in a marking M_1 it may **occur**, changing the marking M_1 to another marking M_2, it is defined by:

$\forall p \in P : M_2(p) = (M_1(p) - \sum_{(t,b)\in Y} E(p,t)\langle b\rangle) + \sum_{(t,b)\in Y} E(t,p)\langle b\rangle$.

The first sum is called the **removed** tokens while the second is called the **added** tokens. Moreover, we say that M_2 is **directly reachable** from M_1 by the occurrence of the step Y, which we also denote: $M_1[Y > M_2$.

From the definitions above we can see that CP-nets provide precise and well-defined semantic base to describe concurrent systems. There are some other definitions and rules of CP-nets. For detailed explanations, see [18].

2.3.4 Coloured Petri Nets and P/T-nets

P/T-nets are low-level Petri nets, while CP-nets are high-level Petri nets. The difference between the low-level nets and the high level nets is similar to the difference between assembly language and the modern programming language [18].

The low-level net has a very simple token. This kind of token has only two attributes: type and token number. In high-level Petri nets, each token can carry many attributes. This means that the token carries more complex information or data that can be used to describe a complex system [18].

A Coloured Petri net can be transformed to a P/T-net. This is done by replacing each place p with a set of places $C(p)$ (one for each kind of tokens p may hold) and replacing each transition t with a set of transitions $C(t)$ (one for each way in which t may fire).

We can also construct a Coloured Petri net from a P/T-net. However, the constructed net is not unique. Given a P/T-net, each partition of the places together with each partition of the transitions determine a Coloured Petri net. At the two extremes we obtain either a Coloured Petri net with the same number of places and transitions as the P/T-net or a CP-net with only one place and one transition. In the first case, each place and each transition have attached a set of coloures with only one element. In the second case, the single place (transition) has a colour for each place (transition) in the P/T-net. For a detailed explanation see [17].

2.3.5 Example of CP-net: the DP Problem

Figure 2.1 describes the DP system by a P/T-net. Next we can transfer this P/T-net into a CP-net [17].

First, we can replace the five places $think1$, $think2$,..., $think5$ by a single place "$think$", which can carry up to five tokens. To distinguish these philosopher tokens, we attach to "$think$" a set of colours $PH = \{ph1, ph2, ..., ph5\}$, and we demand that all tokens on "$think$" must be labelled by an element of PH. Markings of "$think$" are functions in $PH \rightarrow \mathbb{N}$. They are represented as formal sums over PH. For example, $m(think) = ph1+ph2+ph3$ represents that $philosopher1, philosopher2, philosopher3$ are thinking while $philosopher4$ and $philosopher5$ are not.

Second, same as the PH, the places $eat1, eat2, ..., eat5$ are replaced by a single place "eat" with PH as the set of possible colours.

Third, $free\_fork1, ..., free\_fork5$ are replaced by a single place "$freeforks$" with $FORK = \{f1, f2, f3, f4, f5\}$ as the set of possible colours.

Fourth, to replace the five transitions $ph1takefork, ph1takefork, ..., ph5takefork$ into a single transition "$takeforks$", we attach to the transition "$takeforks$" the set of colours, PH, representing the individual philosophers.

x,$LEFT$ and $RIGHT$ are functions from the set of colours PH attached to "$takeforks$" into the sets of colours attached to its conditions: "$think$", "eat", and "$freeforks$". The function indicate that a firing of "$takeforks$", with colour $v \in PH$, removes a token with colour $x(v) \in PH$ to "eat", and remove two tokens from "$freeforks$" with colours $LEFT(v) \in FORK$ and $RIGHT(v) \in FORK$ respectively. x is the identity function on PH. $LEFT$ and $RIGHT$ map each philosopher-colour into the colour of its left and right fork respectively.

Figure 2.2 is the part of the philosopher net after a folding where some places and some transitions are unified.

Finally, the transitions $ph1putdownforks, ..., ph5putdownforks$ are replaced by a single transition "$put\ down\ forks$" with PH as the set of possible firing colours.

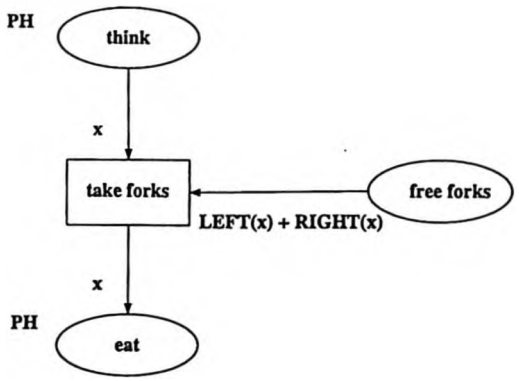


Figure 2.2: part of the DP problem

Figure 2.3 is the final CP-net solution for the Dining Philosopher problem [16]. Obviously, Figure 2.3 is much simple and more concise than Figure 2.1.

We give a more detailed explanation of Figure 2.3:

1. In the **Declaration** part:

   - There are two colour sets: Colour set **PH** and Colour set **FORK**.

     PH has 5 colours: $p1, p2, p3, p4, p5$.

     FORK also has 5 colours: $f1, f2, f3, f4, f5$.

   - One variable x — philosopher id is defined.

   - There are two functions: $LEFT(x)$ and $RIGHT(x)$. These two functions define:

     'x' is a variable;

     '$p1, ..p5, f1, ..f5$' are constants;

     Each function has five clauses, which are separated by " | ".

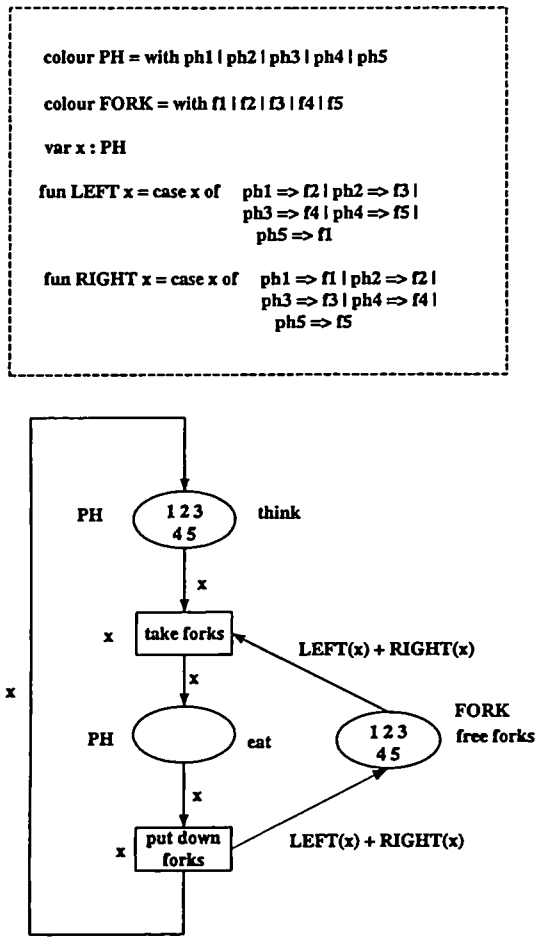


Figure 2.3: The DP problem: CP-net solution

'$LEFT(x)$' is equivalent to

"if x=ph1 then f2, else if x=ph2 then f3, ..".

2. **Net Structure** This figure consists of 3 *places*, two *transitions* and sets of *arcs*.

3. **Net Inscriptions** are strings attached to the places, transitions and declarations.

The *initial places* are PH_think and $Fork\_free\_forks$, they contain 10 colour tokens there initially.

Figure 2.3 is the initial state of the DP problem. Figure 2.4 illustrates the result when *philosopher1* and *philosopher3* fire synchronously.

This chapter describes some basic concepts of P/T-nets and CP-nets. Chapter 4 will provide a more detailed introduction to CP-nets.

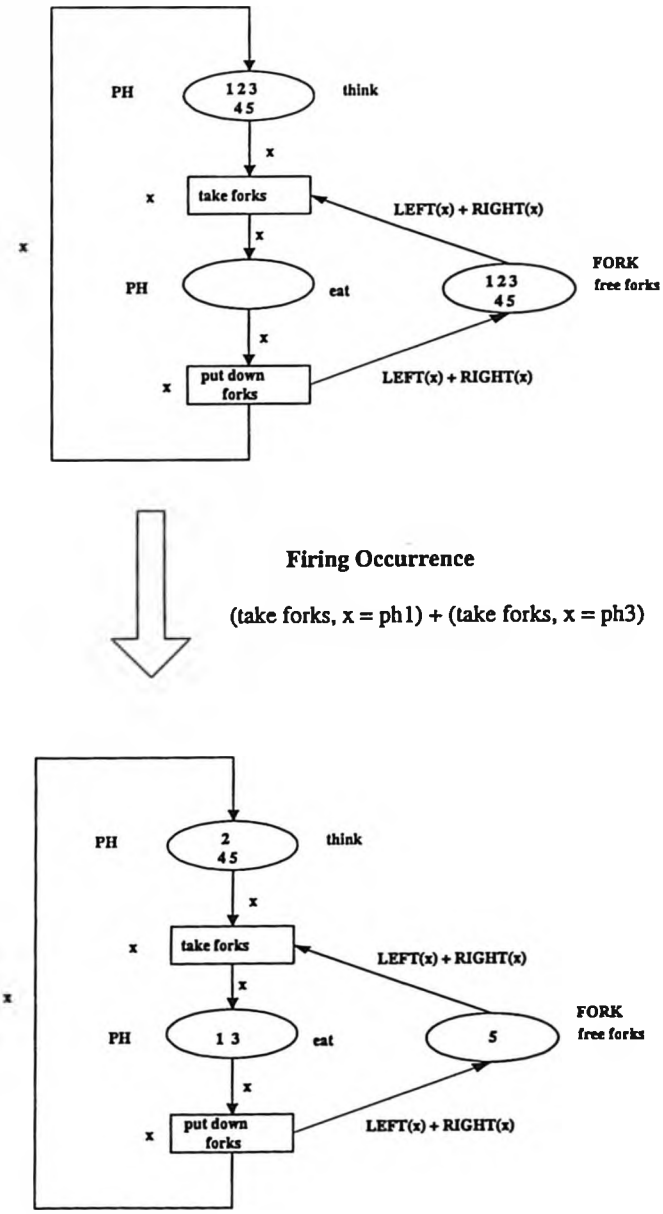


Figure 2.4: Ph1 and Ph3 fire synchronously

Chapter 3

Unified Modeling Language

In this chapter, we introduce some basic concepts of UML. We also provide our UML solution for the DP problem. Up till now, some methods are proposed to describe the DP problem by UML, such as [4, 7], etc. Most of them give partial solutions. In this thesis, we attempt to give a complete UML solution for the DP problem.

Unified Modeling Language (UML) is a graphical language. It provides a standard way to write blueprints of a system, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components [1].

The development of UML was a co-operative effort, which combined the work of many UML partners. UML is organized by OMG (Object Management Group, Inc.). OMG provides the framework by which different opinions can come together to form a consensus. UML is a gradually complete document. In this thesis, we use the version v1.4 of UML as criterion.

All definitions in this chapter are from [1]; [21] and [22] are also used as references.

3.1 Introduction to UML

Generally, UML offers the following diagram types:

- Class diagrams
- Object diagrams
- Use case diagrams
- Component diagrams
- Deployment diagrams
- State machines
- Activity diagrams
- Sequence diagrams
- Collaboration diagrams

These diagrams are different, but complement each other. They work together to describe a system. These nine diagrams are divided into three groups: *static structure diagrams*, *dynamic structure diagrams* and *architecture structure diagrams.*

Static Structure Diagrams describe the structure and functions of a system. They include *class diagrams*, *object diagrams* and *use case diagrams.*

Dynamic Structure Diagrams describe the interactive operations of a system. They include *activity diagrams*, *collaboration diagrams*, *sequence diagrams*, *state machines* and *use case diagrams.*

Architecture Structure Diagrams classify a system according to the running and executing components. They include *component diagrams* and *deployment diagrams.*

In practical applications, not every diagram is used to solve a problem. Probably 40% of UML diagrams would describe 98% of design specifications [21]. The choice of suitable diagrams is a very important step when using UML to solve a problem.

Static structure diagrams use *class diagrams* and *object diagrams* only. *Use case diagrams* are not considered in this thesis.

In dynamic structure diagrams, *sequence diagrams* and *collaboration diagrams* are equivalent, and they can be equally transformed each other. *Activity diagram* methods have already been discussed in [19], and will not be discussed here. The architecture structure diagrams are also beyond the scope of this thesis.

This thesis uses *class diagrams*, *object diagrams*, *statechart diagrams* and *collaboration diagrams* to solve concurrency problems.

This chapter uses the Dining Philosopher Problem to illustrate how to use these diagrams to specify a practical problem. The next several sections will describe these diagrams in detail.

3.2 Class Diagram and Object Diagram

3.2.1 Class Diagram and Object Diagram of UML

First, we consider *class diagrams* and *object diagrams.*

Class Diagram is a graph of classifier elements joined by their different static relationships [1]. It mainly includes class name, attributes and operations.

Object Diagram is a diagram that includes objects and their relationships at a point in time. An object diagram may be thought of as a special case of a class diagram or of a collaboration diagram [1].

There are three parts in a class diagram:

- The top compartment is **class name.**
- The middle compartment is **attribute list.**

  The default syntax is:

  visibility name: type-expression = initial-value {property-string}

  The *visibility* is:

  + : public visibility

  − : private visibility

  # : protected visibility

  Another sign is:

  / : this attribute is read only.

- The last compartment is **operation list.**

  The default syntax is:

  visibility name (parameter-list) : return-type-expression {property-string}

Association is the semantic relationship between two or more classifiers that specifies the connections among their instances [1].

In this thesis, only binary association is considered, which is the connection between two classes.

An association provides a pathway for communication. The communication can be between classes or interfaces. Associations are the most general of all relationships and consequentially the most semantically weak [2].

There are two kinds of associations. A *uni-directional* association is represented by a single arrow at one end of the association. The end with the arrow indicates who or what is receiving the communication. A *bi-directional* association represents communications between two directions.

There are many parameters in an association relationship. Only two of them are introduced in this thesis for they will be used in next part.

Multiplicity specifies the range of allowable cardinalities that a set may assume. The *cardinality* field specifies the number of expected instances of the class. For example 1 represents one instance, 0..* represents from 0 to many instances.

A role can be thought of as the "face" an element presents to the world at a particular time. The role identifies a specific behavior in a particular context at a specific time. Roles can be static (e.g., an association end) or dynamic (e.g., a collaboration role).

3.2.2 Class Diagram and Object Diagram of the DP Problem

In the Dining Philosopher Problem, we define two classes: class **Philosopher** and class **Fork**. Figure 3.1 describes the class diagrams for this problem.

Figure 3.1: Class diagrams for the Dining Philosopher Problem

Philosopher Class Diagram In the **Philosopher** class diagram, every philosopher has an *id* attribute. The values for *id* are integers from 1 to 5, which means that there are five philosophers totally.

In the Philosopher operation list, there are two operations: *take_forks*() and *put_down*().

Fork Class Diagram In the **Fork** class diagram, every fork has two attributes: an *id* number and a fork free status: *is_free*. The initial values for *id* are integers from 1 to 5, which means that there are 5 forks totally. Another attribute *is_free* is to render the fork status: free or not. The data type for *is_free* is boolean. In this thesis, we use 1 to represent "true" and 0 to represent "false". The initial values are 1, which means that all forks are free at the beginning.

We define two operations in this class: *is_taken*() and *is_putdown*(). The attribute *is_free* will be operated by using only these two operations.

Association In the DP problem, the association between two classes is represented in Figure 3.2.

First, this is a uni-directional association, sends communication message from *Philosopher* class to *Fork* class. The role identifies are *left* and *right*. This role

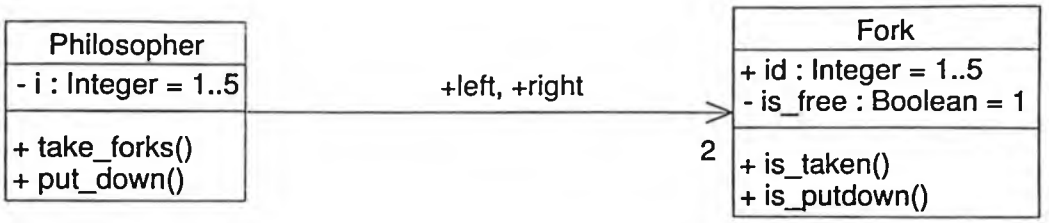


Figure 3.2: Philosopher problem: the associations between two classes

specifies a specific behavior and it is a dynamic collaboration role. The related collaboration roles can be found in Figure 3.5. This role gives the position information about philosophers and forks.

Multiplicity is 2, which means that each philosopher uses two forks each time.

Next, we will discuss two operations in *Fork* class in more details.

In Chapter 2, we assume each philosopher takes and puts down the two forks *simultaneously*. Thus in these two operations, the input parameters are $i.left$ and $i.right$. Integer i is the philosopher id, $i.left$ and $i.right$ are fork id's, which we can get from Figure 3.5. Next we use the operations *is_taken*() and *is_putdown*() to change the fork statuses.

We define *is_taken*() and *is_putdown*() as follows.

$is\_taken(left, right)$ {

....

$left.is\_free = 0;$

$right.is\_free = 0;$

....

}

The operation *is_putdown*() is defined:

$is\_putdown(left, right)$ {

....

$left.is\_free = 1;$

$right.is\_free = 1;$

....

}

We must guarantee atomicity of these two actions.

3.3 Statechart Diagram

3.3.1 Statechart Diagram of David Harel

The statechart diagram is developed by David Harel in 1987. Several variants of statecharts have been introduced, and there have also been several attempts to support statecharts with precise semantics. Harel himself has published different versions of statechart semantics, for example [12, 13]. UML has adopted statecharts and has given them semantics that differs from Harel's in several points.

For example, the definition of state in Harel's statechart is: "there are three types of states in a statechart: OR-states, AND-states and basic states. The state at the highest level is called **root**". "Each state can be associated with static reactions (SRs)", etc. These concepts only appear in Harel's statechart, not in UML statechart.

However, they are similar in many points, such as "The general syntax of an expression labeling a transition in a statechart is "$e[c]/d$," where e is the event that triggers the transition; c is a condition that guards the transition from being taken unless it is true when e occurs; and a is an action that is carried out if and when the

transition is taken"[13]. This definition is similar as the *transition* definition in UML statechart diagram.

Detailed introduction can be found in Harel's papers [12, 13]. In the next section, we introduce the definition of UML statechart diagrams. In this thesis, we only use UML statechart diagrams.

3.3.2 Statechart Diagram of UML

A **statechart** diagrams is a graph that represents a state machine. It specifies the sequence of states that an object or an interaction goes through during its life in response to events or actions [1].

The statechart diagram is used for describing the behavior of class instances. Usually this diagram is used to describe the class that has many interesting states. This thesis uses this diagram to describe the lifetime of every class.

The statechart diagram is comprised by the *states*, *events* and *transitions*.

A **state** has two parts: *Name compartment* and *Internal transitions compartment*.

The *internal transitions compartment* includes a list of internal actions or activities that are performed while the element is in the state [1]. UML reserves some special labels as below:

- **entry**

  This label identifies an action which is performed when the object enters a state.

- **exit**

  This label identifies an action which is performed when the object exit from a state.

- **do**

  This label identifies an ongoing activity, which will be performed by the object from the time it enters the state until it exits.

An **event** is an important occurrence that can trigger a transition [1]. It includes different types, such as $after(5\ seconds)$, $when()$, etc.

A signal or call event has the format:

$event - name(parameter\ list)$

Another important concept is **transition**.

Transition is a relationship between two states. It represent the fact that an object in the first state will perform certain specified actions and enter the second state when a specified event happens, and specified conditions are satisfied [1].

A transition can be labelled by a transition string:

$event - name()[guard - condition]/action - expression$

The $guard - condition$ is a boolean expression. It is executed if and when the transition fires. It can be an operation, an attribute or a link of the owning objects.

3.3.3 Statechart Diagram of the DP Problem

Figure 3.3 is a statechart diagram for the **Philosopher** class. Figure 3.4 is a statechart diagram for the **Fork** class.

Figure 3.3 shows that the **Philosopher** class has two states: *think* and *eat*. Initially there are five philosophers staying at the *think* state. These five philosophers and their relationships with five forks are represented by Figure 3.5. When philosophers are in the *think* state, they do only one action: think. The label *do* means the philosophers will keep thinking until they leave this state.

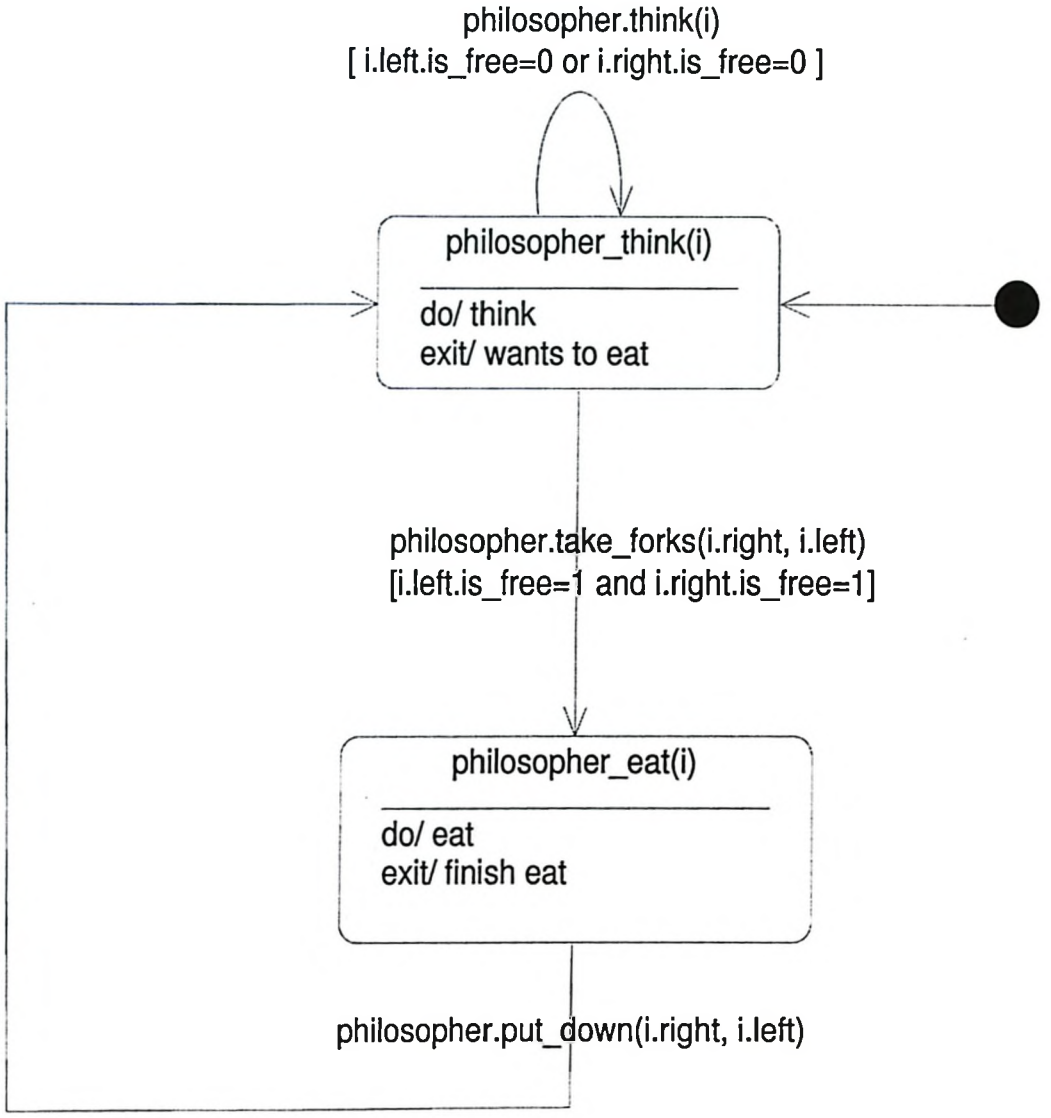


Figure 3.3: Statechart diagram: **Philosopher** class

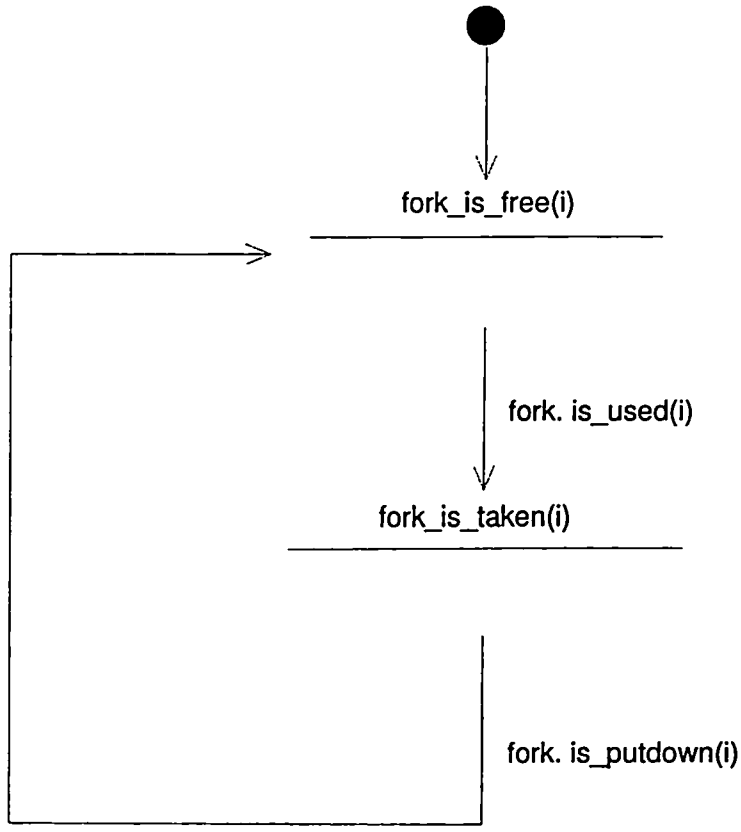


Figure 3.4: Statechart diagram: **Fork** class

When a philosopher i wants to eat, he requires the left and right forks. He keeps checking the statuses of $i.left$ and $i.right$. If one of these two forks is not free, then he will remain at the *think* state. A *transition to self* is used on this state to represent this situation. The guard condition for the *transition to self* is:

$[i.left.is\_free = 0]$ or $[i.right.is\_free = 0]$.

When both his left and right forks are free, then the event $philosopher.take\_forks(i)$ can be fired. The guard-conditions for event $philosopher.take\_forks(i)$ is:

$[i.left.is\_free = 1]$ and $[i.right.is\_free = 1]$

When event $philosopher.take\_forks(i)$ is fired, the *Philosopher* class sends an event to the *Fork* class: $fork.is\_taken(left, right)$. We defined these two operations in Section 3.2.2. The Fork class will change $i.left.is\_free$ and $i.right.is\_free$ from 1 to 0.

After the event $philosopher.take\_forks(i)$ is performed, the object *philosopher(i)* goes into another state: *eat*. When he finishes eating, he puts down forks, sends events $fork.is\_putdown(i.right, i.left)$ to the *Fork* class. The *Fork* class sets the forks to be *free* again. Then he goes back into the *think* state. This is the whole procedure for the *Philosopher* class.

Figure 3.4 is a statechart diagram for the *Fork* class. There are two states in the *fork* class: $fork\_is\_free$ and $fork\_is\_taken$. Evens $fork.is\_used()$ and $fork.is\_putdown()$ are used to change the fork statuses.

In this section, we consider the statechart diagram for the *Philosopher* class and the *Fork* class separately. This is not the final solution. This step helps us to analyse the DP problem.

3.4 Collaboration Diagram

3.4.1 Collaboration Diagram of UML

A **collaboration** diagram describes a set of objects, their relationships and their interactions. An interaction is a sequence of messages that are exchanged between the participating objects [1].

The collaboration diagram is part of the object interaction diagram. The object interaction diagram includes a collaboration diagram and a sequence diagram. It describes the dynamic structure of the objects, and emphasizes the communication structures of different objects in the system.

Compared with the statechart diagram, the collaboration diagram does not focus on the details of states. Instead, it emphasizes the messages (stimulus) between states.

In the *state* specification, normally only the name compartment is shown. The attribute and operation compartments may only be shown when needed.

The *message* is a very important part in this diagram. The default *message* syntax is:

$predecessor sequence - expression return - value := message - name arg - list$

predecessor The default syntax means that the message is not enabled until all of the communications whose sequence numbers appear in the list have occurred.

sequence-expression The order of message is rendered with a sequence of numbers, usually beginning with number 1. The sequence of numbers helps the reader know the sequence of the message. If the sequence of message is very simple and obvious, the sequence numbers do not need to be given in the diagram.

The *sequence-expression* is a dot-separated list of sequence-term followed by a colon. Each term has the following syntax:

[*interger*|*name*][*recurrence*]

The recurrence specifies *conditional* or *iterative* execution.

There are three kinds of messages which are usually found in this diagram: *synchronous*, *asynchronous* and *call-back* messages [1].

synchronous message When the receiver is executing this message, the sender does nothing except wait until the procedure has finished. This message is denoted by "– ►".

asynchronous message The sender dispatches the message and immediately continues to the next step. This message is denoted by "→".

call-back message Return from a procedure call. This message is denoted by "- - →".

An arrowhead on a line between states indicates a link or an association role with one way navigability. This means it is a one-way line.

The collaboration diagram is an important diagram in UML. In Section 3.3, the statechart diagram is used to describe the different states of every separate class. In order to describe the communications among these classes, the collaboration diagram should be used.

3.4.2 Collaboration Diagram of the DP Problem

We give the collaboration diagram of the DP problem in Figure 3.5. Figure 3.5 illustrates how the five instances of the **Fork** class and the **Philosopher** class are related. In Section 3.3.2, we use this diagram to get the values of $i.left$ and $i.right$.

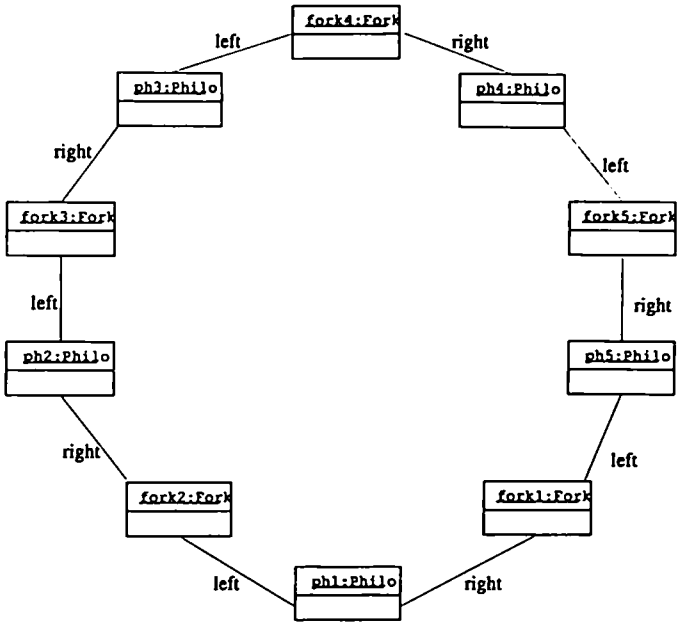


Figure 3.5: Collaboration Diagram: the DP Problem

3.5 Final Solution of the DP Problem

3.5.1 Concurrent Transition in Statechart Diagram

In Subsection 3.3.3, we use statechart diagram to describe the *Philosopher* class and the *Fork* class separately. In this part, we will combine them together. UML statechart diagram provides us a powerful tool to describe the concurrent transition.

A **concurrent transition** is enabled when all the source states are occupied. It is represented by a short heavy bar (synchronization bar) [1]. The synchronization bar also can be used in Activity Diagram of UML.

In this thesis, the synchronization bar will be used to represent concurrent events.

3.5.2 Final Statechart Diagram Solution of the DP Problem

First, we use the collaboration diagram — Figure 3.5 to describe the position information of the *Fork* class and the *Philosopher* class.

Second, we merge two statechart diagrams (Figure 3.3 and Figure 3.4) into one statechart diagram using synchronization bar. During this procedure, some states and actions are combined therefore reducing redundant information. Figure 3.6 shows that this will give us a very clear and concise solution.

We explain Figure 3.6 in more detail.

- The initial conditions are same as before.
- In Figure 3.6, there are three states only: $ph\_think(i)$, $ph\_eat(i)$, $fork\_free(k)$. One *fork* state is reduced.
- Two operations in the *Fork* class *is_taken*() and *is_putdown*() are merged into the operations in the *Philosopher* class: $take\_forks()$, $put\_down()$.
- There are two synchronization bars in this solution. The first one is a *Join* operation. The second one is the *Fork* operations.

  A *join* consists of two of more flows of control that unite into a single flow of control [2].

  A *fork* construct is used to model a single flow of control that divides into two or more separate, but simultaneous flows [2].

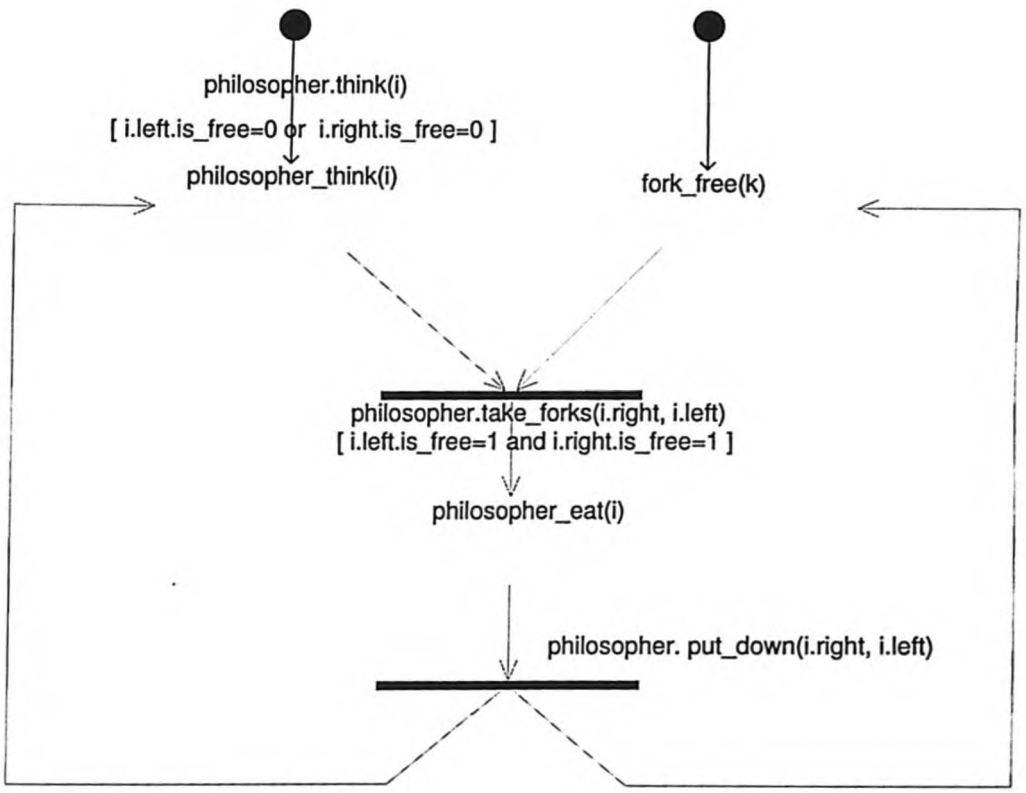


Figure 3.6: Final statechart diagram of the DP problem

3.6 Conclusion

We used an approach to develop a method in this chapter. This method can be used to solve the concurrency problem by using UML methods.

1. After understanding the problem, the *class diagram* and *object diagram* can be used to initially analyse the problem.

2. Then the *statechart diagram* can be used separately to analyse the status of every class.

3. The final *statechart* solution can be produced using *fork* or *join* operations of *statechart* diagram when there are simultaneous actions in the system.

4. The *collaboration diagram* can be used to specify the relationships and interactions of different classes. It helps the *statechart* diagram to describe the concurrent system.

Chapter 4

Transforming UML to Coloured Petri Nets

This chapter is the most important part of this thesis. It is organized into the following parts: In Section 4.2, we propose a set of transition rules. Section 4.3 introduces two special examples, both of them are often used in solving concurrent problems. Section 4.4 illustrates the verification procedure for the transition rules. Section 4.5 uses these transition rules to solve the DP problem. Finally a conclusion of this transition procedure is made.

4.1 Why it is necessary to transform UML into CP-nets

UML is a graphical informal language. The well-defined semantic tools are needed to support it. In this thesis, CP-nets are chosen to provide the formal semantic base for the UML diagrams.

UML is a standard for software engineers. Currently, it is widely used in engineering field. If we have a concurrent problem, we set up an UML model, and next we want to use a more formal, powerful tool to specify it. CPN is a good approach.

In this chapter, a set of rules for the transition from UML expressions to CP-nets are proposed. Then the well-defined CPN tools can be used to formally specify the UML solutions. This is the main motivation for this thesis.

Compared with UML, CP-nets have some advantages as follows.

- UML is a graphical representation. CP-nets also are graphical representations, but have more powerful support tools. They are similar in many places, so it is possible for them to be transformed to each other.
- CP-nets have well-defined semantics.
- UML has stubbed transition. Nested states may be suppressed. Transitions to nested states are subsumed to the most specific visible enclosing state of the suppressed state [1]. CP-nets have hierarchical structure. This property is very important. A complex system can be represented by a simple graph.
- CP-nets have a lot of formal analysis methods [17, 18]. UML can use them to do further specification.
- There are many computer tools to support CP-nets. These tools can be combined with the existing UML tools to produce more powerful tools.

4.2 Transition Rules

The transition rules for the basic UML diagrams are proposed below. The UML expressions are mapped into CP-nets.

4.2.1 Transition String

An **event** is an important occurrence that can trigger a transition [1].

An **Transition** is a relationship between two states. It represent the fact that an object in the first state will perform certain specified actions and enter the second state when a specified event happens, and specified conditions are satisfied [1].

A transition can be labelled by a **transition string** that has the following general format:

event – name() [*guard – condition*] / *action – expression*

In this thesis, UML transition strings will be transformed into CPN transitions.

4.2.2 Transition Rules

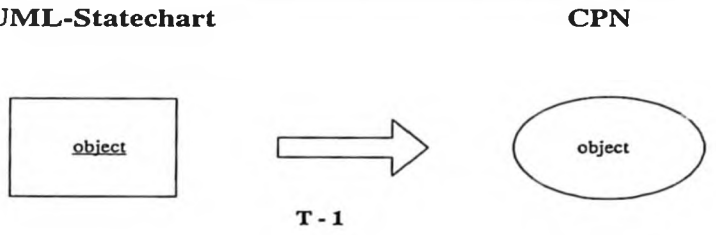
We define eight kinds of transition rules in this section. These rules are used to transform UML diagrams into CPN diagrams. In the next section, we will use some examples to illustrate how to use these rules.

Transition rules T-1, T-2 — basic places and transitions

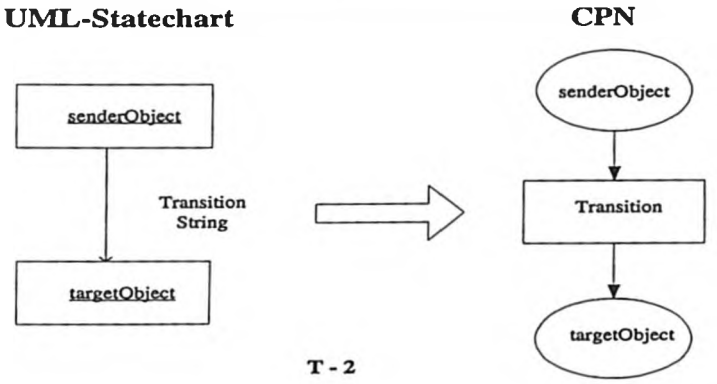
The transition rules T-1 and T-2 are shown in Table 4.1. Both of them are basic transitions.

In Table 4.1, the transition rule T-1 maps an UML statechart state into a CPN place, and the transition rule T-2 maps an UML statechart transition into a CPN transition. The states and transition strings are basic elements of UML statechart diagrams, and the places and transitions are basic elements of CPN. T-1 and T-2 are basic transition rules. Other six rules are developed on the basis of T-1 and T-2.

Figure 4.1 gives 2 examples of T-2. In example 1, the event *depositFunds()* is

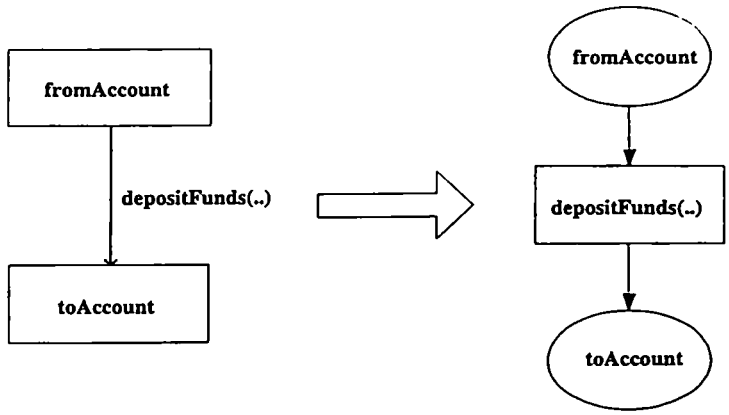


Comments: The transition rule T-1 maps an UML statechart state *object* into a CPN place with the same name. The state of UML statechart diagram is represented by a rectangle, and a CPN place is represented by an ellipse. In this thesis, we talk about a static collection of objects, restrict that one object appears in one statechart diagram only. And we ignore the state structure of objects.

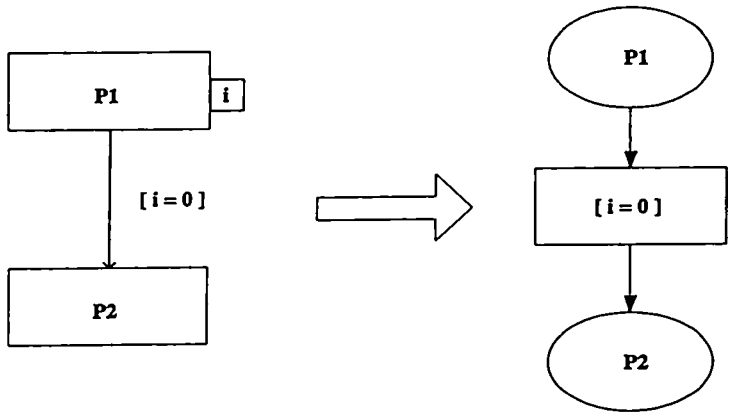


Comments: The transition rule T-2 maps an UML statechart transition string into a CPN transition with same name. Using T-1, *sendObject* and *targetObject* are converted into CPN places. The transition string of UML statechart diagrams includes *event*, *guard conditions* and *action-expression*.

Table 4.1: Transition rules T-1,T-2



T - 2 : Example 1



T - 2 : Example 2

Figure 4.1: Examples of T - 2

transformed into a CPN transition with the same name. In example 2, a guard condition [$i = 0$] is converted into a CPN transition. P1 and P2 are UML states. They are transformed into CPN places. i is a parameter associated with P1.

Transition rules T-3, T-4 — initial and final states

In Table 4.2, the transition rules of the initial states and the final states are proposed. Some examples are given to make the rules easy to understand. We add comments in the table to help understanding the transition procedures.

Transition rules T-5, T-6 — concurrent fork and join

Table 4.3 shows the *join* and *fork* transition rules when the events occur concurrently. They are the most important part of these transition rules. *Synchronization bar* is a sign for this kind of transitions.

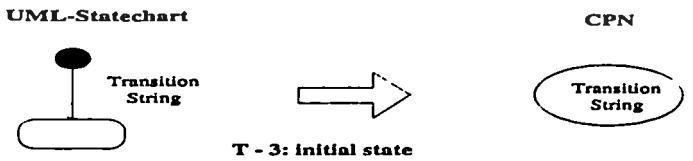
Concurrent transition: A concurrent transition includes a short heavy bar (Synchronization bar). A transition string may be shown near the bar [1].

A concurrent transition is enabled when all the source states are occupied. After a compound transition fires, all its destination states are occupied [1].

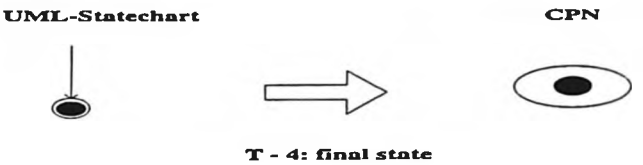
In the statechart diagrams, all source and destination states are places.

In this transition rule, before or after the transition is triggered, its source or destination states are satisfied.

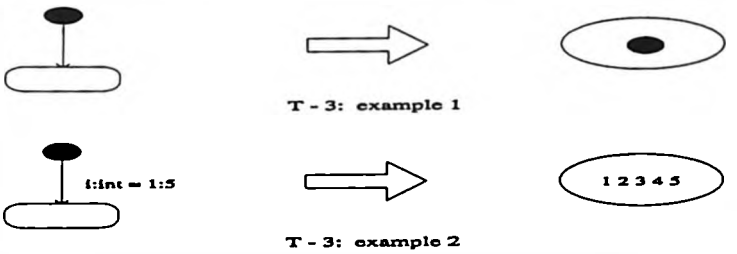
In *fork*, which means that when a transition fires, all its destination states are satisfied.



Comments: The transition rule T-3 maps a UML statechart initial state into a CPN initial place. The transition string of UML statechart diagram is converted into a CPN initial place with the same name.



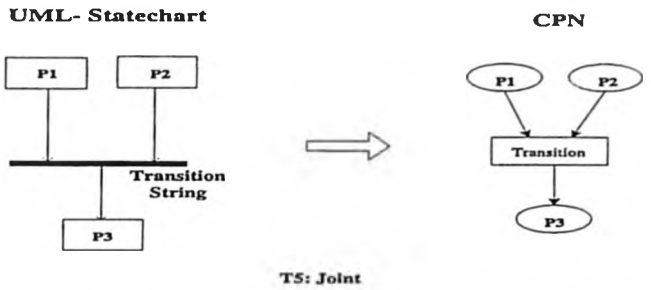
Comments: The transition rule T-4 transforms a UML statechart final state into a CPN final place.



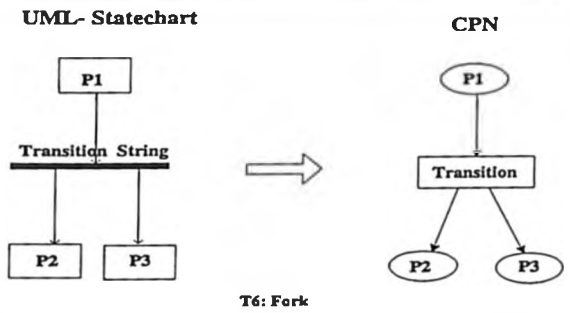
Comments: In example 1 of T-3, one object stays at the initial state of statechart diagram. Using T-3, this object is transformed into a token of CP-net staying at this place.

In example 2, 5 objects with integer ID from 1 to 5 stay at the initial state of statechart diagram. Using T-3, they are converted 5 different tokens of CP-net staying at the initial place.

Table 4.2: Transition rules T-3,T-4 and T-3 examples



Comments: The transition rule T-5 transforms the concurrent *join* UML statechart diagram into the CPN diagram. The three UML states are converted into three CPN places. The transition string of UML is mapped into the transition of CPN with the same name. The black bar is a *synchronization bar*, it is used to ensure that before the transition is triggered, all its source states are satisfied.



Comments: The transition rule T-6 maps the concurrent *fork* UML statechart diagram into the CPN transition. All descriptions of *fork* are the same as the *join* operation except when a transition fires. Then all its destination states will be satisfied.

Table 4.3: Transition rules T-5,T-6

In *join*, which means that when all source states are satisfied, then a transition is enabled.

The examples of these two transition rules will be found in next section.

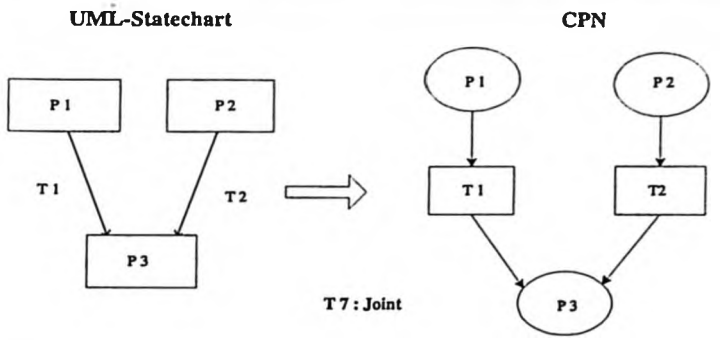
Transition rules T-7, T-8 — sequential fork and join

Table 4.4 gives us the sequential join T-7 and fork T-8 transition rules when the events occur sequentially.

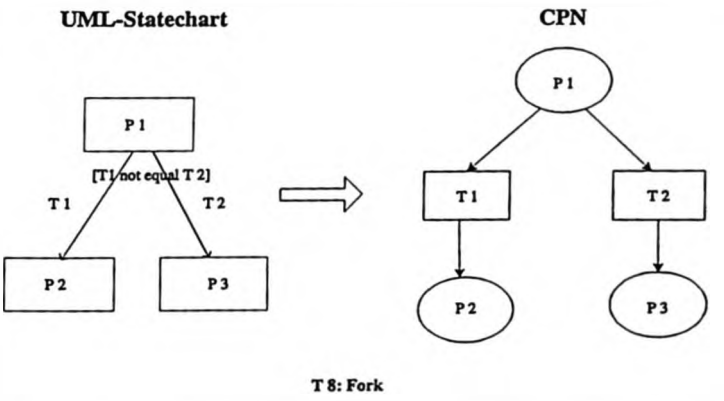
Some examples of T-8 are given in Figure 4.2. This is usually called a *branch*. *T1* and *T2* of T-8 are transition strings we mentioned before. In Figure 4.2, two examples are illustrated.

T8-1 is a guard condition example. *P1,P2,P3* are UML states, *[guard1]* and *[guard2]* are different guard conditions. A detailed example is given in Figure 4.3. In T8-1 example, *P1* has a variable i. When i has different values: $i < 0$, $i = 0$, $i > 0$, *P1* can go to different places: *P2, P3* or *P4*. This is a guard condition branch example.

T8-2 is another normal example. This example has events only. There are not guard conditions and actions. *P1* sents *event1* to *P2*, or sends *event2* to *P3*. In Figure 4.3, the example T8-2-example is used to explain T8-2. The *tentative* customer order has two results: One is confirmed by the customer, then this *tentative* order enters the *confirmed* state. Another one is cancelled by the customer, thus the *tentative* order goes to the *cancelled* state. This is an event branch example.

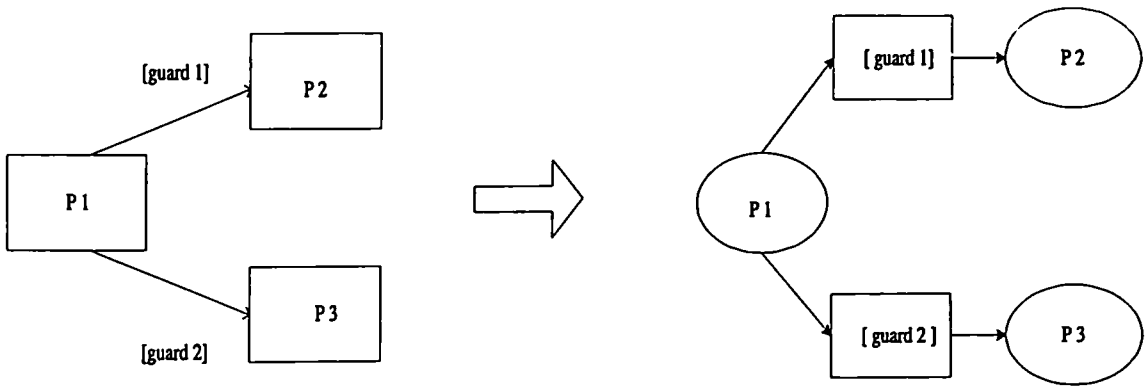


Comments: The transition rule T-7 transforms the sequential *join* UML statechart diagram into CPN transition. *P1*, *P2* and *P3* are states of UML. They are mapped into CPN places. *T1* and *T2* are transition strings of UML. They are converted into CPN transitions. This is a sequential *join* operation. There are two paths to reach P3. One path is *P1-T1-P3*, another one is *P2-T2-P3*.

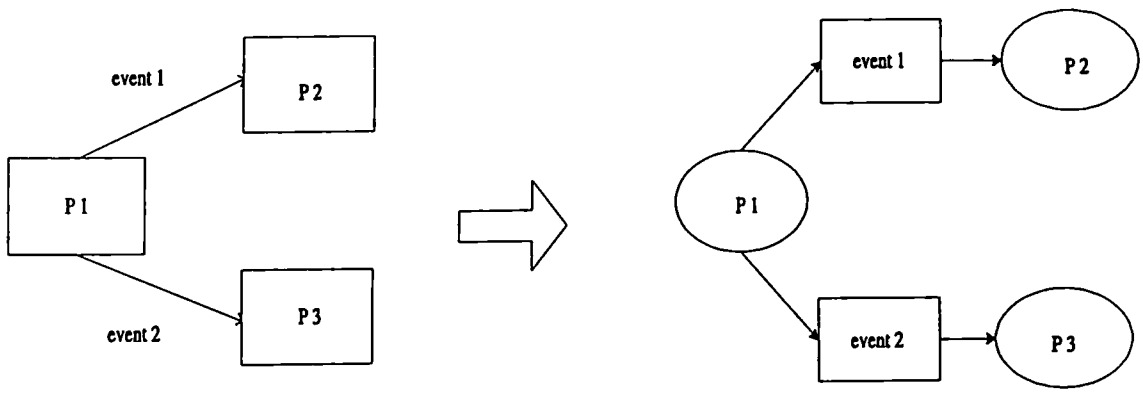


Comments: The transition rule T-8 maps the sequential *fork* UML statechart states into CPN transition. It has the same explanations as T-7, except that it is a *fork* operation. There are two paths to leave *P1*: one is *P1-T1-P2*, another one is *P1-E2-P3*.

Table 4.4: Transition rules T-7,T-8

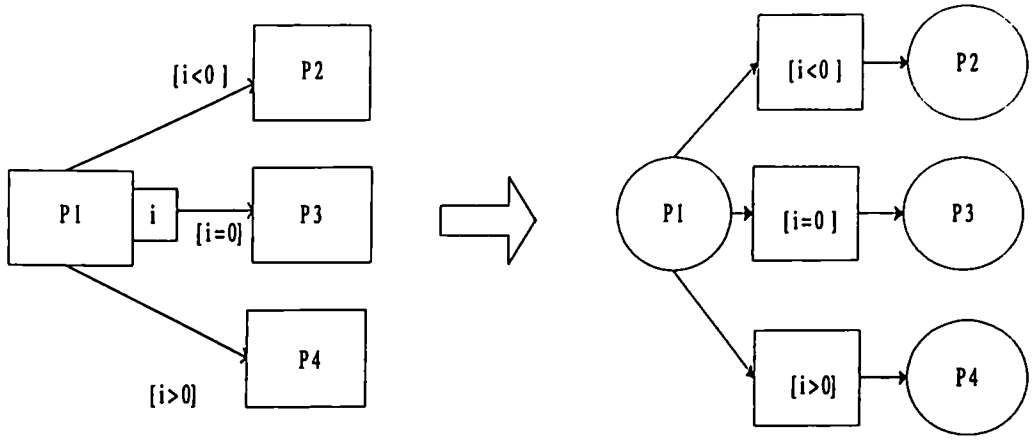


T8-1

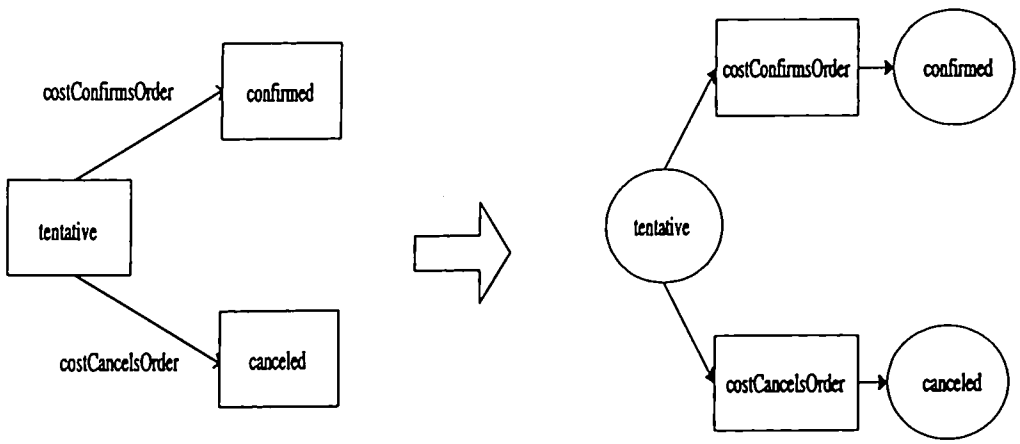


T8-2

Figure 4.2: Transition rules T8-1,T8-2



T8-1-example



T8-2-example

Figure 4.3: Transition rules 8-1,8-2: examples

Tokens

The dynamic behavior of a system represented by CPN needs a token. A token is a basic concept in CPN. There are no such concepts in UML diagrams. Hence this thesis uses some variables and functions to simulate the movement of a token.

For example, the counter i is used to represent token numbers. There are two operations in class Counter. When a token goes out, the variable $i = i - 1$ is used to describe this movement. When this token is back, the variable $i = i + 1$ is used. The value of i can be checked to know how many tokens remain in this place.

This thesis will use this technique to map the UML functions into token movings in CPN. Some examples will be provided later.

4.3 Two Special Cases

This section introduces two special cases. These cases are specified because they are the most common and important structures in a concurrent system.

4.3.1 Join and Fork - Concurrent

In a concurrent system, fork and join is a common structure.

The conversion procedure is shown in Figure 4.4. In the *fork* part, both *Transition* 1 and *Transition* 2 are transition strings. when *Transition* 1 is fired, both states S1 and S2 will be satisfied concurrently. In the *join* part, only when S1 and S2 are satisfied synchronously, then the *Transition* 2 is triggered. Their transition rules are T-5 and T-6.

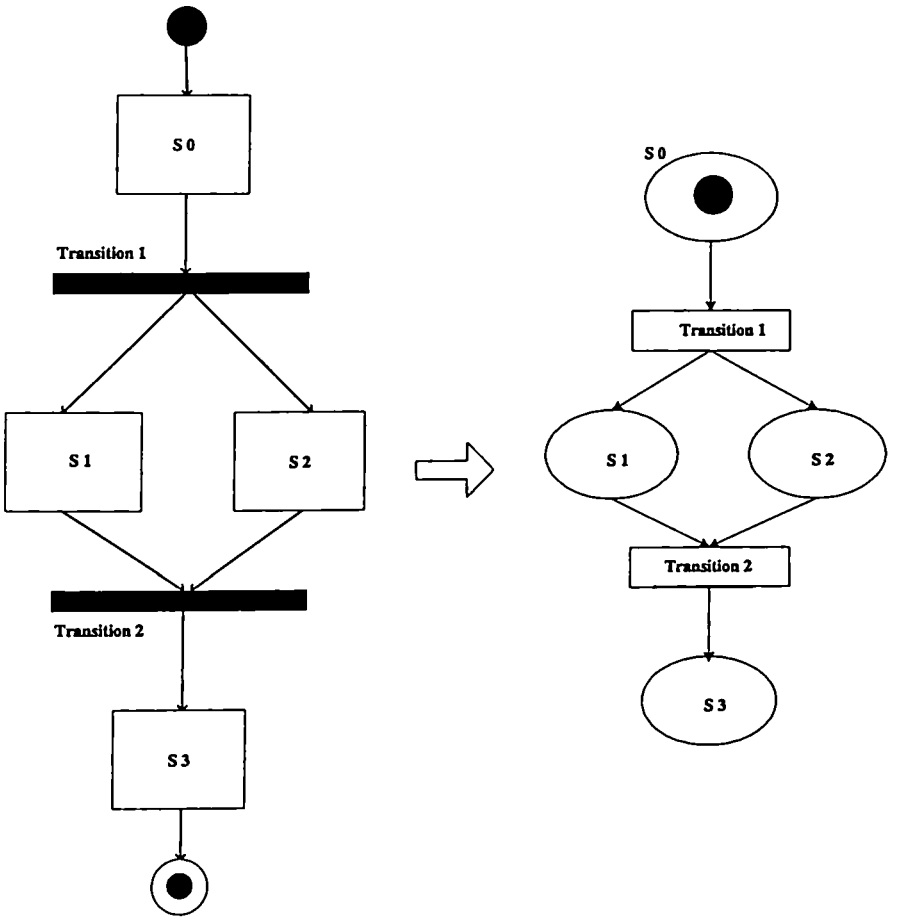


Figure 4.4: Fork and Join — concurrent

4.3.2 Mutual Exclusion Structure

Mutual Exclusion Structure is another structure, which we often use in CPN and P/T-nets. A simple mutual exclusion example of P/T-nets is shown in Figure 4.5.

In this diagram, *T0* is a fork transition, *T5* is a join transition. At beginning, a token is at the initial state *S0*. When *T0* is triggered, both *S1* and *S2* will obtain 1 token separately.

The token in *S1* has the path1:

path1: $S1 \rightarrow T1 \rightarrow S3 \rightarrow T3 \rightarrow S5$

The token in *S2* has the path2:

path2: $S2 \rightarrow T2 \rightarrow S4 \rightarrow T4 \rightarrow S6$

T5 is a join transition. Only when both *S5* and *S6* have tokens, *T5* can be triggered. Then the token goes back to *S0*.

Assuming that it is unfavorable for transition *T1* and *T2* to occur concurrently, this means that if one token stays at *S3*, then no other tokens can walk into *S3* or *S4* until *T3* is triggered. Then this token goes into *S5*, and other tokens can go into *S3* or *S4* from *S0*. This structure is called *mutual exclusion*.

The mutual exclusion structure can be used to control which transition will be triggered in a concurrent system. This structure can avoid to access common resource concurrently. In the next chapter, a detailed example is given.

A corresponding statechart diagram is represented in Figure 4.6. In Figure 4.6, *S0,S1,S2,S3,S4,S5,S6* are states, *T1,T2,T3,T4,T5,T6* are events. $[i > 0]$ is guard condition. $i = i - 1$, $i = i + 1$ are actions.

Figure 4.7 shows the conversion of the UML statechart diagram into the P/T-nets diagram.

This conversion focuses on five parts:

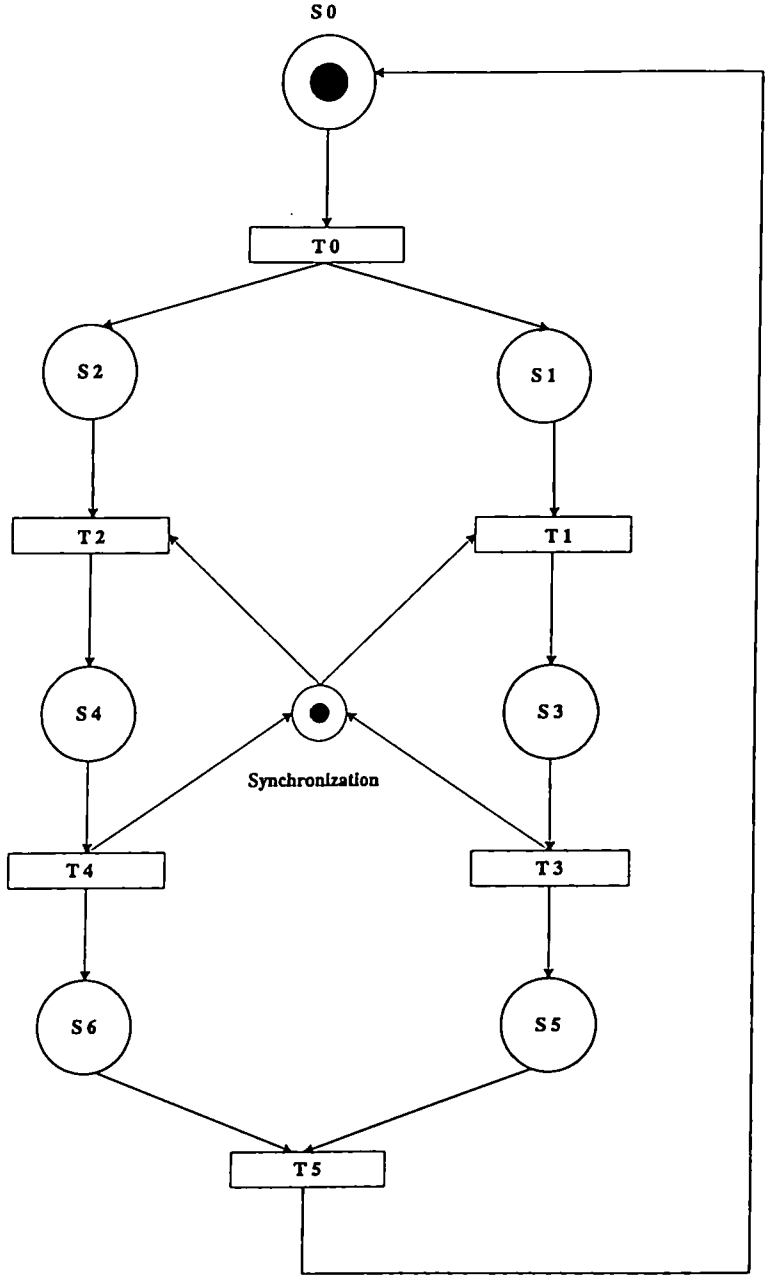


Figure 4.5: Mutual exclusion: the P/T-nets diagram

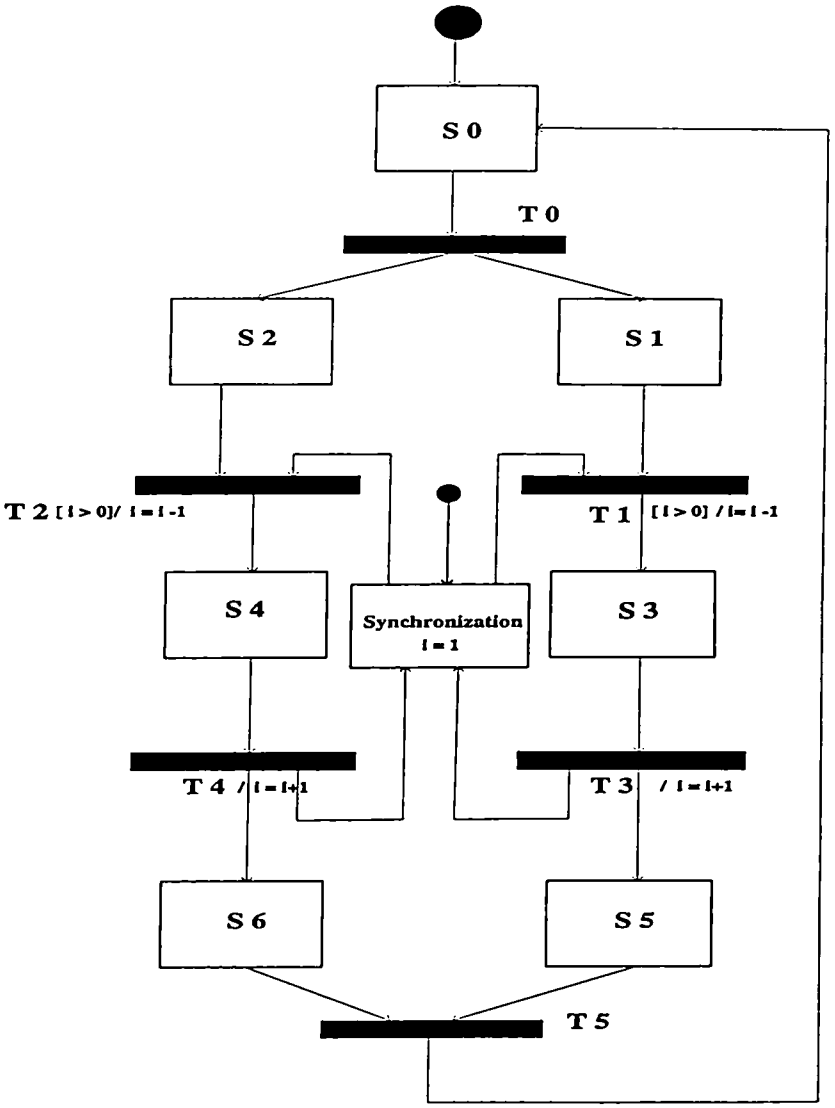


Figure 4.6: Statechart diagram: mutual exclusion

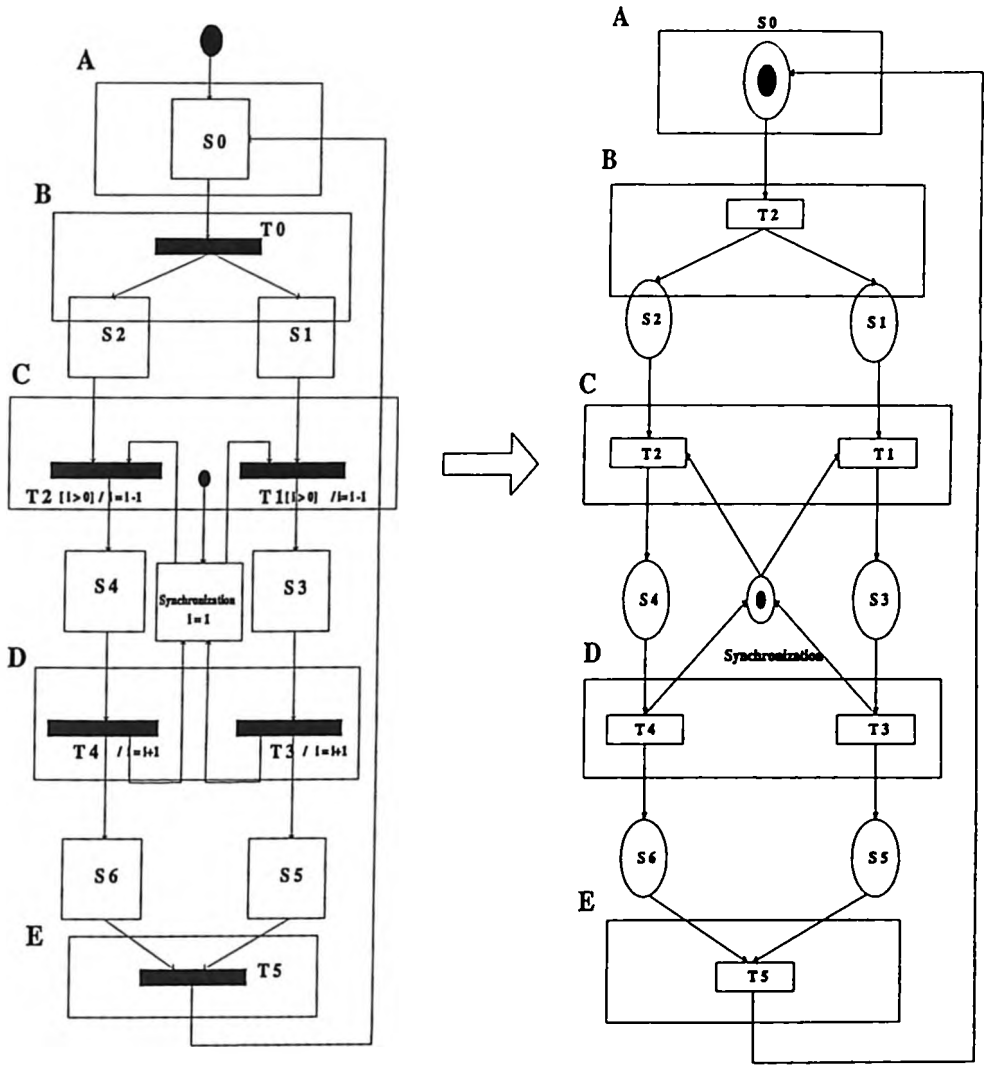


Figure 4.7: Mutual exclusion: from statechart diagram to CPN

- A: using T-3: initial state
- B: using T-6: concurrent fork
- C: using T-5: concurrent join
- D: using T-6: concurrent fork
- E: using T-5: concurrent join

4.4 Verification

4.4.1 Verification Procedure

Section 4.2 proposed some transition rules to map the UML statechart diagrams onto CPN diagrams. To be more confident in these transition rules, they must be verified.

It is necessary to find a way to simplify UML diagrams. The **flat state machine** is used to transform these complex diagrams into simple diagrams.

Flat state machine contains just simple states and arcs. Since the statechart diagram may contain hierarchical or nested states, effective conversion to Petri nets requires that the nested states be "flattened" [25].

On the other hand, all CPN diagrams can be transformed into P/T-nets, and P/T-nets can be transformed into reachability graphs. The reachability graph is a well-known P/T-nets analysis tool. The transition from CPN to P/T-nets is also a complex procedure. Many reference books and articles describing this procedure can be found. This thesis skips this step, assuming that the corresponding P/T-nets are already obtained from CPN.

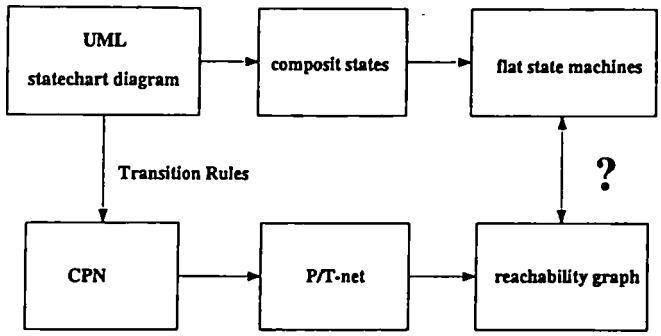


Figure 4.8: The verification procedure

This verification phase transforms the UML diagrams into the flat state machines, and the P/T-nets are transformed into reachability graphs. To complete the procedure the two graphs can be compared to prove that they are equivalent. This procedure is represented by Figure 4.8.

4.4.2 Verifying Two Special Cases

Both the flat state machines and reachability graphs are very complex graphs, since they contain many states. In a large system, it is difficult to draw all the states by hand. In this section, the two special cases we mentioned in last section are verified. These special cases were chosen because they are very common structures in concurrent system. These two examples are used to explain the verification procedure.

Fork and Join — Concurrent

Figure 4.9 shows the procedure to transform the statechart diagram of the Fork and Join (FJ) into a flat state machine. It includes three steps:

- (a) is the statechart diagram of the FJ problem.

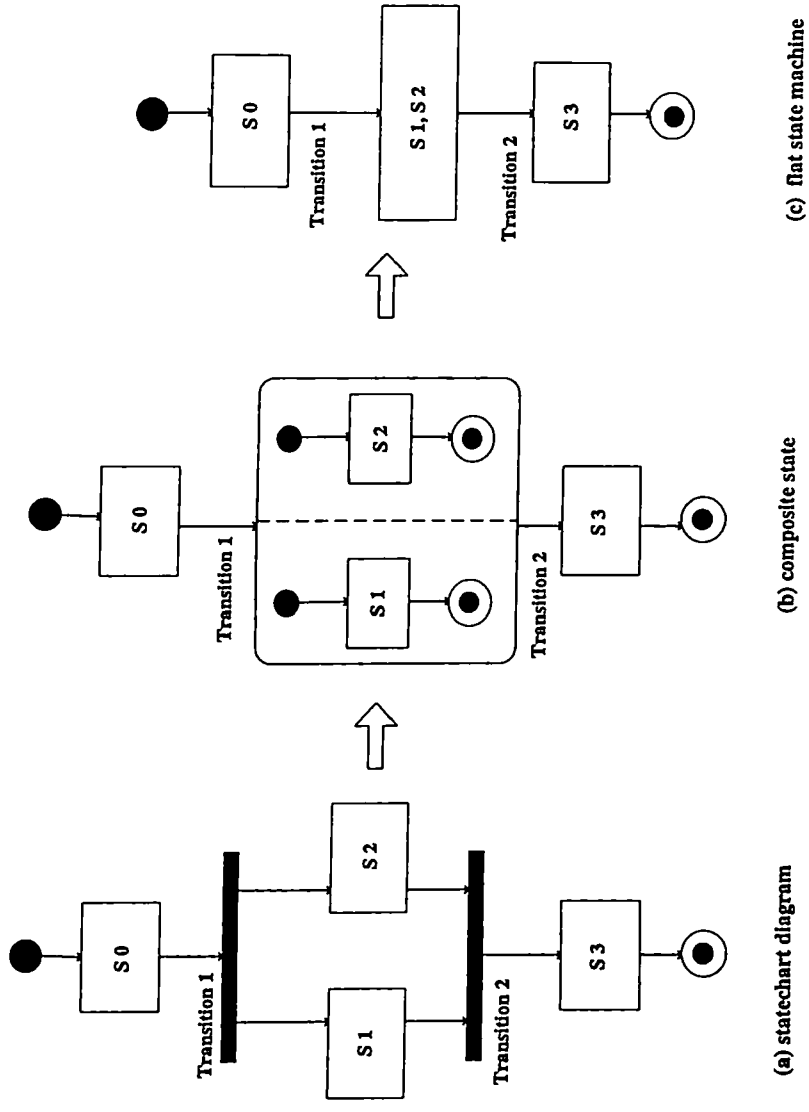


Figure 4.9: Fork and Join: Transforming the statechart diagram into a flat state machine

- (b) is a diagram which has composite states . The swim line in the middle of the diagram illustrates that two concurrent transitions occur in this diagram. The two Synchronization bars are transformed into initial and final states of *S1* and *S2*.

- (c) is the flat state machine. In (b), when *Transition 1* is triggered, two initial states of *S1* and *S2* are fired simultaneously. Then two states *S1* and *S2* are reached concurrently. Because the states *S1* and *S2* are reached and left together, we can combine them into a new state: {*S1*,*S2*}. This is the flat state machine of the FJ problem.

Next we discuss the CPN diagram. We need transform CPN diagram into the reachability graph. Figure 4.10 shows this transforming procedure. Comparing (c) of Figure 4.9 and this reachability graph, it is evident that they are identical, and the verification procedure is finished.

Mutual Exclusion Structure

Figure 4.11 transforms the statechart diagram of *Mutual Exclusion Structure* (ME) into a flat state machine. The conversion procedure includes following steps:

1. (a) is the statechart diagram of ME structure (Figure 4.6).

2. (b) is the composite state diagram transformed from (a).

   T0 and *T5* are the *fork and join* structures we mentioned in last part. We use the same method as in the last part. The synchronization bars are converted into initial and final states.

   The *Synchronization* state is between two swim lines, where the number 1 is the bound of the synch state. This object controls *T1* or *T2* is triggered.

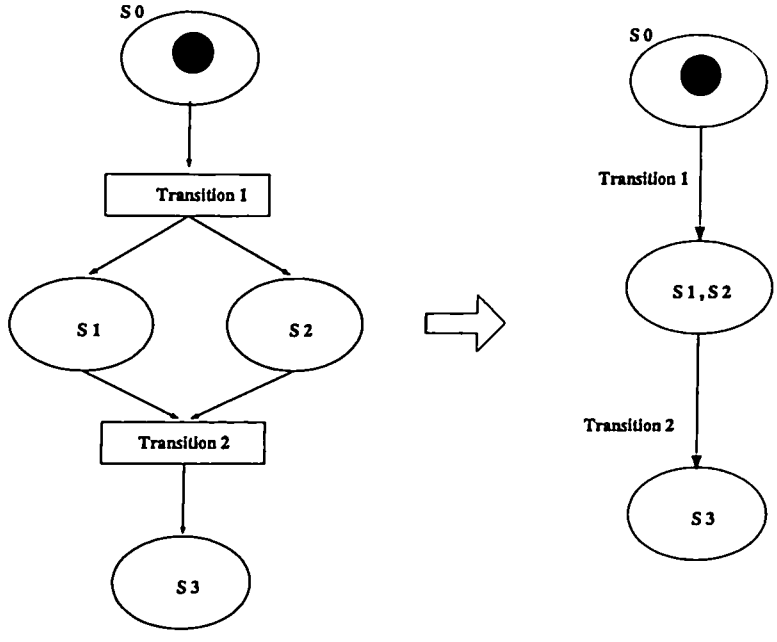


Figure 4.10: Transforming the CPN to the reachability graph

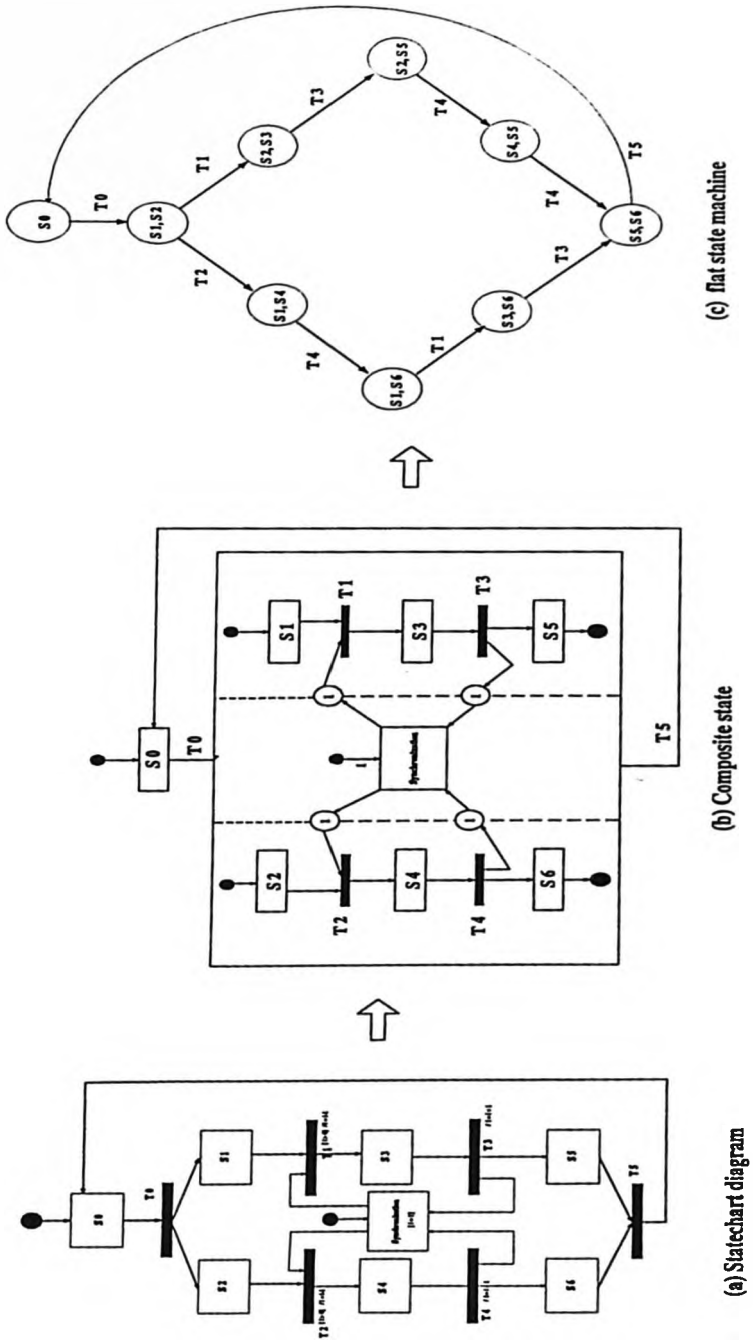


Figure 4.11: Mutual Exclusion: Transforming UML into flat state machine

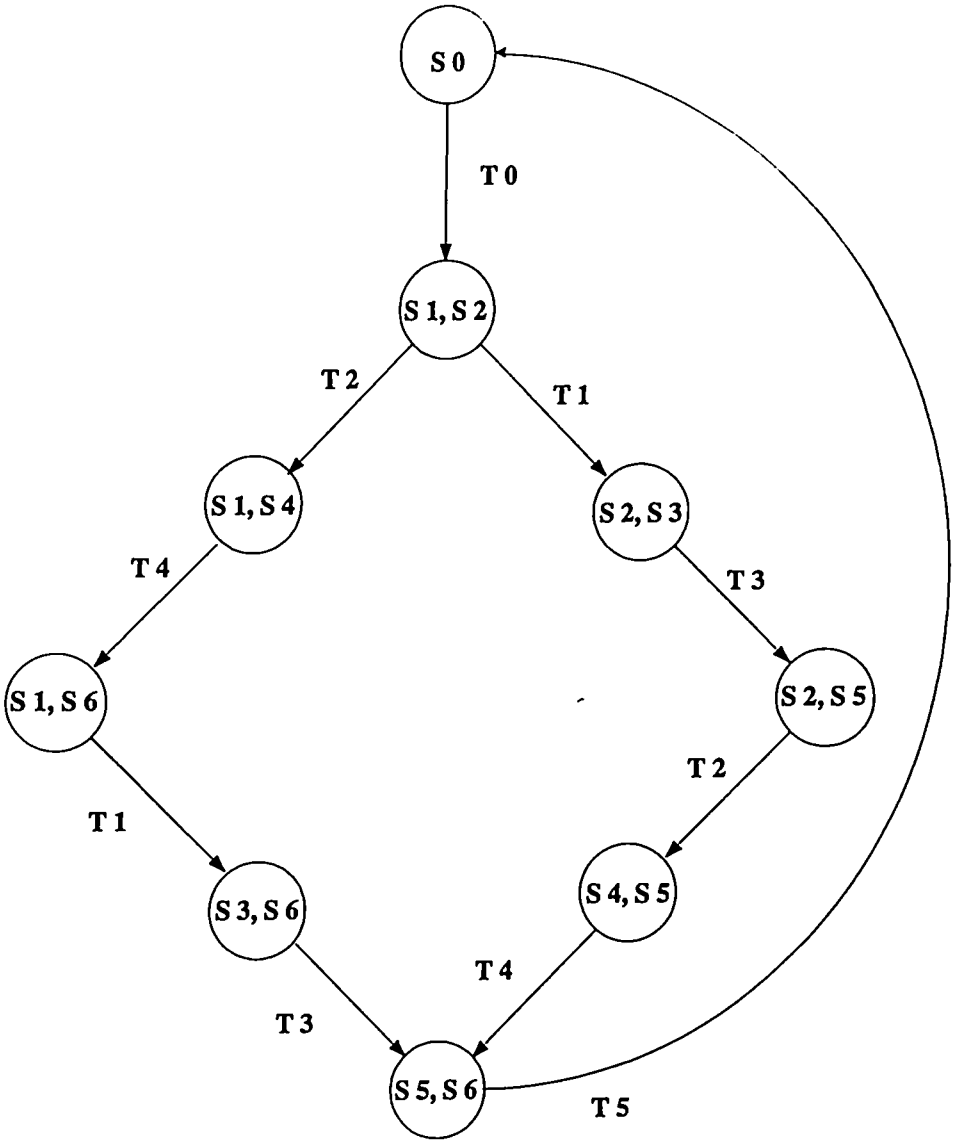


Figure 4.12: Mutual Exclusion: the Reachability graph

3. (c) is the flat state machine. In (c), all compose states of (b) can be decomposed into the single states and arcs, producing the flat state machine (c).

   At beginning, assume one object stays in *S0*. *T0* is a concurrent fork operation. When *T0* is triggered, both *S1* and *S2* have objects simultaneously. In (c), *S1* and *S2* are combined into a new state *S1, S2*.

   In the next step, there are two paths:

   path1: $S1, S3, S5$.

   path2: $S2, S4, S6$.

   Suppose *S1* gets the key — the object *Synchronization*. Then *T1* is fired, and the object of path1 enters *S3*. The object of path2 still remains at *S2*. New state *S2,S3* is produced. Next *T3* is triggered, *Synchronization* object is returned, and in the same time, the object of path1 goes to *S5*. For *T5* is a *concurrent join* operation, the object of path1 must stay at *S5*, waiting the object of path2.

   After the object of path2 gets the key and finishes path2, the state *S5,S6* is produced, then *T5* is fired, and the object returns to the initial state.

When examining this diamond shaped graph, it is evident that the center is empty. This is the effect of the synchronization. It does not allow the state $\{S3, S4\}$ to exist.

Next CPN diagram will be discussed. Figure 4.12 gives the Reachability Graph of the ME problem from the CPN diagram. It uses the well-known reachability graph technique to produce this diagram. By comparing Figure 4.12 and Figure 4.11(c), it becomes clear that they are equivalent.

In this section, two special examples are used to illustrate how to use and verify these proposed transition rules. Next, the rules will be used to solve the DP problem.

4.5 Finishing The DP Problem

The conversion of the DP problem is shown in Figure 4.13.

The conversion includes four parts:

1. In part A, the initial conditions for philosophers are transformed from the statechart diagram into the CPN diagram. Transition rule T-3 is applied.

2. Part B is the initial state for the Fork class. The transition rule T-3 is used again.

3. Part C is a *join* operation. There is a guard condition that must be satisfied — $[i.left.is\_free = 1]$ and $[i.right.is\_free = 1]$. Only when this condition is satisfied, the transition $take\_forks()$ is triggered, we get $i.left.is\_free = 0$ and $i.right.is\_free = 0$.

4. Part D is a *fork* operation. When transition $put\_down()$ is triggered, we have $i.left.is\_free = 1$ and $i.right.is\_free = 1$.

This conversion procedure shows that the DP problem is a Join-Fork operation, and is similar to the special case 1.

4.6 The Transition Procedure

4.6.1 The Iterative and Incremental Method

This section will introduce the important concept of *iterative and incremental software development method.* This section references [22].

Iteration is the repeated execution of a job.

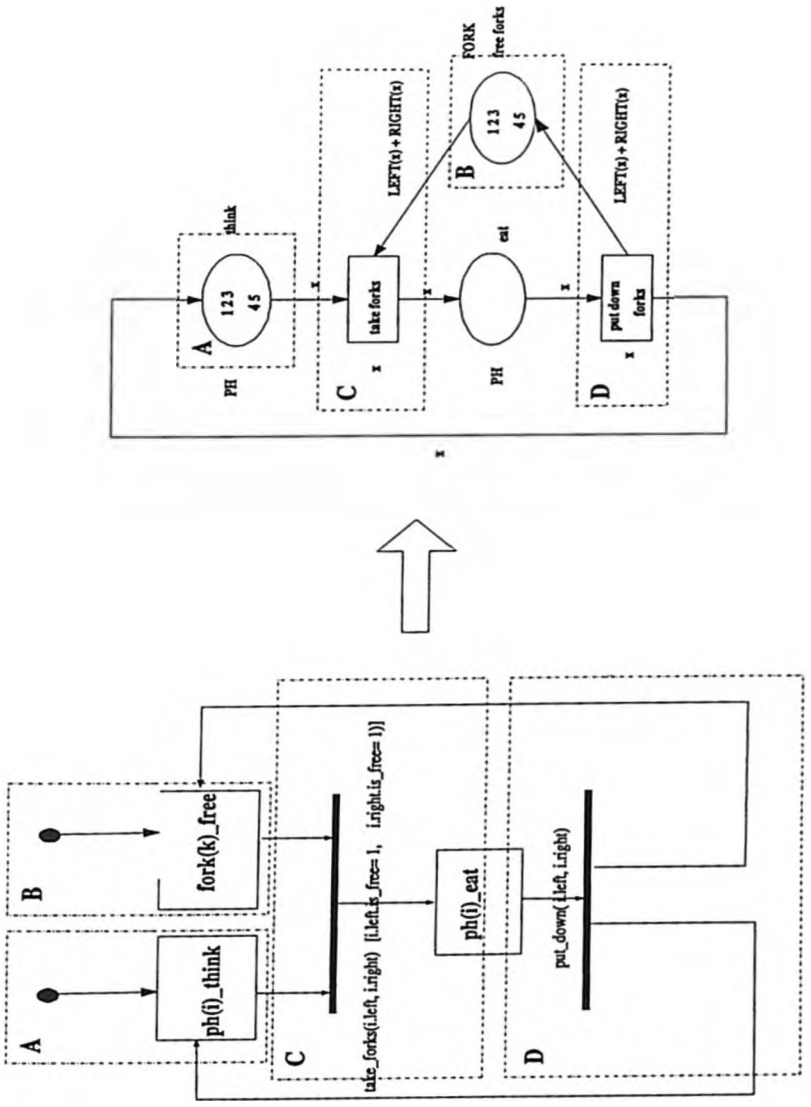


Figure 4.13: Transition rules for the DP problem

Increment Creating middle product. Every increment is an important part for the final product.

Using the increment method is an effective transition procedure. Every step is repeated until the final solution is satisfied. If there any problems found in the middle steps, the previous steps may need to be modified to solve the problem. This same method is repeated until the conversion is complete.

4.6.2 The Transition Procedure Diagram

Usually the diagram representation can provide an explanation that is more clear. At the end of this chapter, the whole transition procedure is shown in Figure 4.14.

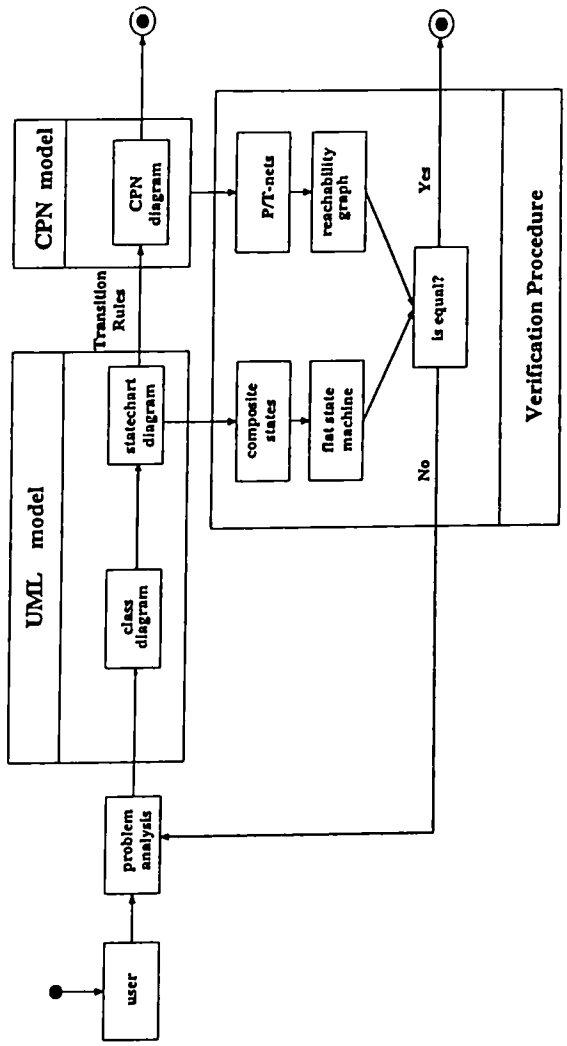


Figure 4.14: The transition procedure

Chapter 5

The Reader and Writer Problem

This chapter will use another classic synchronization problem — the Reader and Writer problem to illustrate the transition rules.

The Reader and Writer Problem There are n processes in an operating system which may read and write in a shared memory. Several reading processes can be executed concurrently, but when a process is writing, no other processes can be reading or writing. This problem will be solved step by step by using the method explained in Chapters 3 and 4.

5.1 UML Solution

The UML solution includes: class diagram, object diagram and statechart diagram.

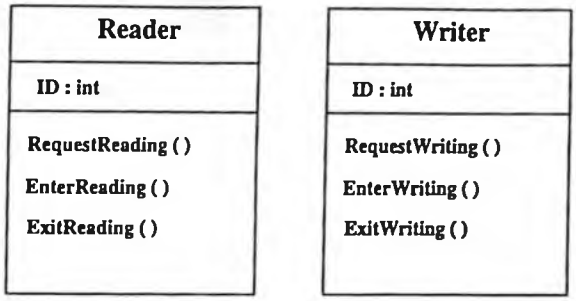


Figure 5.1: Class diagram of RW problem

5.1.1 Class Diagram

There are two classes in this RW problem: **Reader** Class and **Writer** Class. Figure 5.1 describes the class diagrams for these two classes.

In the **Reader** class, three functions *RequestReading()*, *EnterReading()* and *ExitReading()* are defined. These three functions represent three events in the statechart diagram. At this point, these functions that will be needed may be unknown, but this part can be left empty and later filled or modified when the statechart diagrams are finished. This is an iteration procedure mentioned in the previous chapter.

5.1.2 Statechart Diagram

Figure 5.2 is the *statechart diagrams* for the **Reader** class and the **Writer** class.

In the **Reader** class, the reader process will stay in one of these three places: *Local processing*, *Waiting to read* and *Reading*.

At the initial state, all processes stay at *Local processing*. Then a *RequestReading()* event occurs, the reader process comes to *Waiting to read* state. During this state the process can only wait until the *EnterReading()* event occurs and the *Reading* state can be entered. After it finishes reading, it returns to *Local processing*. This is the

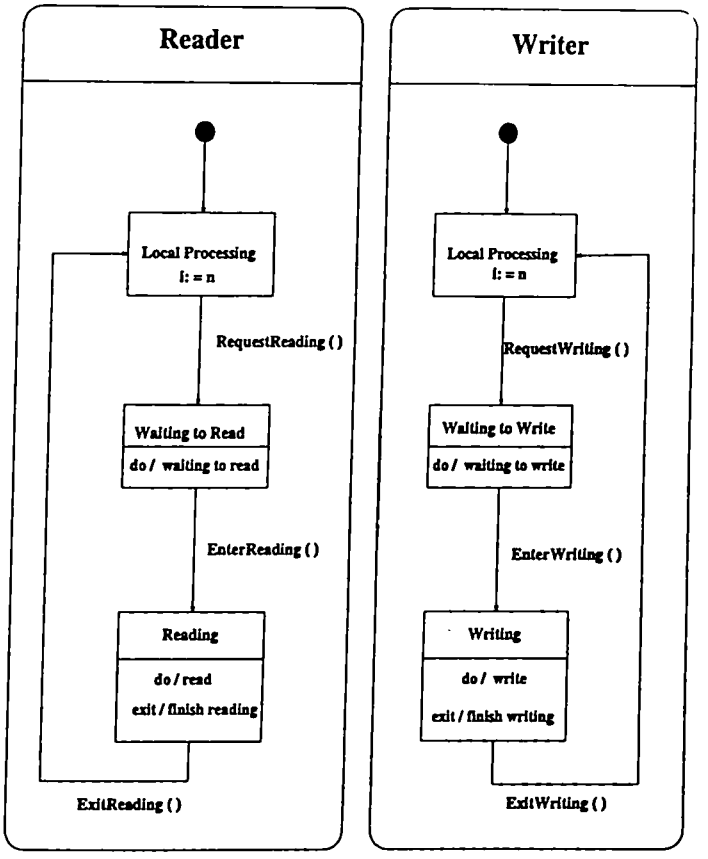


Figure 5.2: The statechart diagrams for the Reader and Writer classes

whole procedure for a reader process. The writer process is similar.

5.1.3 Preparing of the Final Statechart Diagram

The final statechart diagram is the most important and complex phase in this thesis.

The separate statechart diagrams only describe the states for one single class. The other classes do not need to be considered. But when they are put together, they communicate with each other. New problems such as deadlock or starvation will occur. To solve these problems, new controls must be added to the previous statechart and class diagrams. This is also an iteration procedure.

Figure 5.2 shows two very simple statechart diagrams for this RW problem. When these two diagrams are put together to build the final statechart diagram, new problems are encountered: How to control which process can enter the shared memory? How many reader processes can be read synchronously? How to ensure that when a writing process is writing, no other processes disturb it, etc?

To solve these problems, a **Synchronization** class must be added. This class works as a Mutual Exclusion Structure. It uses Synchronization objects to control which transition is triggered. This class includes two synchronization objects: *synchronization1* and *synchronization2.*

synchronization1: To assure that several reader processes can be reading the shared memory synchronously. But when a writer process is writing, all other processes can not enter this shared memory.

synchronization2: To assure that the RW problem has a fairness (starvation free) solution.

Figure 5.3 is the object diagrams and the statechart diagrams for *synchronization1* and *synchronization2.*

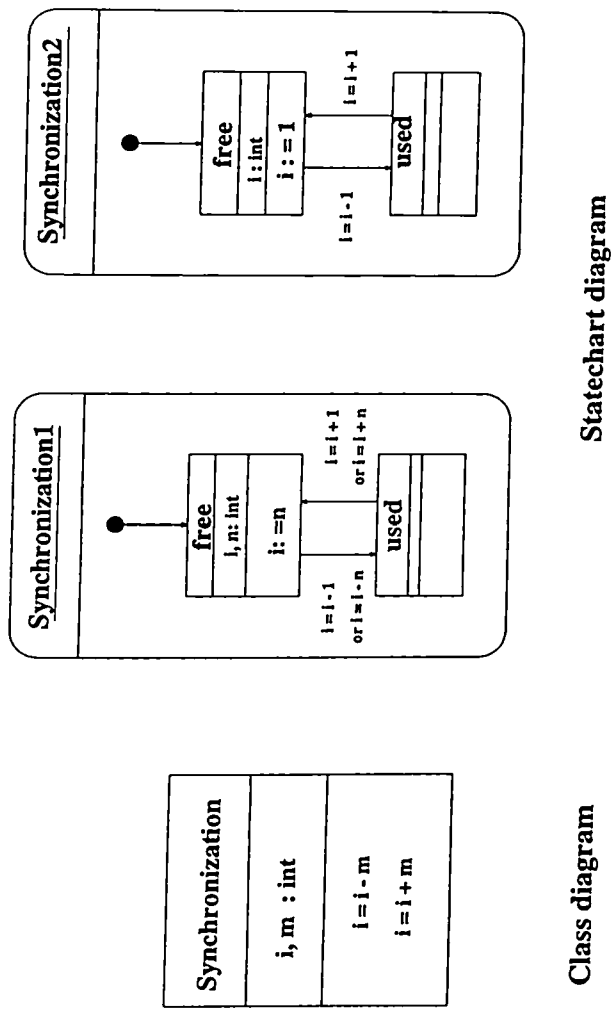


Figure 5.3: Synchronization Class: Class diagram and Statechart diagram

5.1.4 Final Statechart Diagram

In this step, the *writer*, *reader* and *synchronization* statechart diagrams must be combined together to produce the final solution. The combination procedure includes some important steps:

- To reduce the duplicate states,
- To change event communications among different classes,
- The most important thing is to add synchronization bars in the statechart diagram when it is found that some events can occur simultaneously.

The result in Figure 5.4 is the final statechart diagram for the RW problem.
To give a more detailed explanation:

- *states:* After combination, there are two synchronization free states left. On the other hand, both reader and writer requests come from the same LP, so two LPs can be united into one. Finally 7 states are present.
- *initial states:* There are three initial states.

  LP: $k := n$ represents that there are n processes initially. When a reader or writer process goes out, action $k = k - 1$ occurs. The value of k is used to control the number of processes in LP.

  Synchronization2: $i := 1$ represents that only one Synchronization object can be used.

  Synchronization1: $j := n$ represents that there are n Synchronization objects initially.

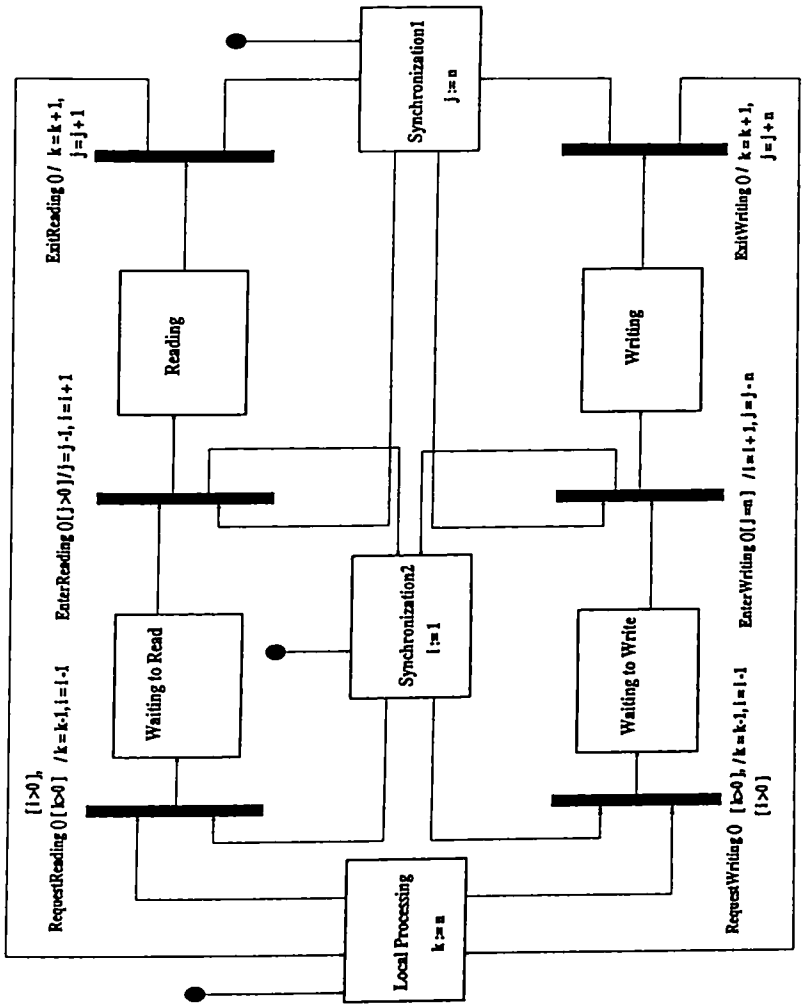


Figure 5.4: RW problem: Statechart Diagram

- *transitions:* Figure 5.2 shows that all events happen sequentially in a single class. But when the three classes are combined together, some events will occur concurrently.

As an example, a reader process is described in detail.

When a reader process occurs, except event *RequstReading*(), two guard conditions must be satisfied. $[i > 0]$ means that the object *Synchronization2* is in *free* state or it is available to be used. $[i > 0]$ means that one or more objects stay at the *Local Processing.*

When two conditions are satisfied concurrently, the event *RequstReading*() is triggered, and at same time, two actions: $k = k-1$ and $i = i-1$ occur synchronously. $k = k-1$ represents one process walks out LP, hence there are (n-1) processes left in LP. $i = i-1$ is a very important action. $i := 1$ is the initial condition, after $i = i-1$ is executed, $i := 0$, thus no more *RequestReading*() or *RequestWriting*() can occur until $i := 1$ again. This is how the *synchronization2* controls processes to work. This transition is a *join* transition.

Next the reader process goes into the state *Waiting to Read.* Before event *EnterReading*() is triggered, one object of *synchronization2* must be available $([j > 0])$.

After these two conditions are satisfied synchronously, the event *EnterReading*() is triggered. At the same time, two other actions occur. $i = i + 1$ makes *synchronization2* available again $(i := 1)$. Hence the other writer or reader processes can go out from LP. $j = j - 1$ causes *synchronization1* to reduce an object. This is a *join and fork* transition. Finally the reader process goes into the state *Reading* to read. After it finishes, the event *ExitReading*() is triggered, and $k = k + 1$ adds one process in LP, $j = j + 1$ also causes *synchronization1* to increase one object too.

The writer processes are similar to the reader processes, except for the *synchronization*1 part.

The initial condition for *synchronization*1 is $j := n$. This means that there are n objects initially. When an event *EnterReading*() is triggered, $j := j - 1$. For the initial condition is $j := n$, thus at most n processes are reading concurrently.

But the writer process is different. Before the event *EnterWriting*() occurrs, another guard condition $[j := n]$ must be satisfied. This means that there is no any reader or writer process in the shared memory (otherwise $j < n$). After *EnterWriting*() is triggered, two other actions, $i = i + 1$ and $j = j - n$, occur synchronously. $i = i + 1$ has the same meaning as the reader process. $j = j - n$ means $j := 0$. When the writer process is in wring state, no other processes can enter *Reading* or *Writing* state. After it finishes writing, *ExitWrinting*() is triggered, and $j = j + n$ lets $j := 0 + n = n$. This difference comes from the different requirements needed for reader and writer processes.

5.2 Transforming a Statechart Diagram to P/T-Nets

Figure 5.4 is the final statechart diagram for the RW problem. In this section the transition rules, which were defined in chapter 4, are used to transform the statechart diagram to P/T-nets. This problem can be easily solved by P/T-nets, so CP-nets are not used. Technically P/T-nets can be seen as a special case of CP-nets, only one colour: Token.

The transition procedure is shown by Figure 5.5. The transition steps for the reader processes are as follows:

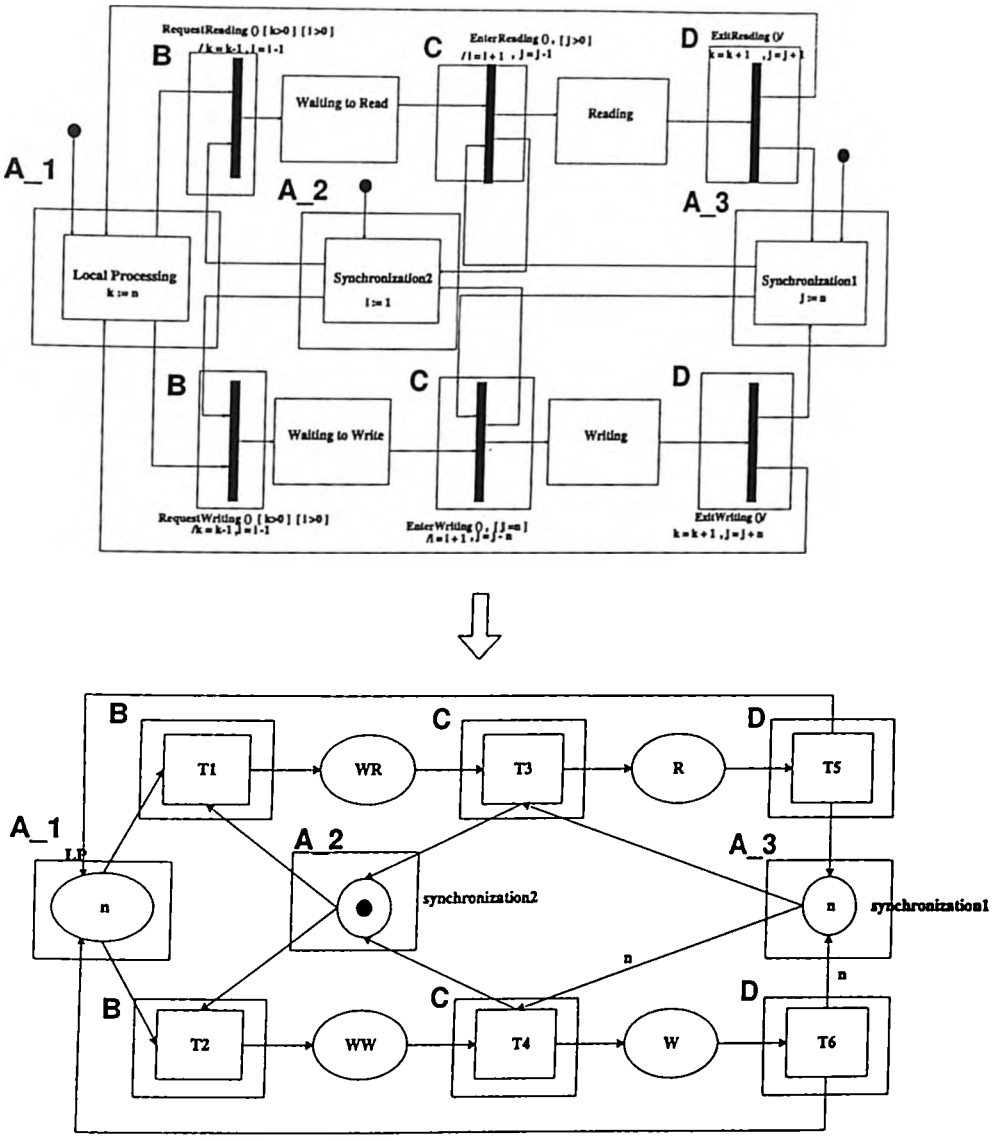


Figure 5.5: RW problem: Transforming procedure

1. All states are mapped into same states in P/T-nets.

2. Part A is the *initial* transitions.

   *A*_1: T-3. We transform $k := n$ into n tokens.

   *A*_2: T-3. We transform $i := 1$ into one synchronization token.

   *A*_3: T-3. We transform $j := n$ into n synchronization tokens.

3. Part B is a *join* transition.

   B: T-5. In P/T-nets the corresponding transition has *RequestReading* only. $k = k - 1$ and $i = i - 1$ in P/T-nets are not needed, because they correspond to the tokens moving. When a token walks out of the state *synchronization*2, there is no token there, thus it is equivalent to $i = i - 1$. $k = k - 1$ has the same situation, making the P/T-nets diagram look simple.

4. Part C is a *fork* and *join* transition.

   C: T-5 and T-6. *EnterReading* is used without actions $i = i + 1$ and $j = j - 1$.

5. Part D is a *fork* transition.

   D: T-6. We transform the transition into *ExitReading*, and set $k = k + 1$ and $j = j + 1$.

6. The writer processes is in the same situation.

There are some common rules from above transitions: states usually of UML are mapped onto same name places of P/T-nets; transition strings are mapped onto same name transitions of P/T-nets but remove the corresponding token moving parts.

5.3 P/T-nets Solution

This section describes the common P/T-nets solution for the RW problem.

In Figure 5.6, there are five states for the reader and writer processes. Each process is in one of these five states. They are:

- LP: Local processing
- WR: Waiting to read
- WW: Waiting to write
- R: Reading
- W: Writing

In the initial state, all processes stay in LP, LP contains n tokens.

There are other two states:

- *Synchronization2*: Key to control T1 and T2
- *Synchronization1*: Key to control T3 and T4

The place *Synchronization2* contains 1 token. It is used to control which token can go to the place WR or WW.

The place *Synchronization1* contains n tokens. It corresponds to the number of processes which are allowed to read on the shared memory concurrently.

We also have set of transitions:

- T1: Request Reading
- T2: Request Writing

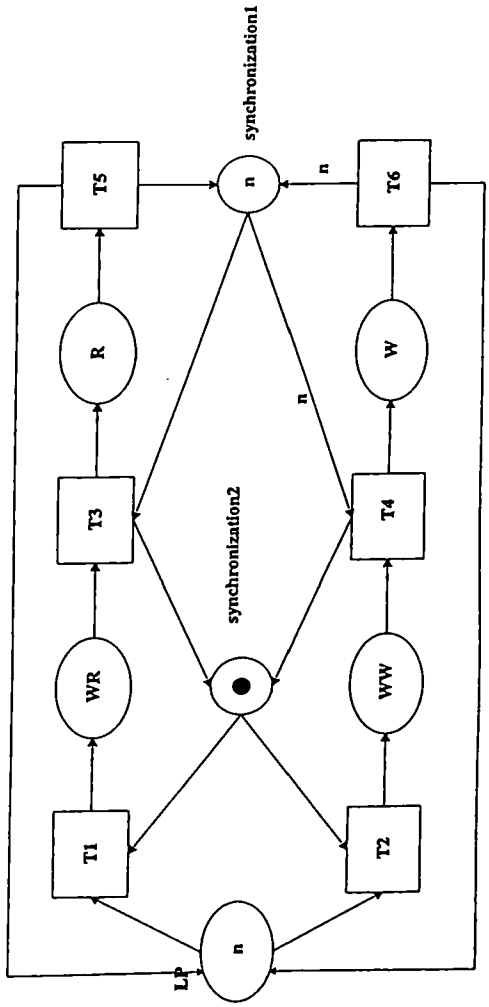


Figure 5.6: P/T-nets solution for the RW problem

- T3: Enter Reading
- T4: Enter Writing
- T5: Exit Reading
- T6: Exit Writing

The comparison of the transition result and this common solution shows that they are equivalent.

5.4 Verification Procedure

The RW problem and Mutual Exclusion (special case 2) are similar, except for the presence of extra *synchronization1* in the RW problem. The verification graph should be very similar to Figure 4.14 and 4.15 but more complex. This verification procedure is omitted in this section.

Chapter 6

Conclusions and Future Work

This chapter summarizes the contributions of this thesis, and suggests future research in this filed.

6.1 Contribution

Transforming the UML diagrams to CPN or P/T-nets is a new field, with not many papers on this topic. Most of papers give partial solutions, and focus on some aspects of this field.

On the other hand, UML is a very large system. It includes 9 kinds of diagrams. When faced with a concurrent problem, it is difficult to decide on what diagram to use.

This thesis has presented a method to transform the UML diagrams into the CPN diagrams. It has also provided a verification and given some examples to illustrate the transition procedure.

The major contribution of this thesis is to provide a method to transform the

Chapter 6

Conclusions and Future Work

This chapter summarizes the contributions of this thesis, and suggests future research in this filed.

6.1 Contribution

Transforming the UML diagrams to CPN or P/T-nets is a new field, with not many papers on this topic. Most of papers give partial solutions, and focus on some aspects of this field.

On the other hand, UML is a very large system. It includes 9 kinds of diagrams. When faced with a concurrent problem, it is difficult to decide on what diagram to use.

This thesis has presented a method to transform the UML diagrams into the CPN diagrams. It has also provided a verification and given some examples to illustrate the transition procedure.

The major contribution of this thesis is to provide a method to transform the

statechart diagrams of UML to CPN diagrams.

Another contribution is to try an approach to describe the classic concurrent problem: *the Dining Philosopher problem* and *the Writer and Reader problem* by the UML diagrams. This thesis tries to give complete specifications for these two problems.

6.2 Future Work

There are several areas of this paper that can be extended in future research.

One area is to develop this method from the non-hierarchical CPN into the Hierarchical CPN. In UML, there is a concept: *stubbed transitions*, which indicates a transition connected to a suppressed internal state [1]. By using this *stubbed transitions* structure, the transition method can be extended from non-hierarchical into hierarchical CPN.

The second area is to try other approaches of UML. There are 9 kinds of UML diagrams, but only 3 are used in this thesis. In the future, other UML diagrams can be used to solve the concurrent problems.

Finally, the computer tools to implement this transition method can be used. A software package to implement this transition can be written, and add this new package into the existing UML tools, so that the UML diagrams can be automatically transformed to the CPN diagrams.

Bibliography

[1] *OMG Unified Modeling Language Specification.* The Object Management Group, Inc.(OMG), September 2001,Version 1.4.

[2] *Rational Rose Help.* Rational Software Corporation, 2002.05.

[3] L. Baresi and M. Pezze, "On formalizing UML with high-level Petri Nets," in *Lecture Notes In Computer Science: Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets*, vol. 201, pp. 276–304, Springer-Verlag, 2001.

[4] L. Baresi, "Some Preliminary Hints on Formalizing UML with Object Petri Nets," *Integrated Design and Process Technology*, vol. IDPT-2002, June 2002.

[5] B. E. Bauskar and B. Mikolajczak, "Abstract Node Method for Integration of Object Oriented Design with Colored Petri Nets," Technical Report, CIS-4-2004, Computer and Information Science Department, University of Massachusetts at Dartmouth, MA, 2004.

[6] M. Beeck, "Formalization of UML-Statecharts," vol. UML2001, pp. 406–421, Springer-Verlag, 2001.

[7] D. Bjorklund, J. Lilius, and I. Porres, "Rialto profile in the SMW toolkit," in *Proceedings of the Third International Conference on Application of Concurrency to System Design - ACSD 2003*, 18-20 June 2003.

[8] C.A.R.Hoare, *Communicating Sequential Processes.* Prentice Hall, 1985.

[9] A. Evans, R. B. France, K. Lano, and B. Rumpe, "The UML as a Formal Modeling Notation," in *Lecture Notes In Computer Science: the First International Workshop on The Unified Modeling Language <<UML>>'98: Beyond the Notation*, pp. 336 – 348, Springer-Verlag London, UK, 1998.

[10] R. B. France, J.-M. Bruel, M. M. Larrondo-Petrie, and M. Shroff, "Exploring The Semantics of UML Type Structures with Z," in *Proceeding of the IFIP TC6 WG6.1 international workshop on Formal methods for open object-based distributed systems*, (Canterbury, United Kingdom), pp. 247–257, Chapman & Hall, Ltd. London, UK, UK, July 1997.

[11] T. Gooch, "History of UML." http://pigseye.kennesaw.edu/ dbraun/csis4650/A&D/UML_tutorial/history/uml.htm, Spring 2000.

[12] D. Harel, "Statecharts: A visual formalism for complex systems," in *Science of Computer Programming*, vol. 8, pp. 231–274, 1987.

[13] D. Harrel and A. Naamad, "The statemate semantics of statecharts," in *ACM Transactions on Software Engineering*, vol. 5(4), pp. 293–333, 1996.

[14] H. Hussmann, M. Cerilli, G. Reggio, and F. Tort, "Abstract Data Types and UML Models," 1999.

[15] R. Janicki and P. Lauer, *Specification and analysis of concurrent systems, The COSY approach.* Springer-Verlag, 1992.

[16] R. Janicki, "Concurrency Theory." Course Notes, 2002.

[17] K. Jensen, "Coloured Petri Nets and the invariant-method." Technical Report, DAIMI PB-104, October 1979.

[18] K.Jensen, *Coloured Petri Nets*, vol. I. Springer-Verlag, 1996.

[19] T. Kowalski, "UML Activity Diagram Verification By Coloured Petri Nets," Technical Report, TR-5-2002, Computer Science Department, Wroclaw University of Technology, Poland, 2002.

[20] J. Lilius and I. Paltor, "A tool for verifying UML models," in *Proceeding ASE'99*, vol. UML, pp. 255–258, IEEE Computer Society, 1999.

[21] M. Page-Jones, *Fundamentals of Object-Oriented Design in UML.* Addison-Wesley, 1999.

[22] J. Paul R. Reed, *Developing applications with Java and UML.* Addison-Wesley, 2002.

[23] W. Reisig, *Petri Nets.* Springer-Verlag, 1982.

[24] R.Milner, *Communication and Concurrency.* Prentice Hall, 1989.

[25] J. A. Saldhana and sol M.Shatz, "UML Diagram to Object Petri Net Models: An Approach for Modeling and Analysis," in *Proc. of Software Engineering and Knowledge Engineering (SEKE)*, (Chicago), pp. 103–110, July 2000.

[26] W.Reisig, *Elements of Distributed Algorithms.* Springer-Verlag, 1998.