Modelling Concurrent Systems with Object-Oriented Coloured Petri nets

Modelling Concurrent Systems with Object-Oriented Coloured Petri nets

By

Angela Wu

A thesis

submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements

for the Dregree
Master of Science

McMaster University
©Copyright by Angela Wu

MASTER OF SCIENCE (2003) McMaster university

COMPUTING & SOFTWARE Hamilton Ontario

TITLE: Modelling Concurrent Systems with Object-Oriented Coloured Petri nets

AUTHOR: Angela Wu

SUPERVISOR: Dr. Ryszard Janicki

NUMBER OF PAGES: xi, 105

Abstract

This thesis presents a new modelling technique for the complex current system. It integrates object-oriented methodology into Petri Nets formalism.

Petri Nets are used for modelling concurrent systems. They have natural graphical representation as well as formal specifications. They have been successfully used in various industrial applications. But with the development of distributed and network systems, their traditional weakness, namely their inadequate support for compositionality, is a big obstacle to their practical use for large, complex systems. To address this problem, we introduce the Object-Oriented Coloured Petri Nets (OO-CPN), which integrates the powerful modularity of an object-oriented paradigm into Petri Nets formalism. OO-CPN is based on Coloured Petri Nets and supports the concepts of object, class, inheritance and polymorphism.

Acknowledgments

I would like to express my sincere thanks to Dr. Ryszard Janicki, my supervisor, for his invaluable and patient guidance, critical insight, and constant inspiration and support.

I am grateful to Dr. Ridha Khedri and Dr.Michael Soltys for their careful review of this thesis and their many valuable comments.

I would like to express my appreciation to Lin He, who proposed the basic idea behind OO-CPN.

Finally, I would like to acknowledge the financial support of the Ministry of Training, Colleges and Universities of Canada.

Contents

List of Figures

Chapter 1

Introduction

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

1.1 Background

Petri Nets were introduced by C. A. Petri in the early 1960s [C.A62] as mathematical tools for modelling distributed systems that incorporate, in particular, notions of concurrency, non-determinism, communication and synchronization. Now, Petri Nets have been successfully used for concurrent and parallel systems modelling and analysis, communication protocols, performance evaluation and fault-tolerant systems.

There are various kinds of Petri Nets and computer tools for using them, which differ quite a lot in their expressive power, legibility of models and analytical capa-

bilities.

The *Place/Transition Nets* (PT-nets) [Jen97], are low-level Petri Nets. They possess the appealing features of the Petri Nets, such as intuitive understanding, graphical representation, simplicity and formality. However, they are not adequate to describe a large system, since large systems often contain many parts that are similar, but not identical. When using PT-nets, these parts must be represented by disjointed subnets with nearly identical structures. Thus the total PT-net becomes very large and it becomes difficult to see the similarities and differences between the individual subnets representing similar parts. This phenomenon is known as the *State Explosion* problem.

Coloured Petri Nets (CPNs) [Jen97] extend PT-nets by adding colours (*i.e.* data elements) to tokens and using expressions to work with them. The introduction of colours significantly reduces the sizes of models, thus increasing their legibility. Nevertheless, this is still not enough and we still need better structuring techniques to describe the system in a more compact form.

Hierarchical Coloured Petri Nets (HCPNs) [Jen97] use the techniques of substitution of transitions and fusions to construct a large description from smaller units that can be investigated more or less independently of one another. HCPNs make it possible to relate a number of individual CPNs to each other in a formal way.

Object-Oriented Petri Nets try to achieve a complete integration of object orien-

tation into Petri Net formalism. Petri Nets have a natural graphical representation, which aids in the understanding of the formal specifications, together with a range of automated and semi-automated analysis techniques. Object-oriented technology provides powerful structuring facilities that stress encapsulation and promote software reuse. This addresses a traditional weakness of Petri Net formalisms, namely the inadequate support for compositionality. OOPNs combine these two paradigms to support both features.

OOPNs are one of the most sophisticated Petri Nets, since combination of object orientation and Petri Net formalism requires complex design and may lead to some problems. In recent years, a number of proposals have been made in this field, focusing on different aspects of OOPNs and solving some of their problems. But these methods still have their weak points. Below I will briefly review them.

Object Petri Nets (OPNs) [Lak01] support an integration of object-oriented concepts into Petri Nets, including inheritance and the associated polymorphism and dynamic binding. They have a single class hierarchy that includes both token types and subnet types, thereby allowing multiple levels of activity in the net. OPNs support synchronization constraints using method guards, but method guards are deficient for history-sensitive synchronization, as indicated by Satoshi Matsuoka and Akinori Yonezawa [MY93].

$OB(PN)^2$ [Lil01] is an object-based Petri Net programming notation. It lays the

foundations for the development of automatic verification methods for concurrent programs written in object-oriented languages, and it can be seen as a set of rules for the translation of object-oriented specifications written in an object-oriented specification formalism into Petri-Nets formalism. The translation relies on the CCS-like composition operators defined for M-nets [BFH+98]. Each program construct is translated into a box (a special kind of net) or an operation for combining boxes [EM01].

The current version of $OB(PN)^2$ does not provide synchronization constraints on the methods of the object, nor does it provide inheritance, because the combination of these two features may create the problem of "inheritance anomaly" which I will specify in the following chapters.

Concurrent Object-Oriented Petri Nets (CO-OPN/2) [BD01] was devised for the specification of large concurrent systems. The new version, called CO-OPN/2, is based on two underlying formalism approaches: order-sorted algebra, and algebraic net. In CO-OPN/2, classes are object templates described by a special kind of algebraic net, while objects are net instances of these classes. Interaction between objects is realized by using "synchronization expressions", which are more general than the classic transition fusion of Petri Nets. At the semantics level, transition systems are used to express the true concurrency of the object behaviours.

CO-OPN/2 provides an **Inherit** section followed by the class modules that the current class inherits as the mechanism of inheritance. In this section three fields can

be specified:

- The **Rename** field allows us to rename some inherited identifiers;
- The **Redefine** field groups the components whose the properties are redefined in the inheriting class;
- The **Undefine** field groups the components that have to be eliminated in the inheriting class.

This mechanism requires total knowledge of and access to the ancestor classes. In this way, the encapsulation of class implementation is broken with respect to synchronization constraints.

Class Orientation With Nets (CLOWN) [BCC01] is an object-oriented concurrent specification language. The main building block of a CLOWN specification is the *elementary class.* The semantics of a class are given by a corresponding OBJSA elementary component. Every object, instance of a class, is represented by a structured token flowing in the associated OBJSA elementary component. The communication between objects is managed by the mutually synchronous execution of corresponding methods. Each object may impose some synchronization constraints, which are specified in the class interface, to guarantee that object synchronization can operate correctly.

The inheritance in CLOWN is that each class can extend the parents' specifica-

tions and specialize in a restricted domain. CLOWN separates the sequential code from the synchronization code in order to prevent the harmful effect of the inheritance anomaly.

Object Coloured Petri Nets (OCP-nets) [CM01] are an extension of Coloured Petri Nets (CPN). OCP-nets are divided into two parts: the static structure and the dynamic structure. The static structure is made up of a set of *class nets*. Each class net offers one or more services (methods). The dynamic structure consists of *object nets*, which are instances of class nets. Object nets communicate through the exchange of tokens. This can be done asynchronously via communication fusion places or synchronously through a modified version of synchronous channels.

Lin He [He01] proposed *Object-Oriented Coloured Petri Nets* (OO-CPN). OO-CPN is based on Hierarchical Coloured Petri Nets and supports the concepts of object, class and inheritance. In OO-CPN, objects communicate through communication channels and concurrency issues are declaratively abstracted by temporal constraints among events.

1.2 Purpose

Lin He proposed several good ideas in the OO-CPN design. OO-CPN not only provides an easy-to-use graphical description tool as well as formal semantics to model concurrency, but also introduces object-oriented concepts into OO-CPN to support

software reuse and maintenance. OO-CPN integrates Coloured Petri Nets and object-oriented methodology in a natural way that the user can grasp quickly. Lin He's work is a good start, and can be extended in the following aspects.

The class declarations in the OO-CPN are clear but too simple to cover complete aspect of the classes. In addition, OO-CPN does not support *polymorphism*, which is an important characteristic of object orientation to build the hierarchical structure. Therefore, OO-CPN can be improved to fully support object orientation.

Hierarchical structure is a good way to solve the *State Explosion* problem. It includes two techniques: substitution of transitions and fusion. But object-oriented mechanisms can describe the system in a more compact form, and they are easy to understand. We can use the object-oriented approach to substitute the hierarchical constructs.

In Lin He's thesis [He01], a method for solving the inheritance anomaly problem is proposed. But this method cannot solve certain kinds of anomalies, such as history-only sensitiveness. In addition, Lin He did not describe how this problem affects Petri Nets.

This thesis proposes an improved OO-CPN. The new OO-CPN will be based on the goals of "simplicity, object orientation, and familiarity".

Simplicity The improved OO-CPN combines the object-oriented concepts with Coloured Petri Nets methodology without adding complex structures and algorithms.

This makes it easy to understand and grasp.

Object Orientation The improved OO-CPN fully supports object-oriented concepts.

Familiarity The improved OO-CPN is based on Coloured Petri Nets. The object-oriented features are added to CPN without changing the main frame of the CPN.

In conclusion, the improved OO-CPN can provide more compact system design, as well as support software reuse and maintenance.

1.3 Outline

Chapter 2 serves as a survey of the main concepts of object orientation. Also, it introduces object-oriented concurrent programming and its problems.

Chapter 3 introduces Coloured Petri Nets, including notation, graphic representation and formal definitions.

Chapter 4 provides an informal introduction to OO-CPN. It describes the main construction of OO-CPN and proposes a new solution to the inheritance anomaly problem. It concludes by modelling a Reader and Writer system using all of the methodologies introduced previoursly.

Chapter 5 formally definitions OO-CPN and gives an example of the Reader and Writer system.

Chapter 6 uses an example of Distributed Program Execution to illustrate the characteristics of OO-CPN.

Chapter 7 makes a conclusion, discusses the contribution of this thesis, and suggests future work in this area.

Chapters 2 and 3 discuss the existing literature, while chapters 4-7 are the author's contributions to the topic.

Chapter 2

Object Orientation and Concurrency

The main purpose of this thesis is to apply object-oriented methodology to concurrent system design. This chapter will briefly describe these two paradigms.

Section 2.1 introduces the concepts of object orientation. Section 2.2 briefly describes the concurrent system. Section 2.3 discusses the problems that exist in object-oriented concurrent programming.

2.1 Object Orientation

Object-oriented programming (OOP) divides a problem into its constituent parts - objects. Each component becomes a self-contained object that contains its own

instructions and data that relate to that object. In this way, complexity is reduced and the programmer can manage large programs.

All OOP languages share three defining traits: encapsulation, inheritance and polymorphism [Sch98].

Encapsulation Encapsulation is the mechanism that binds together code and the data it manipulates, keeping both safe from outside interference and misuse. Here, an object serves as a self-contained "black box" that links code and data together. Within an object, code, data, or both may be private to that object or public. Private code or data cannot be accessed by any part of the program that exists outside of the object. When code or data is public, other parts of program can access it even though it is defined within an object. For all intents and purposes, an object is a variable of a user-defined type, class, which is an Abstract Data Type. Abstract Data Type is a data type whose representation is hidden in the implementation. It can decompose large programs into smaller pieces, provide a way of substituting alternate solutions, and separate compilation. In conclusion, it is a convenient way to organize large programs.

Inheritance Inheritance is the primary feature of object-oriented technology. It is the process by which one object acquires the properties of another. More specifically, an object can not only inherit a general set of properties, but also it add those features that are specific only to itself. Inheritance is important in

object-oriented design because it allows an object to support *hierarchical classification.* Most information is made manageable by hierarchical classification. For example, think about the description of a car. A car is part of the general class called vehicle, which is part of the even more general class of transportation tools. In each case, the child class inherits all those qualities associated with the parent and adds to them its own defining characteristics. Through inheritance, it is possible to describe an object by stating what general class (or classes) it belongs to, along with those specific traits that make it unique.

Polymorphism Polymorphism (from the Greek, meaning "many forms") is the ability to take on assign a different meanings or usages in different contexts - specifically, to allow an entity to have more than one form. For example, given a base class shape, polymorphism enables the derived classes, such as circles, rectangles and triangles, to have different implementation of circumference methods. No matter what shape an object is, applying the circumference method to it will return the correct results. Polymorphism models something quite important about the real world: different things behave differently.

2.2 Concurrent System

A concurrent system has multiple components of control, allowing it to perform multiple tasks in parallel and to control multiple activities that occur at the same time.

Concurrent programs have characteristics that distinguish them from sequential programs.

- When more than one activity can occur at one time, program execution is usually nondeterministic.

- Code may execute in surprising orders: any order that is not explicitly ruled out is allowed, *e.g.* a field set to one value in one line of code may have a different value before the next line of code is executed.

Therefore, concurrent programming requires more rigour on the part of the programmer.

There are two main ways to implement concurrency:

Shared Memory The concurrent processes will access a shared memory - the processes communicate by reading and writing in shared memory locations.

Message Passing The concurrent processes communicate by message passing.

2.3 Object-Oriented Concurrent Programming

Object-oriented mechanisms, such as classes and inheritance, and concurrency mechanisms, such as threads and locks, provide two separate software structuring dimensions. The development of an object-oriented concurrent software model integrates object-oriented features with concurrent execution and synchronization, eliminating the need to consider two separate dimensions when developing object-oriented concurrent software. When we try to integrate these two paradigms, we need to consider the following issues:

Object Coordination In a concurrent object-oriented program, the coordination between cooperating objects should be carefully considered. The main motivation behind the work on object coordination is to allow coordination patterns among several objects to be specified separately from the implementation of individual objects. The benefit of such an approach is that it is possible to coordinate objects in ways that were not anticipated when the objects were implemented, and that it allows the reuse of the coordination patterns themselves.

Dynamic Reconfigurability The programming model must deal with the creation of new objects in the evolution of the system. In particular, to accommodate the creation of new objects, there must be a mechanism for communicating the existence of such new objects to the already existing ones.

Inherent Concurrency In a concurrent program, objects are shared by concurrent threads. The execution of their methods needs to be synchronized in order to provide mutually exclusive access to the objects' states as well as to coordinate the use of an object by concurrent threads. When a synchronization scheme is inherited, there will be substantial rewriting of the inherited code in the subclass to accommodate the new synchronization constraints. This problem is called the inheritance anomaly.

Conceptually, the combination of object-oriented methodology with concurrent system design is attractive, but is not easy to make these two techniques work together effectively. I will try to explain the detail in the following chapters.

Chapter 3

Coloured Petri Nets

The main frame of OO-CPN is based on Coloured Petri Nets (CP-nets). This chapter introduces the notations and definitions of CP-nets. This chapter is based on [Jen97].

Section 3.1 serves as an informal introduction to Colored Petri Nets. Section 3.2 gives the formal definition of CP-nets and their behaviour.

3.1 Informal Introduction to Coloured Petri Nets

Coloured Petri Nets (CP-nets) were developed in the late 1970s. CP-nets use tokens to represent object (*e.g.* resources, goods, humans) in the modelled system. Each token has a value often referred to as a 'colour'. The colour represents the attributes of the object. Transitions use the values of the consumed tokens to determine the values of the produced tokens.

In order to make explanation easy to understand, this section introduces Coloured Petri Nets by means of an example.

Assume there are a set of processes, which share a common pool of resources. The system is comprised of two p-processes, three q-processes, one r-resource, three s-resources and two t-resources. Each process is cyclic, and during the individual parts of its cycle, the process needs to have exclusive access to a varying amount of the resources. The demands of the processes are specified in Figure 3.1 and Figure 3.2.

Figure 3.1 specifies the demands of the processes by describing the possible **states**. P-processes have four different states. In the upper state, no resources are needed. In the second state, two s-resources are needed, and so on. Similarly, q-processes have five different states, and for each of these states the required amount of resources is specified.

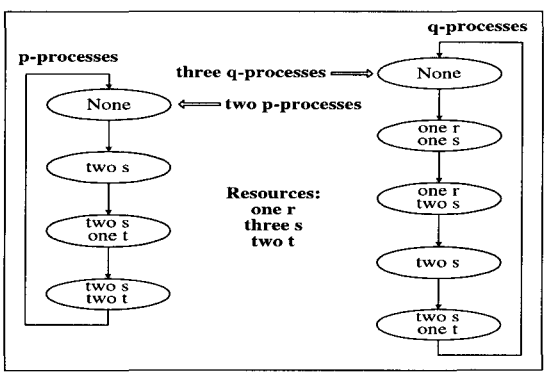

Figure 3.1: The states of the processes in the resource allocation system

Figure 3.2 shows the demands of the processes by describing the possible **actions**. The two white arrows indicate that all processes start by executing the two actions at the top of the diagram.

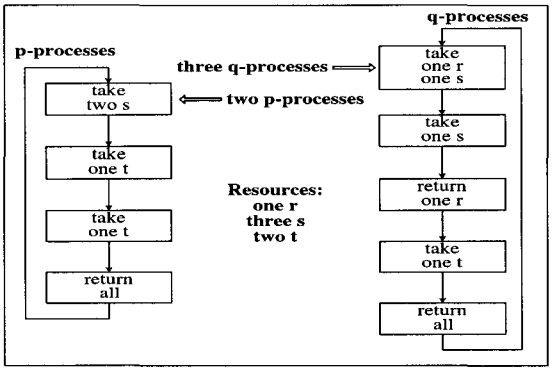

Figure 3.2: The actions of the processes in the resource allocation system

Figure 3.3 gives a Petri Net specification of the resource allocation system.

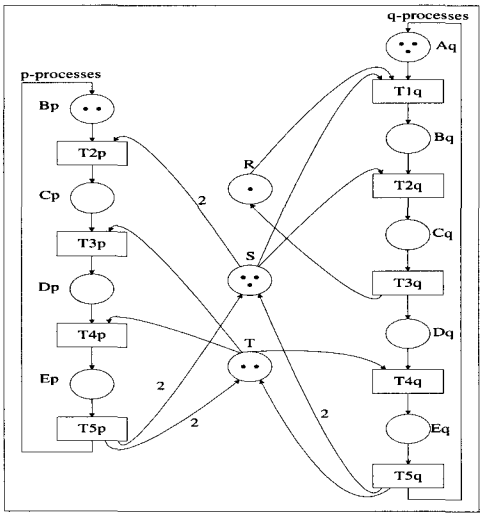

Figure 3.3: The PT-net describing the resource allocation system (initial marking M0)

A Petri Net (PT-net) includes descriptions for both states and actions. In the above diagram, the states of the resource allocation system are indicated by means of ellipses, which are called **places**. Each place may contain a dynamically varying number of small black dots, which are called **tokens**. An arbitrary distribution of tokens in the places is called a **marking**. The actions of the resource allocation system are indicated by means of rectangles, which are called **transitions**. The places and transitions are collectively referred to as the **nodes**. The PT-net also contains a set of arrows, which are called **arcs**. Each arc connects a place with a transition or transition with a place-but never two nodes of the same kind. Each arc may have an expression attached to it. This called an arc expression. A node x is called an **input node** of another node y, if and only if there exists a directed arc from x to y; y is called the **output node**. We shall also talk about **input places**, **output places**, **input transitions**, **output transitions**, **input arcs** and **output arcs**.

Above is the description of the syntax of the PT-net; now let us consider the behaviour. A PT-net can be considered as a game board where the tokens are markers. A move is possible if and only if each place contains at least the number of tokens prescribed by the arc expression of the corresponding input arc. We say that such a transition is **enabled**. When the move take place, we say that the transition **occurs**. Figure 3.4 describes the transition T2p transforming M0 into the marking M1. Figure 3.5 describes the transition T1q transforming M0 into the marking M2.

From the diagrams we can see that both markings M1 and M2 are directly reachable from M0. Thus we say that T2p and T1q are **concurrently enabled** in M0. This means that two transitions may occur in parallel. We also say that the step S1={T2p, T1q} is enabled in M0.

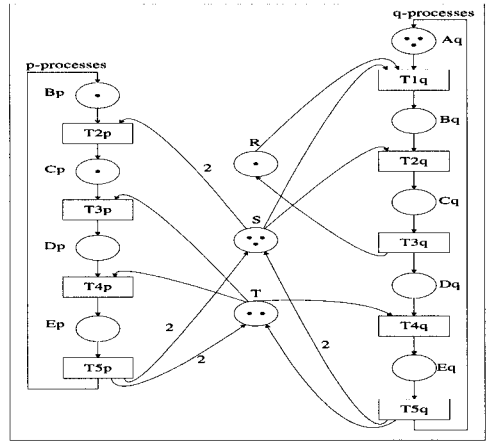

Figure 3.4: The marking M1 (reachable from M0 by T1q)

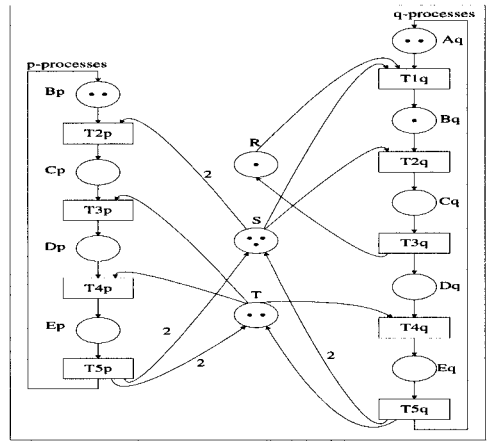

Figure 3.5: The marking M2 (reachable from M0 by T2p)

Above is the PT-net description of the resource allocation system. It represents

the two kinds of processes by two separated subnets, even though the processes use the resources in a similar way. The following is the Coloured Petri Net (CP-net) description of the system.

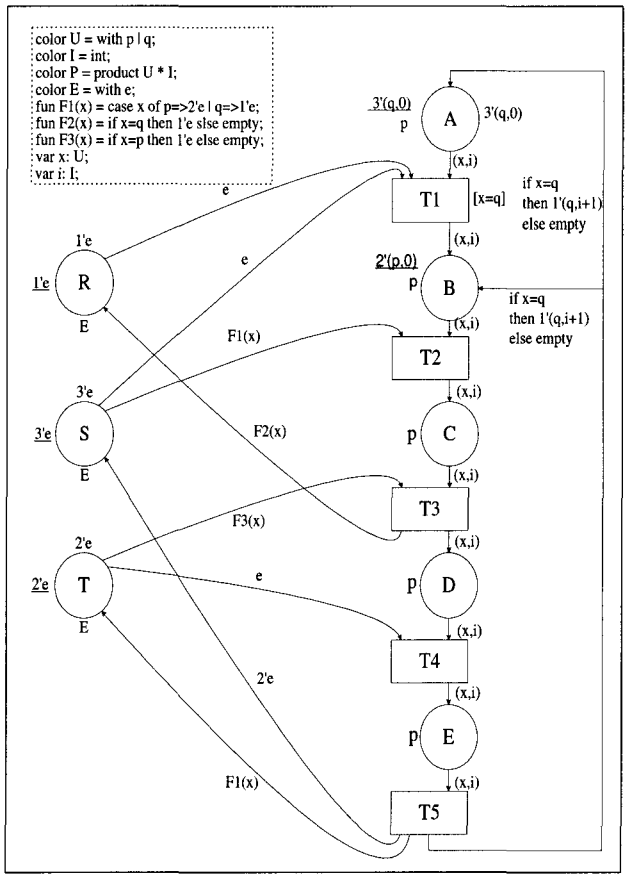

Figure 3.6: The CP-net describing the resource allocation system (initial marking M0)

CP-net attaches a colour to each token and a colour set to each place, allows us to use fewer places than would be needed in a PT-net. Figure 3.6 shows the initial state of the system. There are three (q,0)-tokens in A and two (p,0)-tokens in B,

while C, D and E have no tokens. Moreover, R has one e-token, S has three e-tokens, and T has two e-tokens. The marking of each place is a multi-set over the colour set attached to that place.

Now, let us consider the action of the CP-net. The transition T2 has two variables (x and i). Before we can consider an occurrence of the transition these variables have to be bound to colours of corresponding types. From Figure 3.7 we get the **binding** b = $\langle x = p, i = 0 \rangle$. Since the two input arc expressions evaluate to (p,0) and 2'e, binding b is enabled, because each of the input places contains at least the tokens to which the corresponding arc expression evaluates (one (p,0)-token on B and two e-tokens on S). When a transition is enabled (for a certain binding) it then removes tokens from its input places and adds tokens to its output places. A pair (t,b), where t is a transition and b a binding for t, is called a binding element. The binding element (T2,b) is enabled in the initial marking M0 and it transforms M0 into the marking M1, shown in Figure 3.7. Similarly, the binding element ($T1$, $\langle x = q, i = 0 \rangle$) is enabled in M0 and it transforms M0 into the marking M2, shown in Figure 3.8. In addition, the transition T1 has a **guard**: x=q. The guard is a boolean expression and it may have variables in exactly the same way that the arc expression has. The purpose of the guard is to define an additional constraint that must be fulfilled before the transition is enabled. In this case the guard tell us that it is only tokens representing q-processes that can move from A to B.

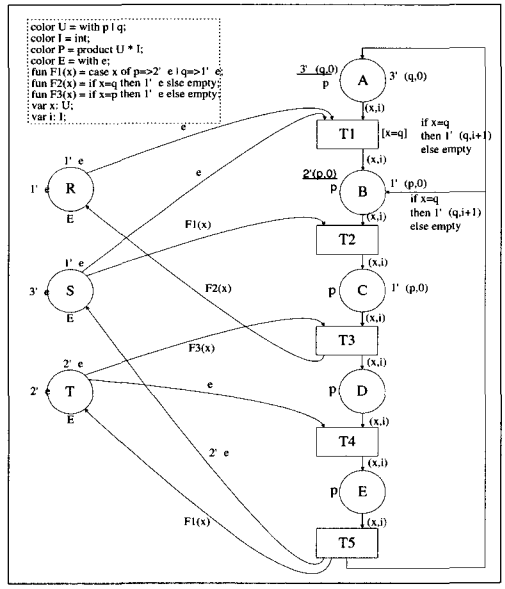

Figure 3.7: The marking M1 (reachable from M0 by (T2,(x=p,i=0)))

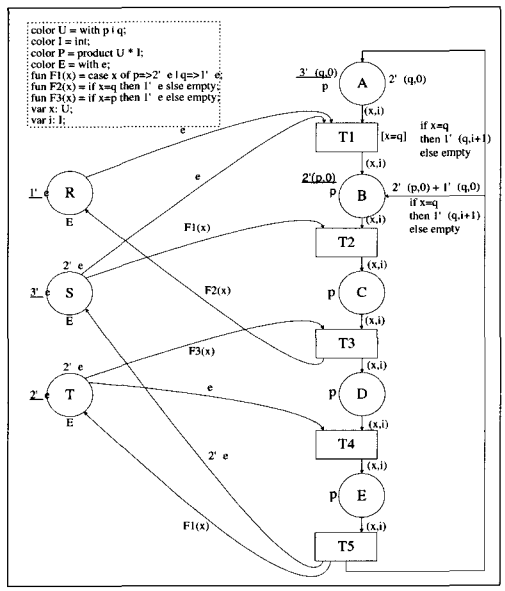

Figure 3.8: The marking M2 (reachable from M0 by (T1,(x=q,i=0)))

Two transitions can also be concurrently enabled in a CP-net. For example, the marking M0 can have an enabled step, which looks like follows: $S1 = 1'(T1, \langle x = q, i = 0\rangle) + 1'(T2, \langle x = p, i = 0\rangle)$. This means that occurrence of S1 moves a (q,0)-token from A to B, moves a (p,0)-token from B to C, and removes an e-token from R and three e-tokens from S. The effect of this step is the sum of the effects of the individual binding elements.

From the above example, we can see that Coloured Petri Net representation is more compact than ordinary Petri Net representation.

3.2 Formal Definition of the Coloured Petri Nets

This section gives the formal definition of the Coloured Petri Nets. Before giving the abstract definition, followings are assumed to be well-defined:

- The **set of all elements** in type **T** is denoted by the type name **T** itself.
- The **type of a variable v**, denoted by Type(v).
- The **type of an expression** expr, denoted by Type(expr).
- The **set of variables in an expression** expr, denoted by Var(expr).
- A **binding of a set of variables** V, associating with each variable $b(v) \in Type(v)$.

- The **value obtained by evaluating an expression expr in a binding b**, is denoted by $expr\langle b\rangle$. Var(expr) must 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.

- A **closed** expression is an expression without variables. It can be evaluated in all bindings, and all evaluation give the same value which we denote as "expr".

- **B** was used to denote boolean type which containing the elements {false, true}.

- When Vars is a set of variables, we use Type(Vars) to denote the set of types $\{Type(v)|v \in Vars\}$.

Definition 3.1. 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: $P \cap T = P \cap A = T \cap A = \emptyset$

5. N is a **node** function. It is defined from A into $P \times T \cup T \times P$

6. C is a **colour** function. It is defined from P into Σ.

7. G is a **guard** function. It is defined from T into expressions such that: $\forall t \in T : [Type(G(t)) = B \wedge Type(Var(G(t))) \subseteq \Sigma]$

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

Use above definition, the CP-net can be defined as following:

- $\Sigma = \{U, I, P, E\}$.
- $P = \{A, B, C, D, R, S, T\}$.
- $T = \{T1, T2, T3, T4, T5\}$.
- $A = \{AtoT1, T1toB, BtoT2, T2toC, CtoT3, T3toD, DtoT4, T4toE, EtoT5, \}$ $\{T5toA, T5toB, RtoT1, StoT1, StoT2, TtoT3, TtoT4, T3toR, T5toS, T5toT\}$.
- $N(a)$ = (SOURCE,DEST) if a is in the form SOURCEtoDEST.
- $C(p) = \begin{cases} P, & if P \in \{A, B, C, D, E\} \ ; \\ E, & \text{otherwise.} \end{cases}$
- $G(t) = \begin{cases} x = q, & \text{if t=T1;} \\ true, & \text{otherwise.} \end{cases}$

- $E(a) = \begin{cases} e, & if a \in \{RtoT1, StoT1, TtoT4\}; \\ 2'e, & \text{if a=T5toS;} \\ casexofp \rightarrow 2'e | q \rightarrow 1'e, & if a \in \{StoT2, T5toT\}; \\ ifx = qthen1'eelseempty, & \text{if a=T3toR;} \\ ifx = pthen1'eelseempty, & \text{if a=TtoT3;} \\ ifx = qthen1'(q, i+1)elseempty, & \text{if a=T5toA;} \\ ifx = pthen1'(p, i+1)elseempty, & \text{if a=T5toB;} \\ (x, i), & \text{otherwise.} \end{cases}$

- $I(p) = \begin{cases} 3'(q, 0), & \text{if p=A;} \\ 2'(p, 0), & \text{if p=B;} \\ 1'e, & \text{if P=R;} \\ 3'e, & \text{if P=S;} \\ 2'e, & \text{if P=T;} \\ \emptyset, & \text{otherwise.} \end{cases}$

Having defined the structure of the CP-nets we are ready to define their behavior. First, we shall introduce some notations in the following:

- $\forall t \in T : Var(t) = \{v | v \in Var(G(t)) \vee \exists a \in A(t) : v \in Var(E(a))\}$ Var(t) is called the set of variables of t. All elements in this set are either variables in guard function of t or variables in arc expression function of t.

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

  $E(x_1, x_2)$ is called the expression of (x_1, x_2). The expression of node function is the addition of arc expressions of the source node and destination node.

Definition 3.2. A **binding** of a transition t is a function b defined on Var(t), such that:

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

2. $G(t)\langle b \rangle$.

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

Note: 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 correct type. $G(t)\langle b \rangle$ denotes the evaluation of the guard expression $G(t)$ in the binding b.

Definition 3.3. 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 *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 markings and steps are denoted by $\mathbf{M}$ and $\mathbf{Y}$, respectively.

Note: Usually, we represent markings as functions defined on P such that: $\forall p \in P, \forall c \in C(p) : M(p) \in C(p)_{MS}$

Definition 3.4. A step Y is **enabled** in a marking M iff 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 $(t_1, b_1), (t_1, b_1) \in Y$ and $(t_2, b_2) \neq (t_1, b_1)$ 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.

Note: The expression evaluation $E(p, t)\langle b \rangle$ yields the multi-set of token colours, which are removed from p when t occurs with the binding b. By taking the sum over all binding elements $(t, b) \in Y$ we get all the tokens that are removed from p when Y occurs. this multi-set is required to be less than or equal to the marking of P. It means that each binding element $(t, b) \in Y$ must be able to get the tokens specified by $E(p, t) < b >$, without having to share these tokens with each other binding elements of Y.

Definition 3.5. When a step Y is enabled in a marking M_1 it may occur, changing the marking M_1 to another marking M_2, 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.

Note:When a step Y is enabled it may occur 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 evaluated for the occurring bindings.

Chapter 4

Informal Introduction to OO-CPN

This chapter contains an informal introduction to OO-CPN.

The chapter is organized into the following parts: Section 4.1 introduces the approach used to integrate properties of CP-nets with the concepts of object orientation. Section 4.2 illustrates the structure of the class diagram. This is the main building block of the OO-CPN. Section 4.3 explains how inheritance is used in the OO-CPN, then discusses the problem of the inheritance anomaly in the OO-CPN, finally proposes a new method to solve the problem of the inheritance anomaly. Section 4.4 defines polymorphism in the OO-CPN. Section 4.5 introduces the communication channel in the OO-CPN. Section 4.6 contains an example of an OO-CPN.

4.1 Objects and Petri Nets

There are two approaches to integrating the concepts of the Object Orientation and the Petri Net: one is to place objects inside Petri Net, the other is to place Petri Net inside objects. The first way treats the tokens as objects, while the second one uses the nets to model the inner behaviour of the objects [Bas95].

This thesis uses the first approach to model the control structure of the system. Objects are treated as tokens. Transitions will call methods on objects. A transition moves objects from one place to another, and objects can be dynamically created and destroyed during the life of the system.

The second approach is used to model the inner behaviour of objects. Every class has a class net, which is a small OO-CPN. This small OO-CPN describes the behaviour of instances of the class.

In this way, the whole system is divided into several objects. Each object is relatively independent. There may be the several objects activate concurrently, which can communicate by communication channels. Object can contain objects. This feature is used to construct the system instead of hierarchical structure.

4.2 Class Diagram

The main building block of OO-CPN is the object. An object is the instance of a class. This mechanism provides the most important feature of object-oriented design: encapsulation. A class templates is composed of two main parts: a set of textual expressions and a class net. (This idea was borrowed from CLOWN [BCC01]. The structure of the main frame is similar to that of CLOWN, but the content is different.) The following is a detailed explanation of the structures of these two parts.

4.2.1 Textual Expressions

Textual expressions provide the textual description for the class.

Class: The class identifier.

Inherits: The inheritance clause describes the relation between the class and all its parents.

Colour: Abstract data type, which is defined from the elementary data type and user-defined data type.

Const: These are typed entities. Their values are fixed at instance creation and never change afterwards.

Var: These are typed entities. Their values can be modified during transactions.

State: Specifies the states an object has during its lifetime.

Transition: Specifies transitions an object goes through during its lifetime. In order to prevent the inheritance anomaly, the constraints are separated from the transition. In this way, the subclass may only need to modify the part of the constraint and does not touch the internal structure of the action. This part of the constraint includes:

Pre: Specifies the precondition.

Post: Specifies the postcondition.

There are three kinds of specifiers for const, var, state and transitions:

Public: Public member can be accessed by everybody in this system.

Private: Private member can only be accessed in this class.

Default: If there is no specifier provided, we treat this member as default. It can be accessed in this class and in its child classes.

4.2.2 Class Net

Class Net models the inner behaviour of objects. The current marking of the net models the inner state of an object, and transitions in the net may be used to model the execution of a method by this object.

In this approach, the whole system is divided into several objects. These objects can communicate through communication channels. Objects are instances of classes, and the relationship between classes is the inheritance.

4.3 Inheritance

Inheritance plays a very important role in object-oriented technology. It can not only provide component reuse, but also manage the system in a hierarchical structure.

4.3.1 Inheritance Structural

There are two terms commonly used when discussing inheritance.

Base class: When one class is inherited by another, the class that is inherited is called the *base class.*

Derived class: When one class is inherited by another, the inheriting class is called the *derived class.*

There are three kinds of way for subclass inheriting from superclass: **public**, **private** and **default**. In OO-CPN, to make things simple, only public inheritance is supported. Therefore, in the derived classes, both public and default members can be inherited, but the private members cannot be inherited. Also, in public inheritance there is no change to the specifiers of the inherited members of the derived classes.

When one class inherits another, it uses this general form:

class derived-class-name: base-class-name { ... }

There two kinds of inheritance: single inheritance, and multiple inheritance. OO-CPN only support the single inheritance. The reasons for omitting multiple inheritance stem mostly from the"simplicity, object orientation, and familiarity" goals that were stated in Chapter1. As a simple modelling tool, OO-CPN should be as similar to Coloured Petri Nets as possible (familiarity) without carrying over unnecessary complexity (simplicity).

In most designers' opinions, multiple inheritance causes more problems and confusion than it solves. One justification of this opinion is a traditional multiple inheritance problem: the diamond problem. The diamond problem is an ambiguity that can occur when a class multiply inherits from two classes that both descend from a common superclass. For example, in Michael Crichton's novel *Jurassic Park*, scientists combine dinosaur DNA with DNA from modern frogs to get an animal that resembled a dinosaur but in some ways acted like a frog.

This *Jurassic Park* scenario could be represented by the following inheritance hierarchy in Figure 4.1

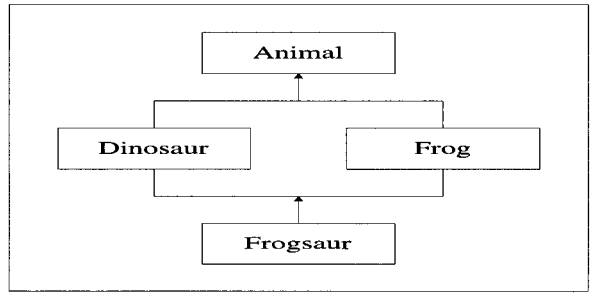

Figure 4.1: Multiple inheritance in Jurassic Park

The diamond problem can arise in inheritance hierarchies like the one shown in Figure 4.1. In fact, the diamond problem gets its name from the diamond shape of such an inheritance hierarchy. One way the diamond problem can arise in the Jurassic Park hierarchy is if both Dinosaur and Frog, but not Frogosaur, override the behaviour **talk**, declared in Animal. The frog says: "Ribbit, ribbit.". The dinosaur says: "Oh, I'm a dinosaur and I'm OK... .:. The diamond problem would arise if we look at the behaviour **talk** of the Frogosaur. Will a Frogosaur croak"Ribbit, Ribbit" or sing "Oh, I'm a dinosaur and I'm okay... "?

Therefore, OO-CPN adopts single inheritance to prevent the ambiguities caused by the diamond problem.

4.3.2 Inheritance Anomaly

Inheritance plays an important role in *sequential object-oriented* programming. It provides the functionality of code reuse. But this feature seems to generate problems

in *concurrent object-oriented* design, since the combination of inheritance and concurrency sometimes leads to heavy breakage of encapsulation. For example, consider a first-in, first-out bounded buffer as illustrated in Figure 4.2. It has two public methods, *get()* and *put()*, where *get()* removes an item from the buffer and *put()* adds an item to the buffer. The synchronization constraint is that when the buffer is empty, we cannot *get()* from the buffer, and when the state of the buffer is full, we cannot *put()* into the buffer.

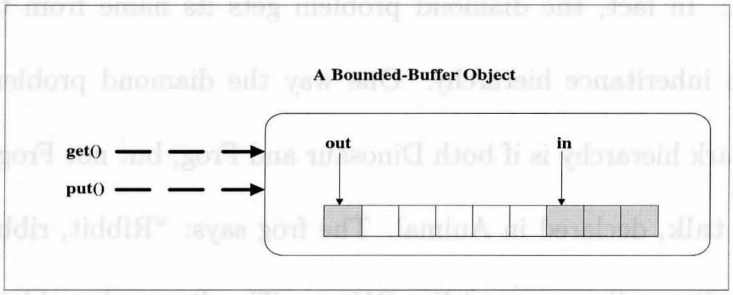

Figure 4.2: A Bounded Buffer Object

First, we define a base class *buf* with two methods: *get()* and *put()*. Then we define a sub-class *pbuf*, which adds a new method called *gput()*. The *gput()* adds an item to the buffer, and should be called immediately after *get()*. In this situation, we need to modify both *put()* and *get()*, since they cannot be called after *get()*. Thus the encapsulation of *put()* and *get()* in the base class is broken. This phenomenon is called *inheritance anomaly*.

In Petri Nets, the problem is considered to be closely connected to the concept

of synchronization of concurrent objects. When a concurrent object is in a certain state, it can accept only a subset of its entire messages according to its restriction. The restriction imposed on the set of acceptable messages is called *synchronization constraint.* Usually, in Petri Nets we use states to control object behaviour with respect to synchronization constraints. In the bounded buffer example, we set up three states: empty, partial, and full. Upon creation, the buffer is in the empty state, and the only message acceptable is *put()*; arriving messages *get()* are not accepted. When a *put()* message is processed, the buffer is no longer empty and can accept both *put()* and *get()* messages, reaching a *partial* (non-empty and non-full) state. When the buffer is full, it can only accept *get()*, and after processing the *get()* message, it becomes partial again. Figure 4.3 shows how Petri Nets model the bounded buffer.

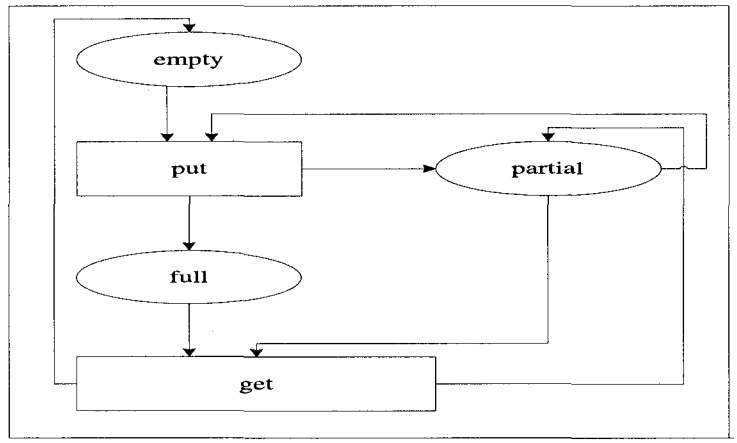

Figure 4.3: A CP-net describing the bounded buffer problem

Unfortunately, sometimes this synchronization scheme cannot be efficiently inher-

ited without non-trivial class re-definitions. This will lead to *inheritance anomaly.* There are three reasons why inheritance anomalies occur [MY93]:

(1) Partitioning of Acceptable States

An object has a set of states. This set can be partitioned into disjoint subsets according to the synchronization constraint of the object. When a new method is added to the definition of the subclass, the partitioning of the set of states in the parent class may need to be further partitioned in the subclass, since the synchronization constraint of the new method may not be properly be accounted for in the partitioning of the parent class. In the bounded buffer example, when we want to add a new transaction *get2* that removes two items from the buffer in the subclass, a partitioning of **partial** into **one** and **partial** is necessary in order to distinguish the state in which one element is in the buffer. In this way, we have to add another state, one, as well as redefine the state partial and set of arcs in the subclass, as in Figure 4.4.

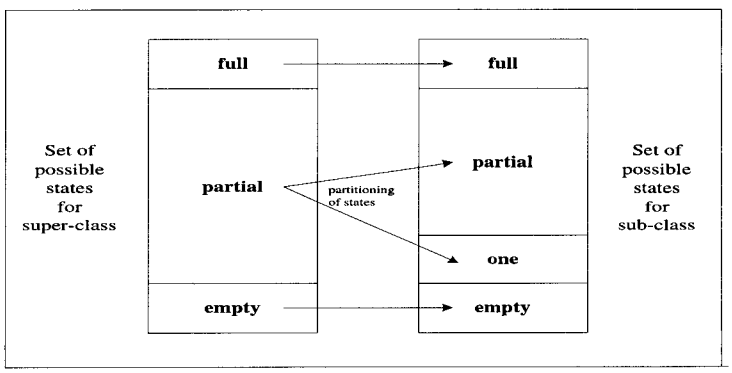

Figure 4.4: Conceptual Illustration of the State Partitioning Anomaly

Figure 4.5 presents this problem in CP-net.

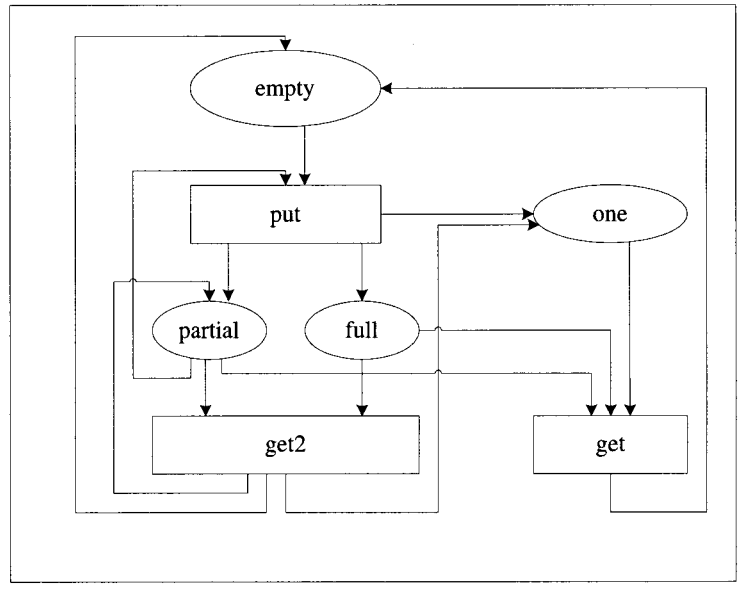

Figure 4.5: A CP-net describing the Partitioning Anomaly

(2) History-only Sensitiveness of Acceptable States

There are two different views in modelling the state of objects. One is the external view, in which the state is captured indirectly by the external, observable behaviour of the object. The equivalence of two objects is determined solely by how they respond to external experiments, not by how their internal structures are composed. The other is the internal view, in which the state is captured by the evaluation of the state variables in the implementation of the object. These two views on state are not identical. There are sets of states with elements that can be distinguished using the external view, but are indistinguishable in the internal view. In the bounded

buffer example, if we add a new transaction *gget* to the subclass, we can see such distinction. The behaviour of *gget* is almost identical to the transaction *get*, with the sole exception that it cannot be accepted immediately after the transaction *put*. From the external view, the set of states for the buffer is {empty, partial, full}. But from the internal view, in order to implement *gget*, we need to add a state variable **after-put** to distinguish the state that has already processed the transaction *put* from at has not yet processed it. Therefore, the state of the object in internal view must be redefined in order to match the state in the external view. For this purpose, the states in a parent class must be modified as in Figure 4.6. Thus, the state of the object is *history-only sensitive* with respect to the internal view.

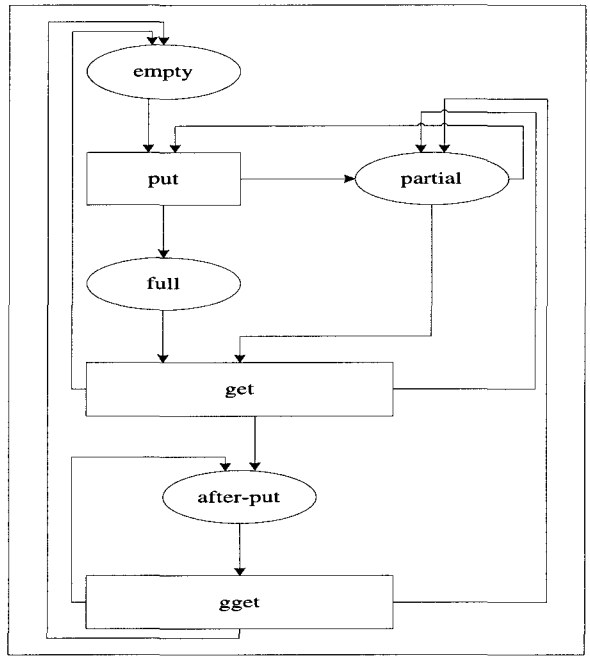

Figure 4.6: A CP-net describing the history-only sensitiveness Anomaly

(3) Modification of Acceptable States

When a set of states is inherited from the parent, according to the new synchronization constraint the condition of these states has to be modified. For instance, in the bounded buffer example, when we add a mix-in class **Lock** to the **b-buf** to create the class **lb-buf**, we would assume that it would not affect the definition of other methods, since the state of the object with respect to lock and unlock is totally orthogonal to the effect of other messages. However, this is not the case: the result of mixing-in of **Lock** modifies the set of states in **lb-buf** in which the transitions inherited from **b-buf** could be executed. Figure 4.7 shows this problem.

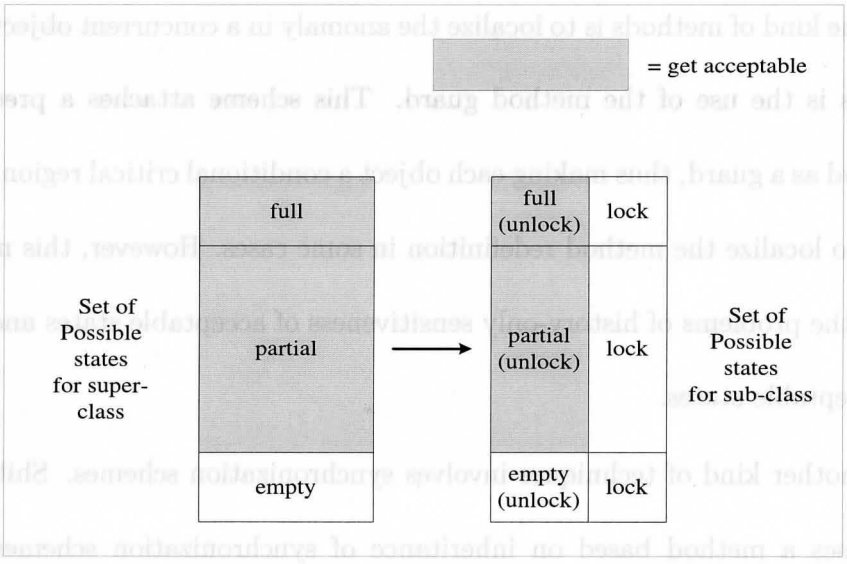

Figure 4.7: Conceptual Illustration of the State Modification Anomaly

Figure 4.8 describes this problem by using CPN.

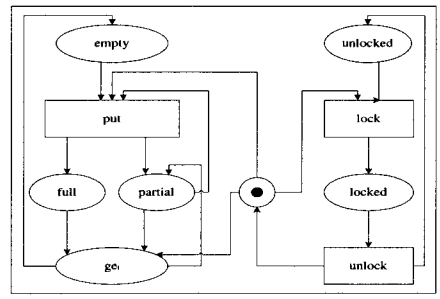

Figure 4.8: A CP-net describing the State Modification Anomaly

Recently, there have been many research proposals on how to minimize the effect of the inheritance anomaly. The following is brief review of them:

One kind of methods is to localize the anomaly in a concurrent object. An example of this is the use of the method guard. This scheme attaches a predicate to each method as a guard, thus making each object a conditional critical region. This method tries to localize the method redefinition in some cases. However, this method cannot solve the problems of history-only sensitiveness of acceptable states and modification of acceptable states.

Another kind of techniques involves synchronization schemes. Shibayaba [Shi90] proposes a method based on inheritance of synchronization schemes, so that the amount of code needing to be redefined can be minimized. Shibayaba categorizes these schemes into *primary*, *constraint* and *transition* methods. Each of them can be separately defined, inherited and overridden. The categorization of methods is as

follows:

- A *primary method* is responsible for the tasks other than object-wise synchronization.
- A *constraint method* needs to be overridden in the event that the method guards of parent class must be changed; the corresponding primary methods are unaffected.
- A *transition method* determines how the messages are delegated. Its redefinition allows dynamic modification of the delegation path.

By separating the synchronization code from other parts of method definition, the amount of redefinition is minimized.

A third way to solve the inheritance anomaly is to completely eliminate the synchronization code. Meseguer introduced his rewriting logic in [Mes90]. He considers the inheritance anomaly a problem caused by the presence of synchronization code. Thus, he completely eliminates synchronization code by using order-sorted rewriting logic. With this rewriting logic, all kinds of information about legal messages, state switches and so on are implicitly contained in the rewriting rules; explicit synchronization code is no longer needed. In this way, the conditions placed on the rewrite rules can serve as a guard; thus, the state partitioning anomaly does not arise. In addition, rewrite rules can be very flexible, since they operate on the term structures

as first class values. Therefore, history information can be encoded within the term structure in a straightforward way. Meseguer's proposal possesses the flexibility to provide a clean solution for the inheritance anomaly. However, there is still work to be done.

In OO-CPN, the solution for the avoiding inheritance anomaly in Petri Nets is to localize the anomaly in a concurrent object, then separate the synchronization constraints from the actions. In every class definition, there is a special part defined for constraints. This part is divided into two sections: *pre* for preconditions, and *post* for post conditions. In this way, when the inheriting class introduces new synchronization constraints, we only need to modify this part without accessing the transaction part. This method not only prevents encapsulation breakage but also minimizes the redefinition. Furthermore, since the constraint part is divided into two sections, both section do not need to be changed simultaneously. This also minimizes redefinition. The following is a detailed description of how this scheme solves the three kinds of inheritance anomaly. (Figure 4.9 shows the super class: buf)

(1) Partitioning of Acceptable States

From Figure 4.10 we can see that we do not need to add another state *one* to distinguish the state in which one element is in the buffer. What we should do is to write a proper precondition for the *get2* method, then simply add *get2* to the class

net without modification to the methods *put* and *get* in the super class. This protects the encapsulation.

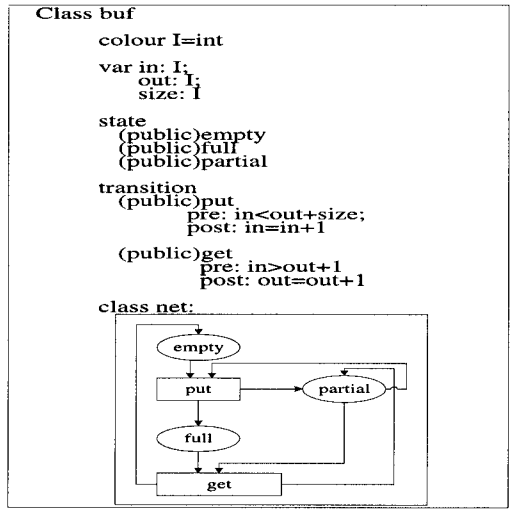

Figure 4.9: The base-class: buf

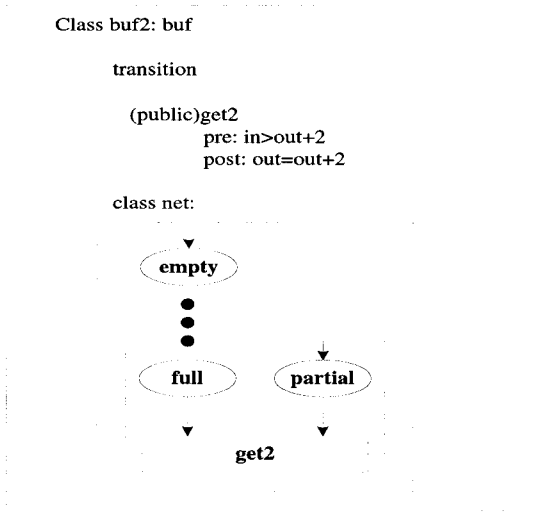

Figure 4.10: The sub-class buf2

(2) History-only Sensitiveness of Acceptable States

In Figure 4.11, another variable, *after-put*, is added, and then the postcondition of *put* and *get* are modified. Similarly, we need not add another state after-put; we simply write the proper precondition for the *gget* method, then add it to the class net. In this case, we modify the constraint parts of the methods in the super class, but do not redefine the states and set of arcs in the super class. This avoids heavy encapsulation breakage.

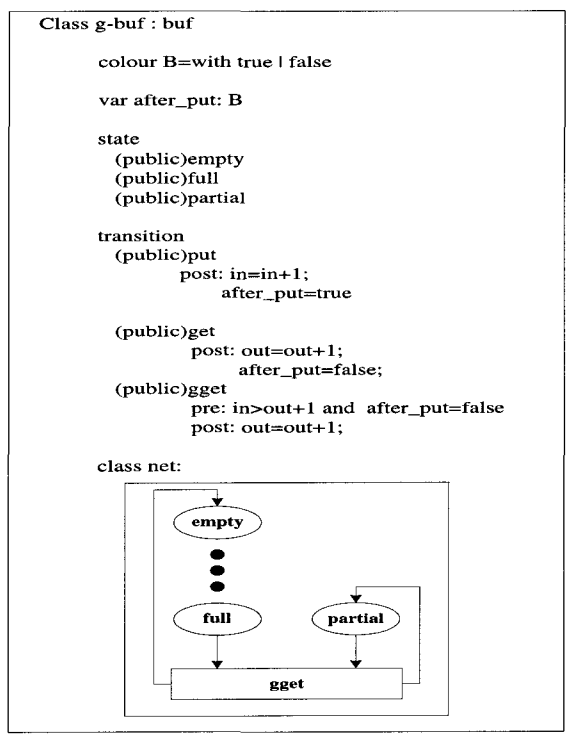

Figure 4.11: The sub-class gBuf

(3) Modification of Acceptable States

Figure 4.12 shows that with the mix-in class lock, all we need to do is change the precondition for *put* and *get*. We do not need to change the class net here. This minimizes the redefinition.

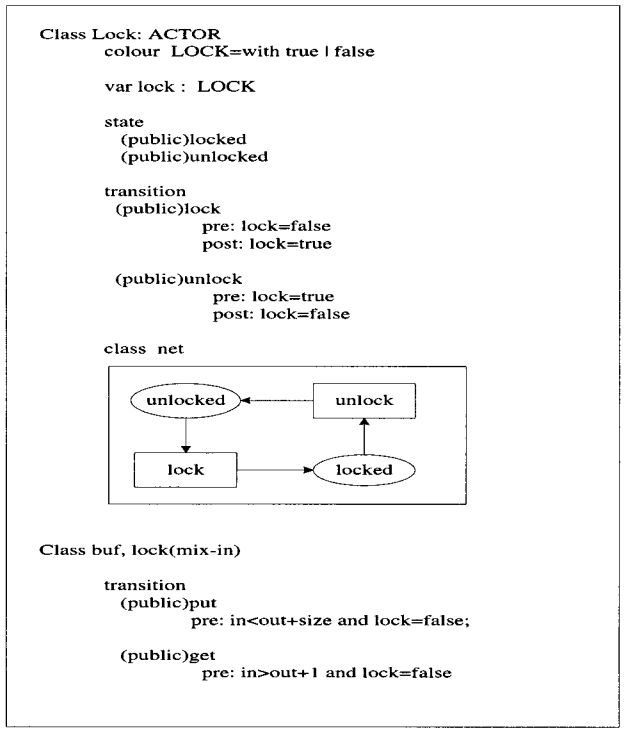

Figure 4.12: The sub-class lBuf

4.4 Polymorphism

Polymorphism allows the programmer to handle great complexity by allowing the creation of a standard interface for related activities.

There are two kinds of polymorphism:

overloading A named function can vary depending on the parameters it is given. For example, we define a function that finds information about a certain employee. If given a variable that is an integer, the function would seek a match against a list of employee numbers; if the variable is a string, it would seek a match against a list of names. In either case, both functions would be known by the same name. This type of polymorphism is known as overloading.

overriding Given a class hierarchy, a sub-class will inherit the methods in the super-class. However, if the sub-class includes a method with the same"signature" as a method in the super-class, the super-class will not be inherited. We say that the sub-class method "overrides" the super-class method.

In OO-CPN, both overloading and overriding are defined. Overriding enables the transition in the sub-class to replace the transition in the super-class in three ways: it can replace only the body, it can replace the constraints, it can replace both of them. For overloading, transitions will behave differently according to the messages they receive. The abstract transition in OO-CPN is a type of overloading. In the super-class, these abstract transitions will declare a certain kind of behaviour, but they will not define how objects will behave. The child classes of this class will specify more detailed behaviour of these transitions. For example, we declare an abstract transition, move, in the animal class. There are two subclasses of animal class: one

is dog, the other is bird. Then, for dog, we define move as run; for bird, the move is defined as fly. In the net description, abstract transitions are represented by dashed rectangles.

Polymorphism plays an important role in object-oriented design. It allows us to design software with great generality. In OO-CPN, by using this feature we can override the constraints of the super-class. This helps prevent the inheritance anomaly.

4.5 Communication Channel

In OO-CPN, objects communicate through the communication channel, which was proposed in Lin He's Master's thesis [He01]. The channel is a one-way channel. Objects communicate through it by token exchange.

OO-CPN adopts CSP notation to describe the token exchange. When an object sends a token **v** through the channel **c**, we denote it as **c!v**. When an object receives a token **v** through the channel **c**, we denote it as **c?v**. If the event **c!v** triggers a transition, we call this transition **!v**-transition. If the event **c?v** triggers a transition, we call this transition **?v**-transition. A communication between two transitions is enabled only if one of the transition is **!v**-transition and the other is **?v**-transition.

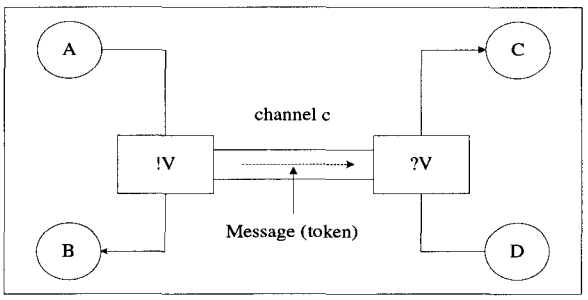

Figure 4.13: Communication Channel in OO-CPN

4.6 Example of OO-CPN: Reader and Writer

To illustrate the structure and behaviour of OO-CPN, this section gives a simple example using OO-CPN. The formal definition will be in the next chapter.

This system contains four roles: reader, writer, checker and marker. The reader reads files from the buffer, and the writer writes files to the buffer. The marker reads files from the buffer; if he or she finds some mistakes, he or she will make mark on the files. The checker reads these marks, then makes corrections according to the rules. There are a few constraints for these roles. When one writer is writing a file, no one else can read, write, check or make the mark on this file. When one marker is working on a file, no one else can write or check this file. When one checker is checking a file, no one else can write, check or make the mark on this file.

Figure 4.14 describes the whole system.

Figures 4.15 to 4.21 depict the classes for the read and write system.

Figure 6.2 shows the class diagram that describes the relationship between classes.

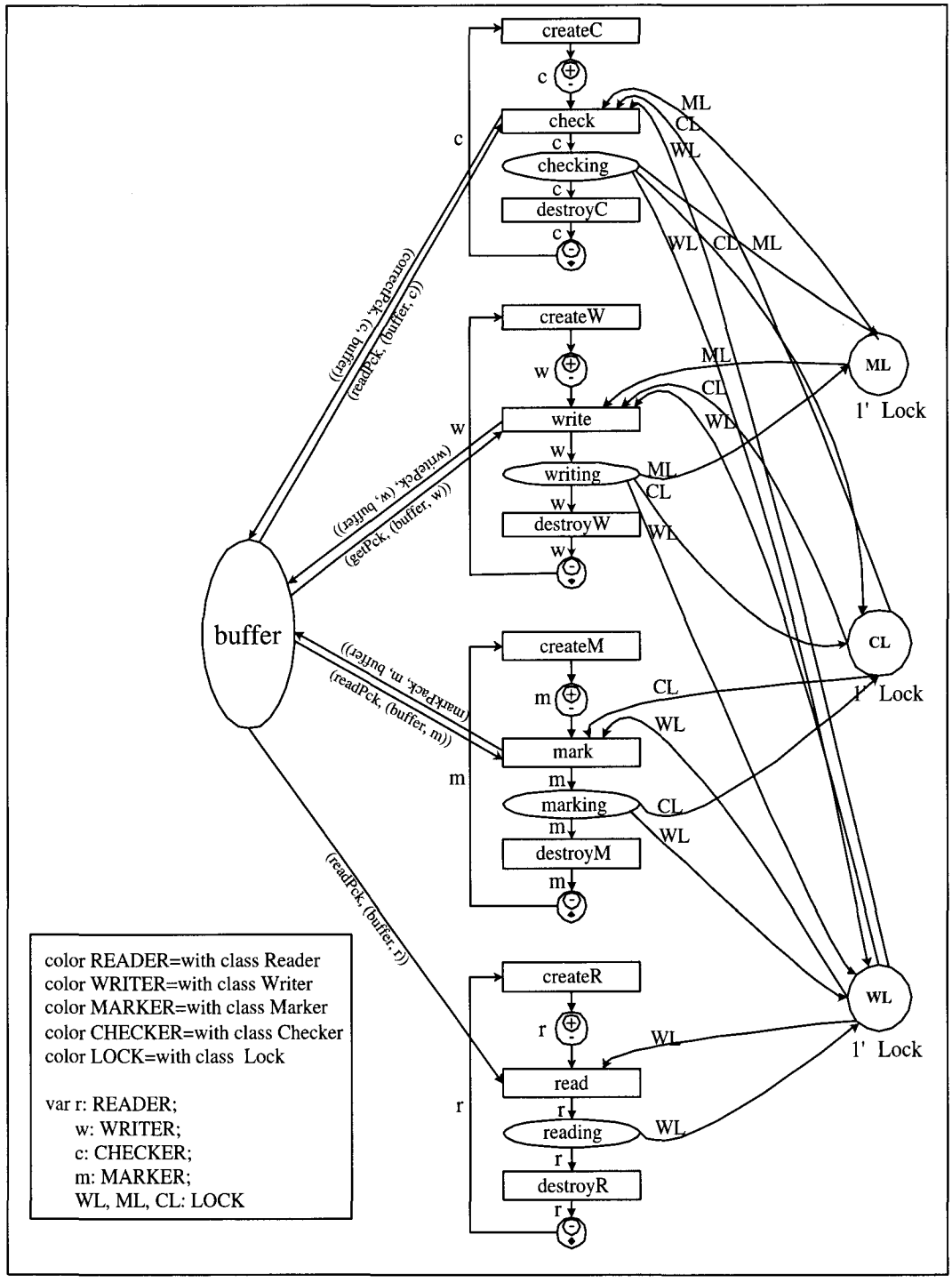

Figure 4.14: OOCPN for read and write system

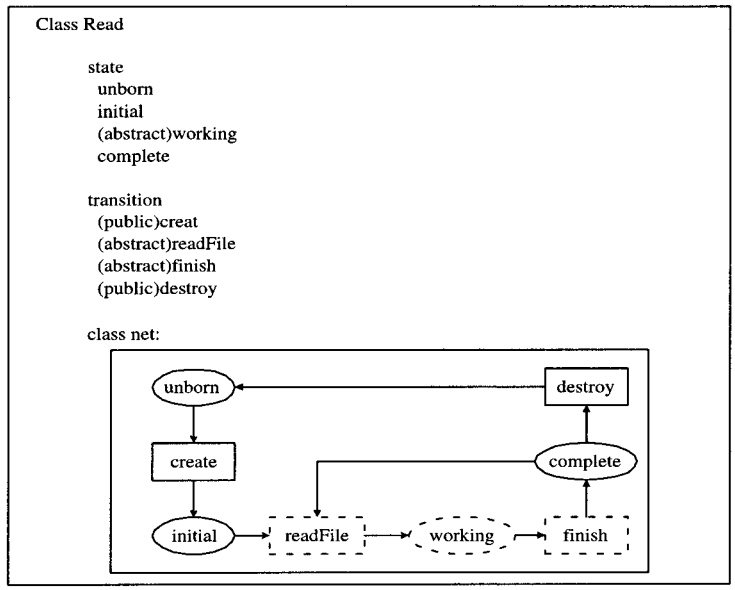

Figure 4.15: Class of Read

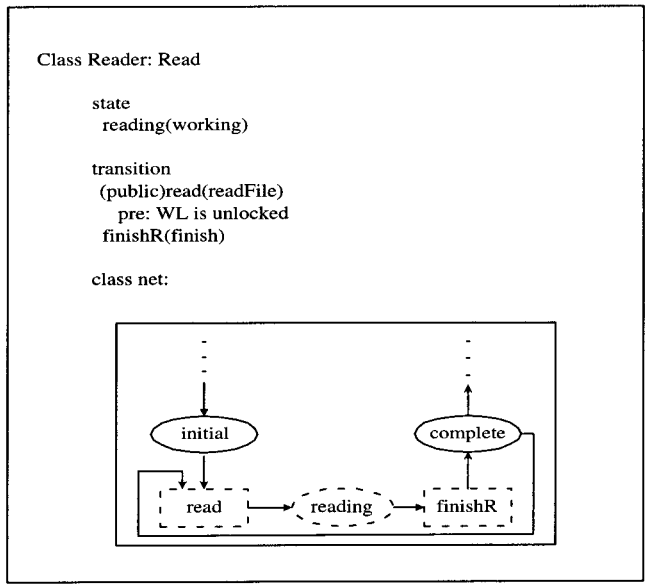

Figure 4.16: Class of Reader

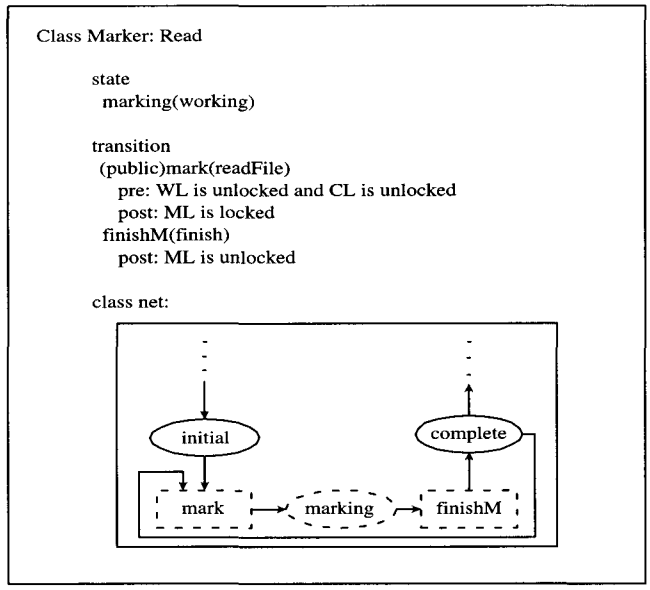

Figure 4.17: Class of Marker

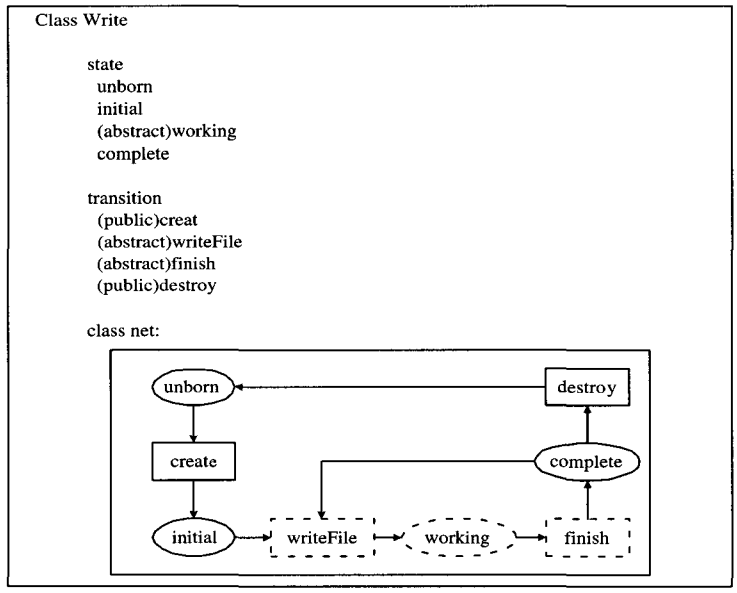

Figure 4.18: Class of Write

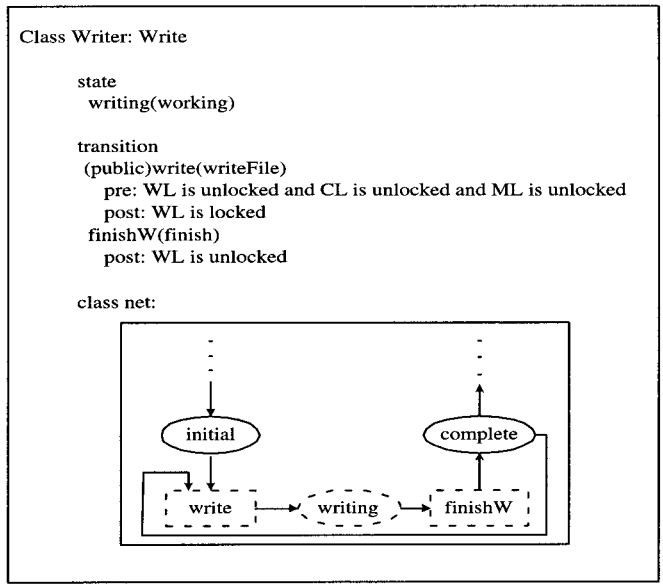

Figure 4.19: Class of Writer

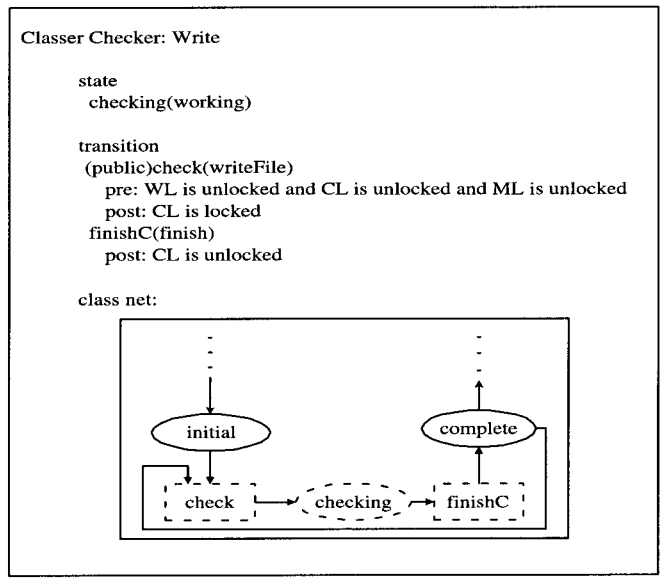

Figure 4.20: Class of Checker

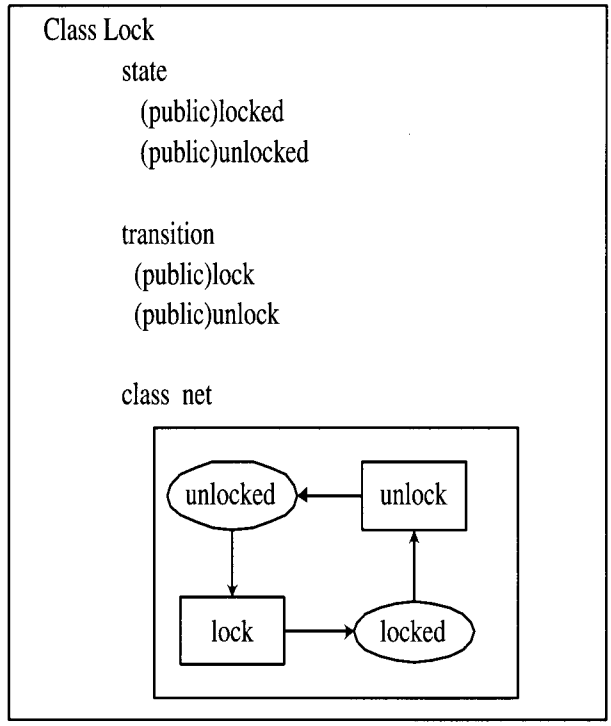

Figure 4.21: Class of lock

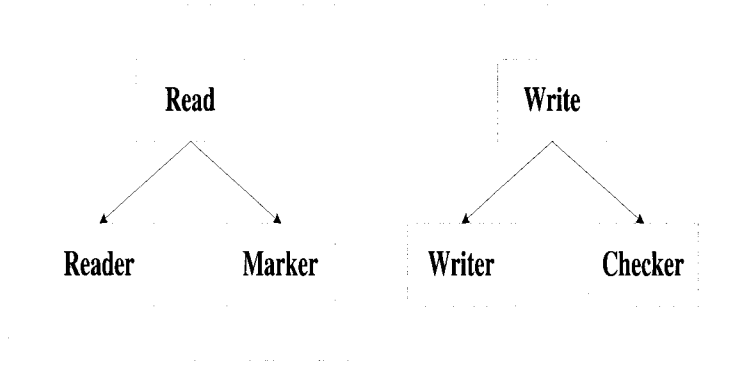

Figure 4.22: Class of diagram

Chapter 5

Formal Definition of OO-CPN

This chapter contains the formal definitions of Object-Oriented Coloured Petri Nets and their behaviour. An Object-Oriented Coloured Petri Net is formally defined as a tuple. This tuple form is suitable for formulating general definitions and proving theorems which apply to OO-CPN. Any concrete net, created by a modeler, will always be specified in terms of an OO-CPN diagram.

Section 5.1 defines the structure of improved Object-Oriented Coloured Petri Nets. Section 5.2 defines the behaviour of improved OO-CPN. Section 5.3 gives the formal specification for the reader-writer system which is stated in the previous chapter.

5.1 Structure of OO-CPN

This section gives the formal definition of the Object-Oriented Coloured Petri Nets. Before giving the abstract definition, let's first introduce some notations.

- Notation $[x \rightarrow y]$ means a map x into y. This is many-to-one correspondence between set x and set y.

Definition 5.1. An OOCP-net is a pair OOCPN = (Net, CLASS)

Definition 5.2. Net is a tuple

$Net = (\Sigma, P, T, OBJ, CH, A, N, C, G, E, I)$

satisfying the requirements below:

1. Σ is 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: $P \cap T = P \cap A = T \cap A = \emptyset$.
5. OBJ is a finite set of objects, each object in this set is a tuple $OBJ = (OID, CID, CRE, DEL)$ satisfying the following requirements:
   (a) OID is a set of **object identifier** with $\forall i, j \in OID : i \neq j \Rightarrow \Big(OBJ(i) \neq OBJ(j)\Big) \Rightarrow (P_i \cup T_i \cup A_i) \cap (P_j \cup T_j \cup A_j) = \emptyset$

(b) CID is a set of **class identifier** with $\forall obj \in OBJ : [Type(obj) \in CID]$

(c) CRE is an initialization function denoting the creation of an object. $CRE \in [P \rightarrow OID]$

(d) DEL is a function denoting the deletion of an object. $DEL \in [P \rightarrow OID]$

6. N is a **node** function. $N \in [A \rightarrow (P \times T) \cup (T \times P) \cup (T \times OBJ)]$.

7. C is a **colour** function. $C \in [P \rightarrow \Sigma]$.

8. G is a **guard** function. It is defined from T into expressions such that: $\forall t \in T : [Type(G(t)) = \mathbf{B} \wedge Type(Var(G(t))) \subseteq \Sigma]$.

9. 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 $N(a)$.

10. I is an initialization function. It is defined from P into closed expressions such that: $\forall p \in P : [Type(I(p)) = C(p)_{MS}]$.

11. CH is a finite set of channels, each channel is a tuple $CH = (CHID, T_{com}, E_{com})$ satisfying the following requirements:

    - $CHID$ is a channel identifier with: $(P_{OOCPN} \cup T_{OOCPN} \cup A_{OOCPN}) \cap CHID = \emptyset$

- T_{com} is a finite set of communication transition along the channel. $T_{com} \subseteq T$
- E_{com} is an expression function for communication transition along the channel. It is defined from T_{com} into expressions. An expression for communication transition has the form: (expr, (send, rec)), with *expr* being an expression,

  $(\forall t \in T_{com}, send(t) \in OID, rec(t) \in OID, E_{com}(t) \neq \emptyset \mid$

  $\forall (expr, (send, rec)) \in E_{com}(t) : [Type(Var(expr)) \subseteq \Sigma)$

Notes:

- 1. The set of **colour sets** determines the types, operations and functions that can be used in the net inscriptions.
- (2)+(3)+(4) The **places**, **transitions** and **arcs** are described by three sets P, T, and A which are required to be finite and pairwise disjoint.
- 5. OBJ is a finite set of objects, each object in this set is a tuple $OBJ = (OID, CRE, DEL)$ satisfying the following requirements:
  1. The places, transitions and arcs of different objects are disjointed.
  2. All objects have types that belongs to the set of classes.
  3. The **initialization** function CRE maps each place, P, to an object identifier. This means that each object on P corresponds to an unique object identifier.

4. The function DEL maps each place, P, to an object identifier.This means that each object on P corresponds to an unique object identifier.

- 6. The **node** function maps each arc into a pair where the first element is the source node and the second the destination node. The two nodes have to be of different kind(i.e., one must be a place while the other is a transition).

- 7. The **colour** function C maps each place, P, to Σ. This means that each token on P must have a token colour that belongs to Σ.

- 8. The **guard** function G maps each transition, t, to an expression of type boolean, i.e., a predicate. Moreover, all variables in $G(t)$ must have types that belong to Σ.

- 9. The arc expression function E maps each arc, a, into an expression which must be of type $C(p(a))_{MS}$. (Here, MS is the notation for the multi-set.) This means that each evaluation of the arc expression must yield a multiset over the colour set that is attached to the corresponding place.

- 10. The initialization function I maps each place, P, into a closed expression which must be of type $C(p)_{MS}$, i.e., a multiset over $C(p)$.

- 11. CH is a finite set of channels, each channel is a tuple $CH = (CHID, T_{com}, E_{com})$ satisfying following requirements:

- The set of channel identifiers must be disjointed with the place, transition, and arc sets.
- The **communication transition** T_{com} is a subtype of transition.
- The expression function for communication transition E_{com} includes expression function, sender's ID and receiver's ID. The type of the variables used must be included in Σ.

Definition 5.3. A class is a tuple $CLASS = (CID, \Sigma, CONST, VAR, S, TRAN, class-net)$

1. CID is a **class identifier** such that $CID \subseteq \Sigma$
2. Σ is a finite set of non-empty types, called **colour sets**.
3. $CONST$ is a finite set of **constant values** such that
   $\forall c \in CONST : Type(c) \subseteq \Sigma$
4. VAR is a finite set of **variables** such that $\forall v \in VAR : Type(v) \subseteq \Sigma$
5. S is a finite set of **states**.
6. $TRAN$ is a finite set of **transitions** such that: $S \cap TRAN = \emptyset$. Every transition is a tuple $TRAN = (Tran, PRE, POST)$
   - $Tran$ is a transaction.

- PRE is a guard function which specifies the precondition that $\forall t \in T$: $[Type(PRE(t)) = \mathbf{B} \wedge Type(Var(PRE(t))) \subseteq \Sigma]$

- $POST$ is a guard function which specifies the postcondition of a transition. It is defined from T into expressions such that $\forall t \in T : [Type(Var(PRE(t))) \subseteq \Sigma]$

7. The classnet is a Net.

Notes: All the elements in the tuple have three access method: public, private and default as defined in the last section.

- 1. The set of **colour sets** determine the types that used in the class inscriptions.

- 2. The set of **class identifier** is types of object. It is subtype of the colour sets.

- (3)+(4)+(5)+(6)The constants, variable, states and transitions are described by sets CONST, VAR, S and TRAN. These four sets are required to be finite, and S and TRAN are pairwise disjoint. Every transition have two guard functions:

  - The guard function PRE maps each transition, TRAN, to an expression of type boolean, i.e., a predicate. Moreover, all variables in PRE(t) must have types that belong to Σ.

- The guard function POST maps each transition, TRAN, to an expression. Also, all variables in POST(t) must have types that belong to Σ.

- 7. The class-net is a Net.

5.2 Behaviour of Object Oriented Coloured Petri Nets

After defining the structure of OOCPN, let's consider their behaviour now.

Definition 5.4. An **object token element** is a pair (p,c) where $p \in P$ and $c \in CID$, while a **binding element** is a pair (t,b) where $t \in T$ and $b \in B(t)$. The set of all object token elements is denoted by OTE while the set of all object binding elements is denoted by OBE.

An **object creating** C_0 is a multi-set over OTE while a **step** is *non-empty* and *finite* multi-set over OBE. It is obtained by evaluating the CRE function: $\forall(p,c) \in OTE : C_0(p,c) = (CRE(p))(c)$.

Note: The object creation is represented as function defined on P.

Definition 5.5. When a step Y which is destroy an object is enabled in a marking M_1 it may occur , changing the marking M_1 to another marking M_2, defined by:

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

where $\forall$ v $\in$ Var(t): b(v) $\in$ CID.

Note: By executing step Y, an object is removed from the P.

5.3 Example

To illustrate the formal definition of OOCPN, the reader and writer problem stated in last chapter was used to show how the net can be represented as a many-tuple.

- $\Sigma = \{READER, WRITER, MARKER, CHECKER, LOCK\}$
- P = {initialR, initialW, initialC, initialM, reading, writing, checking, marking, unbornR, unbornW, unbornC, unbornM}
- T = {createR, createW, createC, createM, read, write, check, mark, finishR, finishW, finishC, finishM}
- A = {createRTOinitialR, initialRToread, readTOreading, readingTOdestroyR, destroyRTOunbornR, unbornRTOcreateR, createCTOinitialC, initialCTocheck, checkTOchecking, checkingTOdestroyC, destroyCTOunbornC, unbornCTOcreateC, createMTOinitialM, initialMTomark, markTOmarking, markingTOdestroyM, destroyMTOunbornM, unbornMTOcreateM, createWTOinitialW, initialWTowrite, writeTOwritinging, writingTOdestroyW, destroyWTOunbornW, unbornWTOcreateW,}

- N(a) = (SOURCE, DEST) if a in the form SOURCEtoDEST.

- $C(p) = \begin{cases} READER, & \text{if p} \in \{\text{initialR, reading}\}; \\ WRITER, & \text{if p} \in \{\text{initialW, writing}\}; \\ CHECKER, & \text{if p} \in \{\text{initialC, checking} \\ MARKER, & \text{if p} \in \{\text{initialM, marking}\}. \end{cases}$

- G(t) =true;

- $E(a) = \begin{cases} ML, & \text{if a} \in \{\text{MLTOcheck, MLTOwrite, chekingTOML, writingTOML}\}; \\ CL, & \text{if a} \in \{\text{CLTOcheck, CLTOwrite, CLTOmark,checkingTOCL, writingTOCl, markingTOCL}\}; \\ WL, & \text{if a} \in \{\text{WLTOcheck, WLTOwrite, WLTOmark, WlTOread, checkingTOWL, writingTOWL, markingTOWL, readingTOWL}\}; \\ (correctPck, (c, buffer)), & \text{if a=checkTObuffer;} \\ (readPck, (buffer, c)), & \text{if a=bufferTOcheck;} \\ (writePck, (w, buffer)), & \text{if a=writeTObuffer;} \\ (getPck, (buffer, w)), & \text{if a=bufferTOwrite;} \\ (markPack, m, buffer)), & \text{if a=markTObuffer;} \\ (readPck, (buffer, m)), & \text{if a=bufferTOmark;} \\ (readPck, (buffer, r)), & \text{if a=bufferTOread.} \end{cases}$

- $I(p) = \begin{cases} r, & \text{if p=initialR;} \\ w, & \text{if p=initialW;} \\ c, & \text{if p=initialC;} \\ m, & \text{if p=initialM.} \\ 1'Lock, & \text{if p=ML.} \\ 1'Lock, & \text{if p=CL.} \\ 1'Lock, & \text{if p=WL.} \end{cases}$

- OBJS = {r, w, c, m, wl, cl, ml} with

r: Type(r)∈ READER,

CRE:r(initialR,Reader)=(CRE(initialR))(Reader)

w: $Type(w) \in WRITER$,

$CRE : w(initialW, Writer) = (CRE(initialW))(Writer)$

c: Type(c)∈ CHECKER,

CRE:c(initialC,Checker)=(CRE(initialC))(Checker)

m: Type(m)∈ MARKER,

CRE:m(initialM,Marker)=(CRE(initialM))(Marker)

wl: Type(wl)∈ LOCK,

CRE:wl(initialWL,Lock)=(CRE(initialWL))(Lock)

cl: Type(cl)∈ LOCK,

CRE:cl(initialCL,Lock)=(CRE(initialCL))(Lock)

ml: Type(ml)∈ LOCK,

CRE:ml(initialML,Lock)=(CRE(initialML))(Lock)

- CHS = {bufCheckChannel, checkBufChannel, bufWriteChannel, writeBufChannel, bufMarkChannel, markBufChannel, bufReadChannel}
  - bufCheckChannel:
    - T_{com}(bufCheckChannel) = c?readPck
    - CHE(bufCheckChannel) = (readPck, (buffer,c)).

- checkBufChannel:
  - T_{com}(checkBufChannel) = c!correctPck
  - CHE(checkBufChannel) = (correctPck, (c,buffer)).
- bufWriteChannel:
  - T_{com}(bufWriteChannel) = w?getPck
  - CHE(bufWriteChannel) = (getPck, (buffer,w)).
- writeBufChannel:
  - T_{com}(writeBufChannel) = w!writePck
  - CHE(writeBufChannel) = (writePck, (w,buffer)).
- bufMarkChannel:
  - T_{com}(bufMarkChannel) = m?readPck
  - CHE(bufMarkChannel) = (readPck, (buffer,m)).
- markBufChannel:
  - T_{com}(markBufChannel) = m!markPck
  - CHE(markBufChannel) = (markPck, (m,buffer)).
- bufReadChannel:
  - T_{com}(bufreadChannel) = r?readPck
  - CHE(bufReadChannel) = (readPck, (buffer,r)).

Class Read:

- CID = Read
- $\Sigma = \emptyset$
- CONST = $\emptyset$
- VAR = $\emptyset$
- P = {unborn, initial, working, complete}
- T ={creat, readFile$_{abstract}$, finish$_{abstract}$, destroy}
- classnet: {
  $\Sigma = \emptyset$;
  P = {unborn, initial, working, complete};
  T = {creat, readFile, finish, destroy};
  A = {unbornTOcreat, creatTOinitial, initialTOreadFile, readFileTOworking, workingTOfinish, finishTOcomplete, completeTOdestroy, completeTOread, destroyTOunborn};
  N(a) = (SOURCE, DEST) if a in the form SOURCEtoDEST;
  C(p) = Read;
  G(t) = true;
  E(a) = $\emptyset$;
  I(p) =$\emptyset$;
  OBJ = $\emptyset$.

Class Reader:Read

- CID = Reader
- P = {reading(working)}

- T = { *read(readFile)* : (**PRE:** WL is unlocked);
  finish

- classnet: { $\Sigma = \emptyset$;
  P = {wait, reading, complete};
  T = {read, finishR,};
  A = {waitTOread, readTOreading, readingTOfinishR,
  finshRTOwait, finishRTOcompletRe, completeRTOread};
  E(a) = $\emptyset$.

Class Marker:Read

- CID = Marker

- P = {marking(working)}

- T = { *mark(readFile)*, **Pre:**WL is unlocked and CL is unlocked.
  Post:ML is locked ;
  finishM(finish), **Post:** ML is unlocked .

- classnet: { $\Sigma = \emptyset$;
  P = {wait, marking, complete};
  T = {read, finishM,};
  A = {waitTOmark, markTOmarking, markingTOfinishM,
  finshMTOwait, finishMTOcompleteM, completeMTomark};
  E(a) =$\emptyset$.

Class Write:

- CID = write

- $\Sigma = \emptyset$

- CONST = $\emptyset$

- VAR = $\emptyset$

- P = {unborn, initial, working, complete}

- T ={creat, writeFile$_{abstract}$, finish$_{abstract}$, destroy}

- classnet: $\begin{cases} \Sigma = \emptyset; \\ \text{P = \{unborn, initial, working, complete\};} \\ \text{T = \{creat, writeFile, finish, destroy\};} \\ \text{A = \{unbornTOcreat, creatTOinitial, initialTOwriteFile,} \\ \text{writeFileTOworking, workingTOfinish, finishTOcomplete,} \\ \text{completeTOdestroy, completeTOwriteFile, destroyTOunborn\};} \\ \text{N(a) =(SOURCE, DEST) if a in the form SOURCEtoDEST;} \\ \text{C(p) = Write;} \\ \text{G(t) = true;} \\ \text{E(a) = } \emptyset; \\ \text{I(p) = } \emptyset; \\ \text{OBJ = } \emptyset. \end{cases}$

Class Writer:Write

- CID = Writer

- P = {writing(working)}

- T = $\begin{cases} \textit{write}, & \textbf{Pre:} \text{ WL is unlocked and CL is unlocked} \\ & \text{and ML is unlocked} \\ & \textbf{Post:} \text{ WL is locked ;} \\ \textit{finishW}, & \textbf{Post:} \text{ WL is unlocked .} \end{cases}$

- classnet: $\begin{cases} \Sigma = \emptyset; \\ \text{P = \{wait, writing, complete\};} \\ \text{T = \{write, finishW,\};} \\ \text{A = \{waitTOwrite, writeTOwriting, writingTOfinishW,} \\ \text{finshWTOwait, finishWTOcompleteW, completeWTOwrite\};} \\ \text{E(a) = } \emptyset. \end{cases}$

Class Checker:Write

- CID = Checker
- P = {checking(working)}
- T = $\begin{cases} check(readFile), & \textbf{Pre:} \text{ WL is unlocked and CL is unlocked and ML is unlocked} \\ & \textbf{Post:} \text{ CL is locked ;} \\ finishC(finishC), & \textbf{Post:} \text{ CL is unlocked .} \end{cases}$
- classnet: $\begin{cases} \Sigma = \emptyset; \\ \text{P = \{wait, checking, complete\};} \\ \text{T = \{mark, finishC,\};} \\ \text{A = \{waitTOcheck, checkTOing, checkingTOfinishC,} \\ \text{finshCTOwait, waitTOcompleteC, completeCTOcheck\};} \\ \text{E(a)} = \emptyset. \end{cases}$

Class Lock:

- CID = Lock
- $\Sigma = \emptyset$
- CONST = $\emptyset$
- VAR = $\emptyset$
- P = {locked, unlocked}
- T ={lock, unlock}

- classnet:
  - $\Sigma = \emptyset$;
  - P = {locked, unlocked};
  - T = {lock, unlock};
  - A = {unlockedTOlock, lockTOlocked, lockedTOunlock, unlockTOunlocked};
  - N(a) = (SOURCE, DEST) if a in the form SOURCEtoDEST;
  - C(p) = Lock;
  - G(t) = true;
  - E(a) =$\emptyset$;
  - I(p) = $\emptyset$;
  - OBJ = $\emptyset$.

Chapter 6

Example of the Distributed Program Execution

This chapter models a distributed system specified in [JM96] using OO-CPN.

Section 6.1 introduces distributed program execution and the protocol used to implement remote object invocations. Section 6.2 presents the OO-CPN model of the protocol. Section 6.3 makes a conclusion for this example.

6.1 Introduction to Distributed Program Execution

BETA is a modern object-oriented language that has been developed at Aarhus University over the last 20 years. An introduction to the BETA language can be found in [MBMP93]. Basic BETA has the construct to deal with concurrency issues. Recently, BETA has been extended to cope with distributed specifics. It allows objects to reside on different computers. Such objects may interact by means of remote object invocation, which in many respects is similar to remote procedure calls (RPC). Remote object invocation is implemented by means of a protocol contained in an application framework called DistBeta. The structure of distBeta is composed of as the following:

- *Ensemble*: An ensemble is the operating system, which is on a concrete computer in a network.

- *Shell*: A shell is a process that resides in a specific ensemble. A shell may communicate with other shells, which are in a remote ensemble or in their own ensembles. Moreover, a shell may communicate directly with its own ensemble.

- *Thread*: Each shell may contains a set of threads (lightweight processes). The number of threads may vary dynamically. However, each shell always has:

- at least one **main thread** executing the main program of the shell.
- exactly one **RPC handler** taking care of messages passing along the network and serialization, and unserialization of parameters.

- *Identifier*: Inside a shell, each object has a local identifier. However, each object also has a unique global object identifier, OID, which can be used in the entire system. Each shell keeps two tables that are used to map local identifiers into global identifiers and vice versa. One of the tables is for local objects while the other is for remote objects.

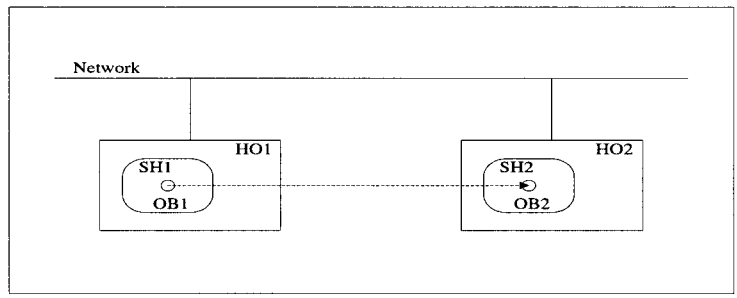

Figure 6.1: Example of a remote object invocation

Figure 6.1 shows an object OB_1 (in a shell SH_1 in a host HO_1) invoking a remote object OB_2 (in a shell SH_2 in a host HO_2). The protocol for the remote object invocation involves the following sequence of actions:

1. OB_1 determines the OID of OB_2 by looking up the global object identifier table. The parameters are serialized. OB_1 uses a method from the RPC handler of SH_1 to start the remote object invocation.

2. A request message is sent from HO_1 to HO_2. The message contains OIDs of both the sender and the receiver. $OB_1$is blocked.

3. The RPC handler in SH_2 receives the incoming request. The parameters are unserialized. A worker thread is allocated. The local identifier of OB_2 is determined from the OID using a table in SH_2.

4. The object OB_2 is invoked.

5. The worker thread receives the result. The result is serialized. Control is returned to the RPC handler in SH_2. The worker thread is released.

6. The message package is sent from HO_2 to HO_1.

7. The result is received by the RPC handler of SH_1, The result is unserialized.

6.2 Introduction to Distributed Program Execution

The OO-CPN model emphasizes describing the basic flow of control, the sharing of resources and the competition for access to critical sections.

There are seven classes in this system: Ensemble, Shell, Thread, Main, User, Worker and RPC. Figure 6.2 shows the relationship between them.

Figure 6.3 shows the OO-CPN of DistBeta at the system level.

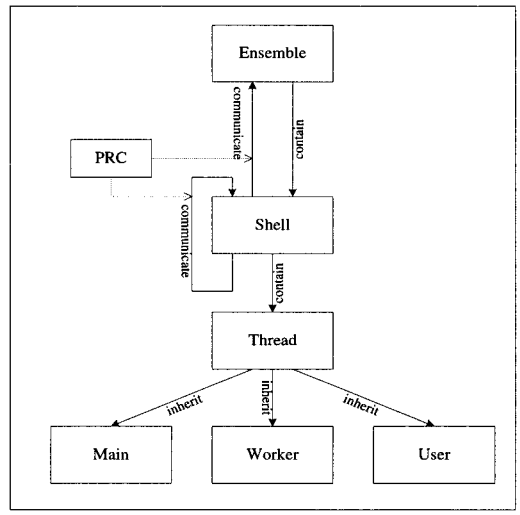

Figure 6.2: Example of a remote object invocation

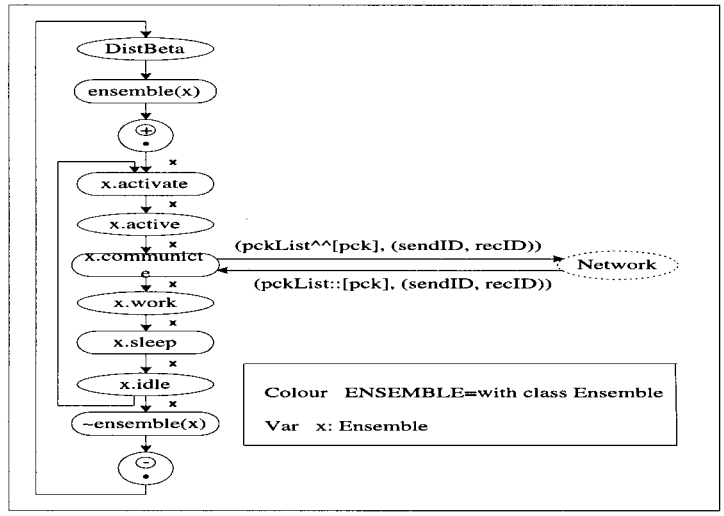

Figure 6.3: Example of a remote object invocation

The $\oplus$ in the net represents the addition of a new object to the system. The $\ominus$ represents deletion of an object from the system. When the transition *communicate* occurs, a new packet is added to the output queue for the corresponding shell or a

packet is removed from the head of the input queue. In the net, ($pckList \wedge \wedge[pck]$, (sendID, recID)) represents sending a packet to net, where $\wedge\wedge$ operator adds a new packet to the packet list. (pckList::[pck], (sendID, recID)) represents receiving a packet from the net, where :: operator takes a packet from the packet list.

Figures 6.4 to Figures 6.14 show the classes of Ensemble, Shell, RPC, Thread, Main, Worker and User, where Thread is the parent of Main, User and Worker.

```
Class Ensemble:

Colour SHELL = with class Shell;
       I = int

Const ensID: I /* The ID of the current ensemble */
      #ens: I /* The ID of the ensemble with which current ensemble
communicate */

Var x: SHELL;

State
  unborn: Not been created.
  born: A new ensemble is created.
  active: The ensemble is active.
  work: The ensemble is communicating.
  idle: The ensemble is inactivate.

Transition
  (public) ensemble(): create an ensemble.
      post: ensembleNumber = ensembleNumber + 1
  getID: The ensemble is assigned an ID.
      post: ensID is attached to current ensemble.
  (public) activate: activate current ensemble.
      pre: The state of ensemble is idle.
      post: The state of ensemble is activate.
  (public) sleep: The ensemble sleeps
      pre: The state of ensemble is activate.
      post: The state of ensemble is idle.
  (public)communicate: The ensemble communicates with network.
      pre: ensID and #ens are valid
  (public)~ensemble(): Destroy the ensemble.
      post: ensembleNumber = ensembleNumber - 1
```

Figure 6.4: Class of Ensemble

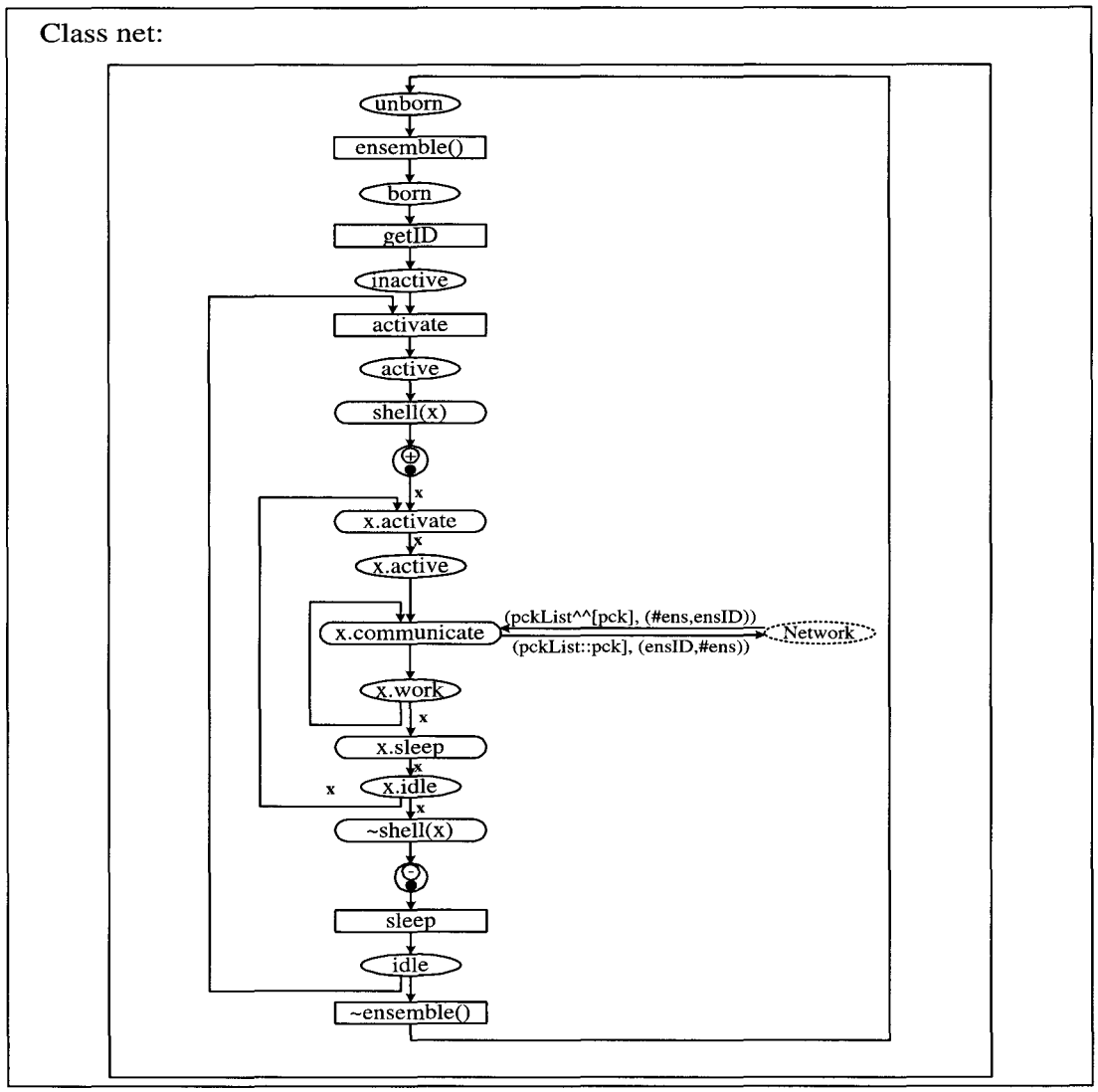

Figure 6.5: Class of Ensemble

```
Class Shell:

Colour RPC = with class Rpc
        MAIN = with class Main
        USER = with class User
        WORKER = with class Worker
         I = int

Const shelID: I /* The ID of the current shell */
      #shel: I /* The ID of the shell with which current ensemble
communicate */

Var r: RPC;
    m: MAIN;
    obj: USER;
    w: WORKER;

State
  unborn: Not been created.
  born: A new shell is created.
  inactive: The shell is inactive.
  active: The shell is active.
  idle: The shell is inactivate.

Transition
  (public) shell(): create an ensemble.
       post: shellNumber = shellNumber + 1
  getID: The shell is assigned an ID.
       post: shelID is attached to current shell.
  (public) activate: activate current shell
       pre: The state of shell is idle.
       post: The state of shell is activate.
  (public) sleep: The shell sleeps.
       pre: The state of shell is activate.
       post: The state of shell is idle.
  (public) send: The shell sents message to network.
       pre: shellID and #shel are valid && r is available
  (public) receive: The shell gets message from the network.
       pre: shellID and #shel are valid && r is available && w is  allocated
  (public)~shell(): Destroy the shell.
       post: shellNumber = shellNumber - 1
```

Figure 6.6: Class of Shell

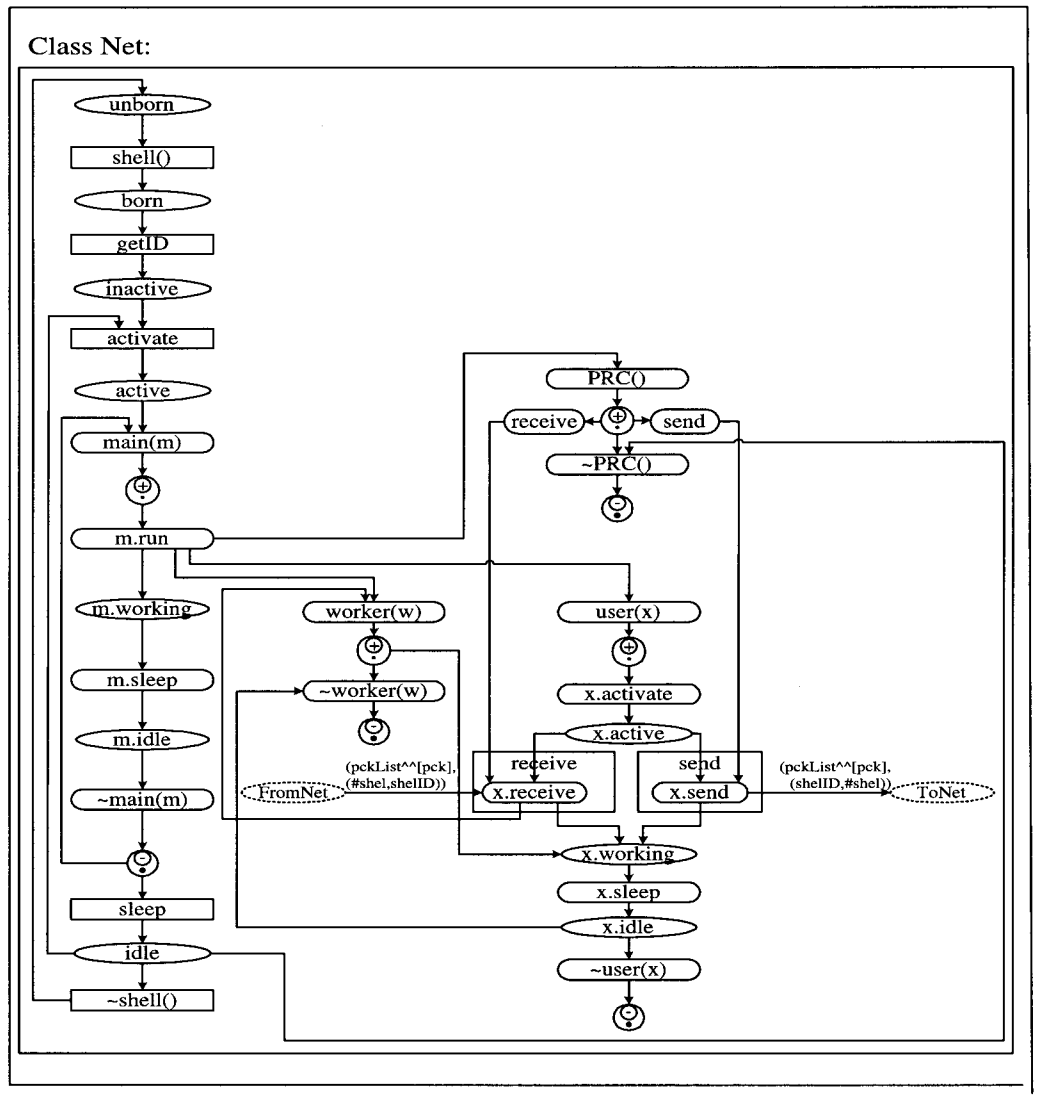

Figure 6.7: Class of Shell

Class Rpc

State:

unborn: Not been created.

active: The RPC is active.

Transition:

(public) RPC(): Creat a object RPC.

Pre: RPCNumber = 0

Post: RPCNumber = 1

(private) serialize: Serialize the parameters.

(public) send: Send the message to Network.

(public) receive: Get the message from the Network.

(private) unserialize: Unserialize the parameters.

(public) ~RPC(): Destroy the RPC.

Pre: RPCNumber = 1

Post: RPCNumber = 0

Class net:

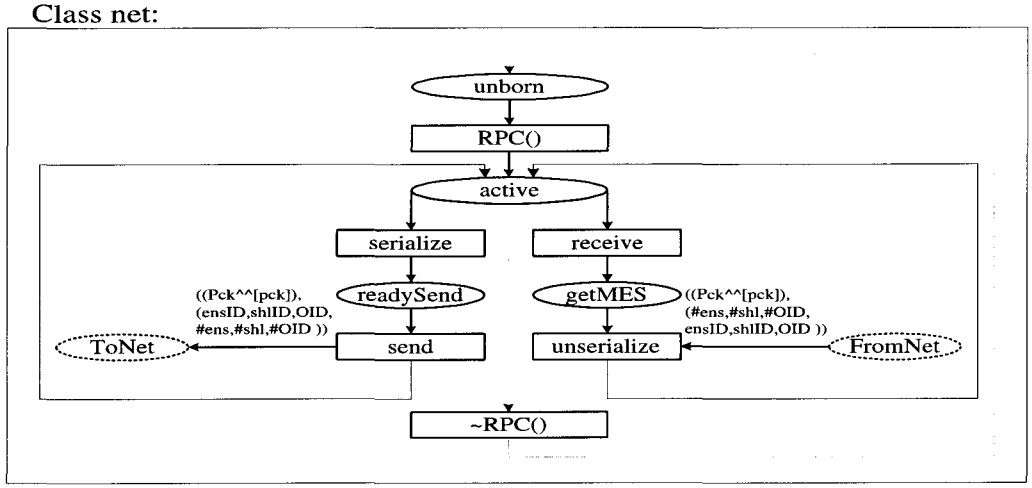

Figure 6.8: Class of Rpc

```
Class Thread:

Colour: (public)SEMAPHONE = with  0 | 1;
        (public)I = with int

Variable (public)s:SEMAPHONE;
         (public)threadNumber: I

State:
     (public)unborn: Not been created.
     (public)born: A new thread is created.
     (public)readyForLocalID: Enter the critical section and can get the ID.
     (public)getGlobalID: The thread is  assigned  a global ID.
     (public)inacive: The thread is not activated.
     (public)idle: The  thread  is not activated.
     (public)activate: The thread is activated.
     (public)working: The thread is working.

Transition:
     (public) thread(): Creat a thread.
          Post: threadNumber = threadNumber + 1 and  s = 0
     (private) P(s): Enter the critical section.
          pre: s = 0
          post: s = 1
     (private) cacheID: Get ID from global catche.
     (private) V(s): Exit from the critical section.
          pre: s = 1
          post: s = 0
     (public) activate: The thread is activated.
          pre: The state of the thread is inactivated.
          pre: The state of the thread is activated.
     (abstract)(public) work: The thread is working.
               pre: The state of the thread is active.
     (public) sleep: The thread goes to sleep.
               pre: The state of the thread is active.
               post: The state of the thread is idle.
     (public) ~thread(): Destroy the thread.
               post: threadNumber = threadNumber -1
```

Figure 6.9: Class of Thread

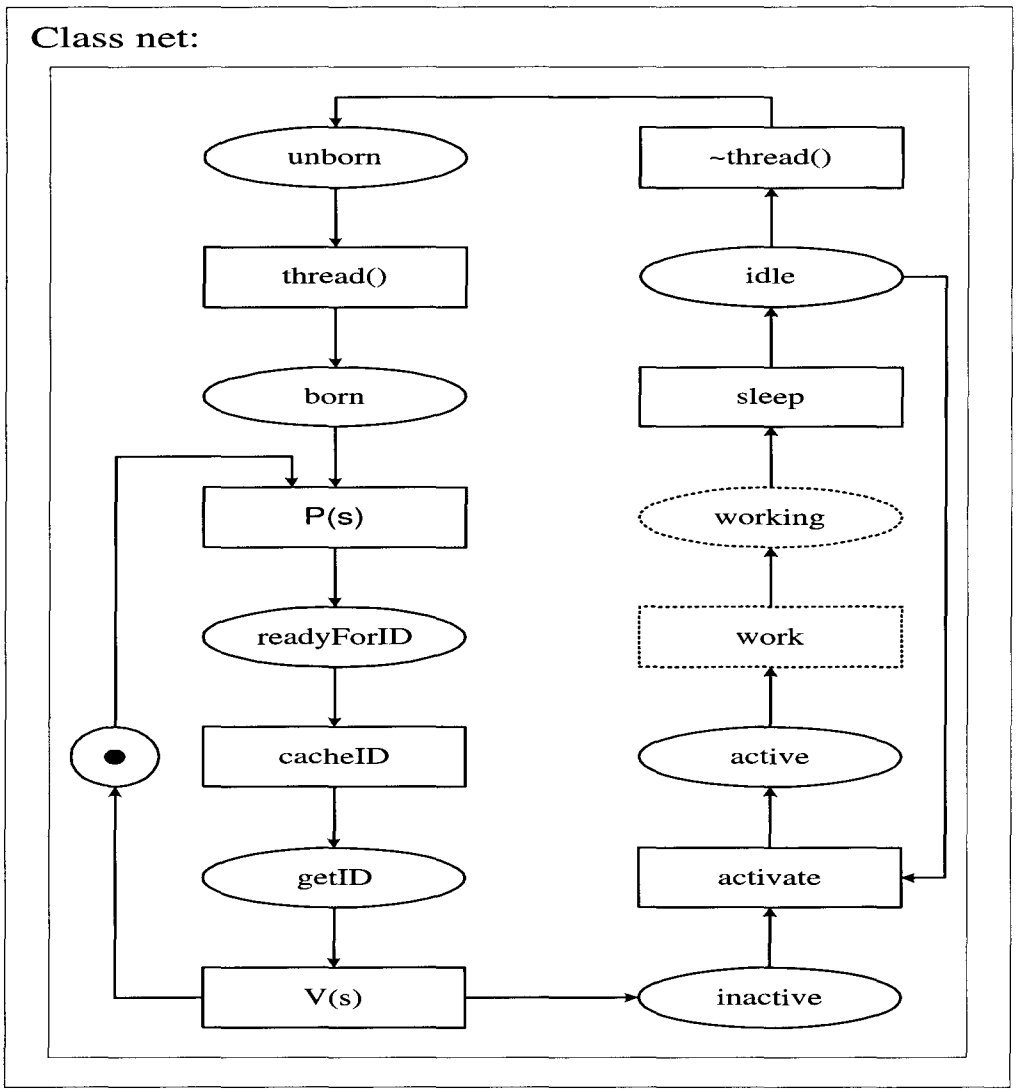

Figure 6.10: Class of Thread

Class Main: p ublic Thread

State:

(private)working: The thread is working with the request.

Transition:

(public)run: The main thread runs .

pre: The state of the thread is active.

(public)forkRPC: The main creates the object RPC.

pre: RPCNumber = 0

post: RPCNumber = 1

(public)forkThread: The main forks a new thread.

(public)destroyRPC: The main destroies the RPC.

(public)destroyThread: The main destroies the thread.

Class net:

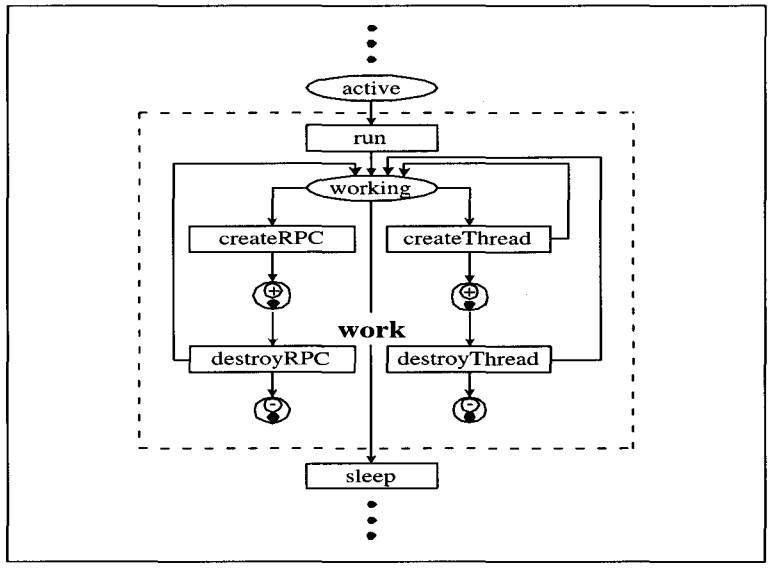

Figure 6.11: Class of Main

Class Worker: p ublic Thread

State:
(private)working: The thread is working with the request.
(private)readyForID: In the critical section.
(private)getLocalID: Gets the local ID for the callee thread.
(private)readyToWork: The thread is ready to work..
(private)wait: Wait for the result.
(private)getResult: Gets the result.
(private)finish: finish work.

Transition:
(private)P(s): Enter the critical section.
pre: s = 0;
post: s = 1;
(private)getID: Get the local ID of the callee thread.
(private)V(s): Exit the critical section.
pre: s= 1;
post: s = 0;
(public)invoke: Invokes the callee thread.
(public)getRes: Gets results from the callee thread.
(public)sendRes. Sends the results to the caller thread.
post: Pck = Pck ^^ [pck]

Class net:

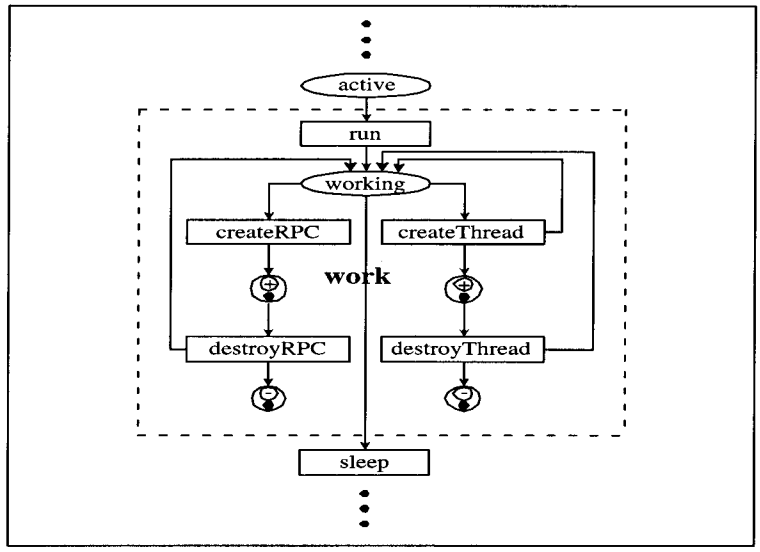

Figure 6.12: Class of Worker

```
Class User: public Thread

Colour: (public)SEMAPHONE = with  0 | 1;

Variable (public)s:SEMAPHONE;

State:
    (private)readyForID: Enter the critical section and can get the ID.
    (private)getID: The thread assigns the local ID for the receiver.
    (private)readySend: The thread is ready to send message.
    (private)invoked: The thread is invoked to do certain work.
    (private)finish: The thread finishes the work.
    (private)working: The thread is working with the request.

Transition:
    (private)P(s): Enter the critical section.
        pre: s = 0
        post: s = 1
    (private)getLocalID: Get local ID of the receiver from the ID table.
    (private)V(s): Exit from the critical section.
        pre: s = 1
        post: s = 0
    (public)send: The thread sends the message.
        pre: OID and #OID are valid.
        pre: Pck ^^ [pck].
    (public)receive: The thread receives the message.
            pre: OID and #OID are valid.
            post: Pck::[pck].
    (public)invoke: The thread is invoked to do some work.
            pre: The state of the thread is active.
            post: The state of the thread is idle.
    (public)doingJob: The thread is working on the task.
```

Figure 6.13: Class of User

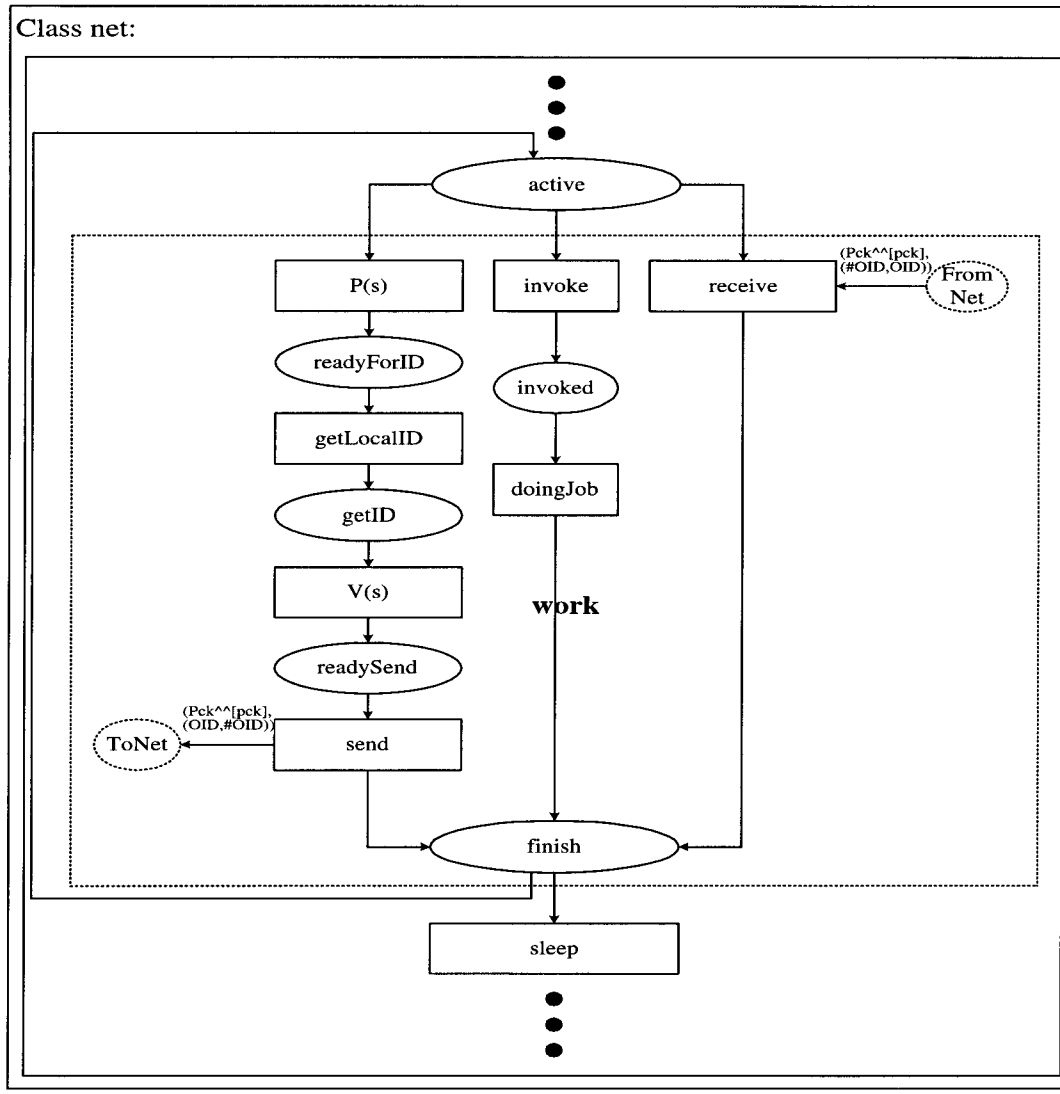

Figure 6.14: Class of User

6.3 Conclusion

In this example, a protocol for remote object invocation in BETA language was modelled using OO-CPN methodologies. The whole system was divided into several objects, an objects communicate through the communication channel. This example shows the object-oriented characteristics of OO-CPN.

Encapsulation In the design above, when a new object is created, only the public transition can be accessed by another object. Private transition cannot be accessed by a part of system that exists outside the object. Usually, the public parts of an object are used to provide the functions of the object; the detail flow control is defined as private part. This mechanism provide the information hiding which makes it easy to change the behaviour of the object without modifying the whole system structure.

Inheritance Inheritance enables us to extend or change the already existing "parent" class, and make a "child class" that stays linked with its parent class. This is the key for reusability. In the above example, inheritance provides the way to model the system in a hierarchical structure. This makes whole system manageable and reusable.

Polymorphism In the above example, one use of polymorphism is to allow the parent class "thread" to specify the general behaviour of the threads, and the

child class of "thread" can dynamically have more specific behaviour. This reduces the complexity of the system, and models the whole system in a clear, structured way.

In conclusion, OO-CPN combines the characteristics of both object-oriented design and Petri Nets methodology. The object-oriented features provide the way to model a system in a structured and manageable way. The Petri Nets methodology provides a graphical design that is easy to understand.

Chapter 7

Conclusions and Future Work

The work presented in the preceding sections provides a way to model and specify a system using Petri Nets in an object-oriented approach. This chapter concludes the thesis and suggests future work.

Section 7.1 examines the contribution of this new Petri Nets model. Section 7.2 gives some suggestions for future work.

7.1 Contribution

The OO-CPN introduced in the preceding sections allows the modelling complex systems and data flows. Its most distinctive feature is the object-oriented structural design combined with Coloured Petri Nets formalism. CP-nets provide graphical representation with well-defined semantics. Graphical representation describes the sys-

tem in a very vivid and straightforward way. This makes CP-nets easy to understand. The well-defined semantics unambiguously define the behaviour of each CP-net. They make it possible to implement simulations for CP-nets, and also form the foundation for formal analysis methods. However, Petri Nets lack thorough modularization techniques. The result of any Petri Net model is one overall net. During the execution of the modelled software system, the Petri Net cannot copy with the addition of new, previously unknown models, and a minor change may lead to modification of the whole system. This is a big obstacle for their practical use in advanced distributed and network software systems. The object-oriented mechanism can make up for this defect. The encapsulation characteristic of object-oriented technique isolates the behaviours that belong to different categories. This enables changes in requirements to be accommodated without affecting the entire system. The inheritance characteristic enables reuseing parts that have been already created. Therefore, OO-CPN combines the characteristics of these two paradigms. It improves traditional CP-nets in the following ways.

7.1.1 Proposing a new hierarchy construct

Hierarchical Coloured Petri Nets include three hierarchy constructs: substitution places, invocation transitions, and fusion transitions. These constrcts enable the construction of a larger CP-net by combining a number of smaller nets. This improves

the compositionality of CP-nets. OO-CPN merges basic ideas behind these hierarchy constructs into a single new construct: object. The whole system is composed of several objects. These objects are independent of each other. This not only makes it possible to construct a large description from smaller units that can be investigated independently, but also allows the change of one object without affecting whole system design.

7.1.2 Defining the class diagram

The main building block of OO-CPN is the object. An object is an instance of a specific class. The class description can be divided into two parts:

Textual Expressions: Describing the structure of the class.

Class net: Showing the behaviour of objects that are instances of this class.

The relationship between two classes is inheritance. This not only provides the design parts reuse but also supports the hierarchical classification.

7.1.3 Proposing a new solution to the inheritance anomaly problem

CP-nets are suitable for modelling concurrent systems. But the combination of inheritance and concurrency sometimes leads to heavy encapsulation breakage. This

phenomenon is called an *inheritance anomaly.* Usually the problem is caused by a synchronization scheme that cannot be efficiently inherited without non-trivial class redefinitions. In OO-CPN there is one unique part defined for synchronization constraints, thereby separating the synchronization constraints from the action part. In this way, if there are any new synchronization constraints defined in the inheriting class, only this part needs to be changed. In addition, the action part is divided into two sections, which do not need to be changed simultaneously. Both of these mechanisms not only minimize redefinition but they also keep the behaviour of the object independent of further modification.

7.1.4 Defing the Polymorphism

OO-CPN supports two methods of polymorphism: overriding and overloading. By using overriding, we can redefine more proper characters and behaviours in sub-classes. Overloading enables us to define more specific behaviours in sub-classes. This mechanism is necessary in the hierarchical structure, since sub-classes usually behave differently from the super-class.

In conclusion, OO-CPN integrates object orientation into Coloured Petri Net formalism, thereby reaping the complementary benefits of these two paradigms.

7.2 Future Work

The work presented in this thesis provides the concepts and construction of the OO-CPN. The practical use of OO-CPN is highly dependent upon the existence of adequate computer tools. These tools may help their user in various ways:

Graphical Editor This helps the user to construct, modify and check the syntax of OO-CPN. It offers with a precision and quality that exceed the normal manual drawing capabilities of human beings.

Simulator The simulator traces the different occurrence sequences in an OO-CPN. Between each step, the user can see the enabled transitions and choose between them to investigate different occurrence sequences. It allows the user to make an interactive investigation of a complex occurrence graph using a special purpose search system.

Token Game Animation This automatically traces the flow of tokens in the OO-CPN.

Analysis Tool Includes Structural Analysis and Performance Analysis. It allows use to apply complicated analysis methods without having detailed knowledge of the underlying mathematics.

In conclusion, OO-CPN tools can make system construction faster and more accurate. Moreover, they can provide interactive presentations of analyses by hiding the

technical aspects of OO-CPN theory inside the tools. These tools are necessary for the real industrial applications. Therefore future work in this area would be significant in order to produce such tools.

Bibliography

[Bas95] Remi Bastide. Approaches in unifying petri nets and the object-oriented approach. page 6. 16th International Conference on Applications and Theory of Petri Nets, 1995.

[BCC01] E. Battiston, A. Chizzoni, and F. D. Cindio. CLOWN as a testbed for concurrent object-oriented concepts. *Lecture Notes in Computer Science: Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets*, 1:131–163, 2001.

[BD01] Oliver Biberstein, Didier Buchs and Nicolas Duelfi. Object-oriented nets with algebraic specifications: The CO-CPN/2 formalism. *Lecture Notes in Computer Science: Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets*, 2001:73–130, 2001.

[BFH+98] E. Best, W. Frączak, R.P. Hopkins, H. Klaudel, and E. Pelz. M-nets: an algebra of high level petri nets, with an application to the semantics of

concurrent programming languages. *Acta Informatica*, 35:813–857, 1998.

[C.A62] C.A.Petri. *Kommunikation mit Automaten.* PhD thesis, University of Bonn, 1962.

[CM01] Daniel Moldt Christoph Maier. Object coloured petri nets - a formal technique for object oriented modelling. In *Concurrent Object-Oriented Programming Petri Nets*, pages 406–427, University of Hamburg, Computer Science Department, 2001. Springer-Verlag Berlin Heidelberg.

[EM01] R.Devillers E.Best and M.Koutny. *Petri Net Algebra.* Springer Verlag, 2001.

[He01] Lin He. Object oriented concurrent system. Master's thesis, McMaster University, January 2001.

[Jen97] Kurt Jensen. *Coloured Petri Nets (Basic Concepts, Analysis Methods and Practical Use)*, volume 1. Springer, Februry 1997.

[JM96] Jens Beak Jorgensen and Kjeld Hoyer Mortensen. Modelling and analysis of distributed program execution in beta using coloured petri nets. In *Lecture Notes in Computer Science*, volume 1091, pages 249–268. 17th International Conference in Application and Theory of Petri Nets (ICATPN'96), Springer-Verlag, 1996.

[Lak01] Charles Lakos. Object oriented modelling with object petri nets. *Lecture Notes in Computer Science: Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets*, 2001:1–37, 2001.

[Lil01] Johan Lilius. An object based petri net programming notation. *Lecture Notes in Computer Science: Concurrent Object-Oriented Programming and Petri Nets, Advances in Petri Nets*, 1:247–275, 2001.

[MBMP93] O.L. Madsen and K. Nygaard B. Moller-Pedersen. *Object-Oriented Programming in the Beta Programming Language*. Addidon Wesley, 1993.

[Mes90] Jose Meseguer. A logical theory of concurrent objects. *ACM SIGPLAN Notices*, 25:101–115, 1990.

[MY93] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[Sch98] Herbert Schildt. *Teach Yourself C++*. Osborne/McGraw-Hill, third edition edition, 1998.

[Shi90] Etsuya Shibayama. Reuse of concurrent object descriptions. *Proceedings of TOOLS 3*, pages 254–266, 1990.