Modelling Object Oriented Non-Sequential Systems by Coloured Petri Nets

Lin He[1]
Deptartment of Computing and Software
McMaster University
Hamilton, Ontario
L8S 4K1
email: lin@mcmaster.ca, linhe@nortelnetworks.com

[1]Current affiliation: Nortel Networks, Kanata, Ontario K2J 4T9

Abstract

With the development of distributed and network system, we have seen the need of using Petri Nets to analyze and model complex systems which have multiple levels of activity. This motivates the definition of Petri Nets with multiple levels of activity in more compact form. In order to achieve this goal, we introduce the Object Oriented Coloured Petri Nets (OO-CPN) formalism, which combines the power of modularity of the object orientated paradigm and the capability of formally modeling concurrency in Petri Nets. OO-CPN is based on Hierarchical Coloured Petri Nets [10] and supports the concepts of object, class and inheritance.

In our OO-CPN, concurrency issues are declaratively abstracted by temporal constraints among events. Object behaviour and interaction are determined by precedence constraints among events. In class declaration, we separate the constraint part from the part of event-related places and transitions. When we declare a subclass, it can inherit all the same places and transitions of its superclass, if and only if it has the same corresponding precedence constraints as its superclass. This seperated class declaration minimizes the dependence between application functionality and concurrency control so that it allows software reuse to be more effective and using the Petri Net to model concurrecy issues to be more flexible and powerful.

In our OO-CPN, we emphasize the interaction between objects and classes, based on the client-sever protocol [11] and communication channel [12], as we believe in a real communication network, it is necessary and important to describe the individual service components as well as their interaction.

Chapter 1

Introduction

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

1.1 Background

Peti Nets, especially high-level ones (Coloured Petri Nets, CPN), are the widely used formalism for modeling concurrent systems in which processes have to communicate and synchronize in addition to performing a computation. Petri Nets have some interesting features, such as intuitive understanding, grahical representation as well as simplicity and formalization. Moreover, their executability makes them very suitable for simulation, rapid prototyping, and programming. Examples of some industrial applications can be found in [6, 7, 8, 9].

However, a disadvantage of Petri Nets is the absence of modularity. As we know, in general for a concurrent system, the state space grows exponentially in the number of components in the system and hence a simple algorithm that traverses the complete state space using an explicit representation for states and transitions will face the problem, *State Explosion*. High-level Petri Nets, Coloured Petri Nets, especially Hierarchial Coloured Petri Nets [10] made the significant contribution to this problem, which made Petri Nets a more power tool to describe the similarities and differences as well as the hierarchical relationship between the individual components in a large, complex concurrent system.

The object oriented paradigm provides a better method for writing programs in a well studied structured way and offers a simple but powerful

model for representing programs as computational entities which communicate with each other via message passing. In recent years, there has been considerable interest in combining the power of modularity in the object oriented approach with the benefit of formality in the Petri Nets formalism. Focusing on the different aspects, there are LOOPN++ in [14], OOB(PN)^{2}in [15], and PNtalk in [17].

LOOPN++ seeks to establish a formal framework that how Coloured Petri Nets can be enhanced to produce Object Petri Nets. It does so by defining a number of intermediate Petri Net formalisims and identifying the features introduced at each step of the development. Object Petri Nets support a intergration of object-oriented concepts into Petri Nets, including inheritance and the associated polymorphism and dynamic binding. The single class hierarchy readily supports multiple levels of activity in the net and the generation and removal of token that have been defined so that all subcomponents are simultanously generated or removed. Interaction between subnets can be either synchronous or asynchronous.

OOB(PN)^2 is an experimental parallel object-based high-level Petri Net programming notation. The semantics of OOB(PN)^2 is defined in terms of M-nets, a class of high-level nets equipped with CCS-like composition operators. The objects of OOB(PN)^2 are inherently parallel, meaning that each object may services method requests in parallel, and the variables are accessed with a mutual exclusion semantic. The mutual exclusion semantics is enforced by the resource consciousness of Petri Nets. The objects can communicate either synchronously or asynchronously. OOB(PN)^2 provides rudimentary synchronization constraints on the methods of the object, and inheritance.

PNtalk is a language and system based on object oriented Petri Net with Smalltalk inscriptions. In PNtalk, Object Oriented Petri Net is a set of classes partially ordered according to class inheritance. Every class consists of an object-net describing object's internal activity, and a set of method-nets describing object's repones on messages. All method-nets share access to the object-net(places of object-net are accessible by transitions of method-nets). Each method-net is reentrant and it has parameter-places and a return-place. Class inheritance is defined by inheritance of object-nets (nodes may be added, transitions together with surrounding arcs can be replaced) and inclusion of message sets(set of messages of a class is a subset of a set of messages of its arbitrary subclass). Tokens in nets are pointers to objects. Each object is an instance of some class and it consists of an instance of class' object-net. Each time the object receives a message, a new instance of corresponding method-net is created. In PNtalk, messages

sending or object creation is specified as an action attached to a transition.

Inheritance is one of the central mechanisms of object oriented programming. The main benefit of inheritance is the reduction of the implementational effort, since new classes can be constructed from the features of already existing classes. In particular, the growing importance of software reuse and maintenance has attracted a lot of interest in this approach.

In sequential object oriented programming, inheritance is widely used for automatical code reuse. But in recent years, when attempting to merge the object orientation paradigm with concurrent computation, researchers pointed out conflicts between inheritance and concurrency in object oriented languages [2]. This conflict is commonly referred to as *Inheritance Anomaly*. Several researchers have proposed solutions to this problem [13], but so far none of them has been commonly accepted.

1.2 Purpose

As a promising graphical description tool, Petri Nets have a formal semantics as well as an appropriate means to model concurrency. However, they also quickly yield a combinatorial explosion in the number of states and transitions. This phenomenon is commonly referred to as the *State Explosion* problem. This drawback of Petri Net makes it inapplicable to the large, complex system. High level Petri Nets were introduced to solve this problem. In this thesis, we intoduce Object Oriented Coloured Petri Nets (OO-CPN), which combines the power of modularity of the object-oriented paradigm and the capability of formally modeling concurrency in Petri Nets. OO-CPN is based on Hierarchical Coloured Petri Nets [10] and supports the concept of object, class and inheritance.

Coloured Petri Nets describe the system in a more compact form, in which tokens may be arbitrarily complex data values which move throughout the system as transition fire. But with the development of distributed and network system, we have seen the need of using Petri Nets to analyze and model complex systems which have multiple levels of activity. This motivates the definition of Petri Nets with multiple levels of activity, where tokens are no longer passive objects but are allowed to be subnets encapsulating their own activity and/or their own interaction strategy. Furthermore, code reuse is one of the major concerns in many software design. In order to achieve these goals, we introduce OO-CPN which supports the object-oriented principles and still keep the power of fomality of Petri Net.

In order to reduce the harmful effect of the Inheritance Anomaly, which

makes the necessity of reprogramming a high proportion of reused code when using inheritance [13], in our OO-CPN, concurrency issues are declaratively abstracted by temporal constraints among events. An object is an instance of a class net, which is the primary and vital unit when we model a system. An object net specifies an object's behaviour which is determined by precedence constraints among events, as well as the interaction between the objects which is achieved by message sending and receiving along a certain channel. In class declaration, we separate the constraint part from the part of event-related places and transitions. When we declare a subclass, it can inherit all the same places and transitions of its superclass, if and only if it has the same corresponding precedence constraints as its superclass. This seperated class declaration minimizes dependence between application functionality and concurrency control so that it allows software reuse to be more effective and the using Petri Net to model concurrecy issues to be more flexible and powerful.

In any concurrent system, one of the concern issues is how the different system components communicate with each other. In the OO-CPN, the interaction between objects and classes is based on the client-sever protocol [11] and communication channel [12].

Code reuse if one of the major concerns in many software design, in this thesis, we introduce the concept of component in the object oriented concurrent system. Moreover, physical and virtual or abstract componenets can be in a same model.

1.3 Outline

Chapter 2 is a survey of the major concepts in object orientation. One of the critical problems of the intergration of concurrency in an object orientated framework, the so called the *inheritance anomaly*, is also discussed in this chapter.

Chapter 3 briefly introduces concurrent system and Petri Nets, especially high-level ones, Coloured Petri Nets and Hierarchical Coloured Petri Nets

Chapter 4 plays the most important role in this thesis. In this chapter, we firstly present the concept of object in the OO-CPN as well as the precedence constraints in concurrent system. Secondly, we describe how to deal with the problem of *inheritance anomaly* in our OO-CPN and different communication features of the OO-CPN. Thirdly, we give an example of our approach by taking the dining philosophy problem. Finally, we formally definite the OO-CPN.

Chapter 5 discusses the contribution of this thesis, suggests future research in this area, and draws some conclusions.

Chapter 2

Object Orientation

This chapter presents a general definition of object-orientation. According to Wegner's taxonomy, a object orientation paradigm involves the combination of the three notions: object, class and inheritance" [1]. The problem of the inheritance anomaly is also be discussed in this chapter.

2.1 ADT, Class and Object

There are a wide variety of views of the object orientation paradigm, but the most fundamental concept is the *abstract data type (ADT).*

In real life, when we try to solve a problem, no matter what kind of problem it is, we should try to understand the problem in order to separate necessary from unnecessary details as we try to obtain our own abstract view or model of the problem. This process of modeling is called abstraction. The ADT in object-orientation defines an abstract view to the problem, no matter whether the problem is system design or program writing. ADT focuses only on problems related issues and the properties of the problem. These properties include

- the data which are affected and
- the operations which are identified by the problem.

In object orientation an ADT is referred to as a *class.* The definition of a class is the implementation of an ADT. It defines *attributes* and *methods* which implement the data structure and operations of the ADT, respectively.

An *object* is an instance of a class. It has a unique identity together with an evolving internal state. The internal state of an object can be

manipulated from the outside by a set of operations (methods). Moreover, we refer to these possible sequences of state changes as the behaviour of the object. In object oriented analysis and design, an object is any thing, real or abstract, about which we store data and have operations that manipulate the data. An object may be composed of other objects. These objects, in turn, may be composed of objects.

2.2 Encapsulation

The principle of packaging data and operations together is called *encapsulation.* Therefore encapsulation conveys the essential notion of the existence of objects, which hides its data from other objects and allows the data to be accessed only via its own operations. The user of the object sees only the services that are available from the object, but not how those services are implemented. Only the supplier has visibility into the object's procedures and its data.

If all programs could access the data in any way users wished, the data could easily be corrupted or misused. Encapsulation protects the object's data from arbitrary and unintended use.

2.3 Inheritance

Inheritance is a central mechanism of object oriented programming. The main benefit of inheritance is a reduction of the implementational effort, since new classes can be constructed from the features of pre-existing classes. In particular, the growing importance of software reuse and maintenance has attracted a lot of interest on this approach.

In sequential object oriented programming, inheritance is widely used to automatic code reuse. But in recent years, when attempting to merge the object orientation paradigm with concurrent computation, several researchers have discovered a conflict between inheritance and concurrency in object oriented languages. This conflict is commonly referred to as the *inheritance anomaly*[2].

Concurrency implies the need for *synchronization*, without which the state of the concurrent objects may become inconsistent: when a concurrent object is in a certain state, it can accept only a subset of its entire set of messages in order to maintain its internal integrity. We call such a restriction on acceptable messages the *synchronization constraint* of a concurrent object. For example, consider a bounded buffer with two methods *put*() and

remove(), where *put*() stores an item in the buffer and *remove*() removes an item from the buffer; here, the synchronization constraint is that one cannot *remove*() from a buffer whose state is *empty* and cannot *put*() into a buffer whose state is *full*.

In most objected-oriented concurrent programming languages, the programmer explicitly programs the *synchronization code* (the portion of the method code) to control the objects behavior so that it will satisfy the synchronization constraint. The synchronization code must always be consistent with the synchronization constraints of an object; otherwise, the object might accept a message that really should not be accepted, resulting in a sematic error during program execution.

Unfortunately, it has been pointed out that *synchronization code cannot be effectively inherited without a non-trivial class redefinition*[2]. For example, in the class *BUFFER* which has the methods *remove*() and *put*(), we may express the synchronization through constraints such as:

empty: {put};
partial: {put, remove};
full: {remove};

Assume that you want a descendant *NEW_BUFFER* to provide an extra method *remove_two*() which removes two buffer items at a time (provided the buffer size is at least two). Then you need an almost completely new set of states:

empty: {put};
partial_one: {put, remove};
partial_two_or_more:{put, remove, remove_two};
full: {remove, remove_two};

If the routines specify what states they proceed in each possible case, they must all be redefined from *BUFFER* to *NEW_BUFFER*, defeating the purpose of inheritance.

In [13], the authors identified three situations where the benefits of inheritance are lost:

1. Definition of a new subclass K' of class K necessitates re-definitions of methods in K as well as those in its ancestor classes.

2. Modification of a new method m of class K within the inheritance hierarchy incurs modification of the seemingly unrelated methods in both parent and descendent classes of K.

3. Definition of a method m might force the other methods (including those to be defined as subclasses in the future) to follow a specific

protocol which would not have been required had that method not existed.

There is considerable debate in the object-oriented community about this problem as well as the appropriate properties to be satisfied when one class inherits from another. Wegner outlines a number of possibilites [3]. The basic options are characterised by *subtyping* and *subclassing*. In the first scheme, we say that objects belonging to a type, or having the same type, share the same external behavior. The definition of a type depends on the precise meaning of *externally observable behavior*. Hence, any instance belonging to the type can produce the common behavior. In other words, any instance in a subtype can produce the behavior common to its supertype, thus it can be used in place of an instance belonging to the supertype.

In the second scheme, class incorporates both the module and type concept. On the one hand, a class is a *module* in the sense that it is an autonomous entity offering services to the outside. On the other hand, a class is a *type* in the sense that it describes a set of instances, or objects which encapsulate some data.

From this view, inheritance is a mechanism which relies on the module point of view of classes. It is used to express module refinements. An heir class owns all the features of its ancestor to which it may add new functions. Subtyping by contrast mainly relies on the type point of view of a class. It is used to express data refinements. A subclass specializes the set of objects of its superclass. Subtyping represents what is often called the *"is-a"* relation, like "every dog is a mammal" and "every mammal is an animal".

From our own understanding and experience, the differentiation of subtyping and inheritance in some cases is significant. But the practice of software design and implementation, the notation of subtype should be compatible with the notation of subclass so that it can support code reuse. In fact, in C++ and Java, when we see the word "subclass", we think "subtype" and vise versa.

Based on the scheme of differentiating type and class several researchers have proposed solutions to the problem of the inheritance anomaly [13]. However, after presenting a formal analysis of the inheritance anomaly and comparing the various proposals, three authors (Crnogorac, Rao and Ramamohanarao) [4] showed that an ideal solution for the version of the anomaly that has been investigated in the concurrent object oriented programming literature does not exist. As a result it becomes clear that the problem of the inheritance anomaly has not been solved. Furthermore, McHale [5] has shown that:

"The inheritance of synchronization code vs. the inheritance of sequential code conflict is *instrict* to languages that contain both sequential code and synchronization code and hence the conflict cannot be "cured"(solved). However, this does not preclude the possibility of *controlling* the conflict to reduce its harmful effects."

As a way of controlling the harmful effects, two mechanisms are proposed:

1. synchronization code and sequential code are seperated from each other, and

2. a means is provided for a subclass to incrementally modify inherited synchronization code.

In this thesis, we support the first mechanism. In Chapter 4, we explain this in more detail.

Chapter 3

Concurrency and Petri Nets

In this chapter, we briefly introduce concurrent systems, precedence constraints in concurrent systems, Petri Nets, Coloured Petri Nets and Hierarchical Coloured Petri Nets.

3.1 Concurrent Systems

Concurrent systems is a generic name for systems consisting of a number of components executing simultaneously while interacting with each other as well as with their environment. Examples of concurrent systems are protocols in telecommunications, embedded systems such as those appearing in modern electronic equipment or control systems such as flight control systems.

When modeling and verifying concurrent systems, the emphasis is on describing the control aspects and communication capabilities while the pure computational aspects are often abstracted away from. For a set of component programs $P_1, \ldots, P_n$ we let the concurrent system obtained by their composition be denoted by:

$(P_1 | \ldots | P_n)$

where $'|'$ is a parallel operator that embodies some synchronization mechanism between the $P_i's$ ranging from the one extreme of pure interleaving to that of global synchronization. Each component P_i describes how it is supposed to interact with the remaining components P_j, $j \neq i$. The global behaviour of a concurrent system is then understood as the joint behaviour of the involved component. In this thesis we emphasize on how to use a formal method, Object Oriented Coloured Peti Nets (OO-CPN) to analyze

and model complex concurrent systems.

3.2 Concurrency and Precedence Constraints

Concurrent systems are composed of different components, called *processes*, that act in parallel and interact with each other. Each process can be viewed as a reactive system. The behaviour of a reactive system is defined by its ongoing behaviour over time. In this thesis, how to describe timing dependence between processes is based on classic first order logic. The process is explicitly described as partially ordered sets of events.

The event ordering relation $X < Y$ denotes "X precedes Y".

A relation $< \subseteq X \times X$ is called a partial order if the relation $<$ is asymmetric and transitive, i.e.,

$a < b \Longrightarrow \neg(b < a)$

$a < b < c \Longrightarrow a < c$

We write $a \sim b$ if

$\neg(a < b) \wedge \neg(b < a) \wedge (a \neq b)$

This means a and b are distinct incomparable elements on the domain X.

Here, events are atomic, which are excuted in the specified order. For example, the expression

$think < hungry < eat < think+$

defines a philosopher who goes through the state $"think"$, $"hungry"$, $"eat"$, and loops these three states. Here $"+"$ means the offsprings of an event. The offsprings of E are named $E + 1$, $E + 2$, and are implicitly preceded by E, i.e. $E < E + n$, for all n.

The reason that we introduce precedence constraints here is that in chapter 4, we will use it to describe the object behaviour and interaction in our Object Oriented Coloured Petri Nets.

3.3 Petri Nets

Petri Nets are the graphical and mathematical modeling tool for describing and studying systems that are characterized as being concurrent, asynchronous, distributed, and nondeterministic. It relies on analyzing the set of *states* through which a system or system component can go, and *transitions* between these states.

A Petri Net consists of *places, transitions*, and *arcs* that connect them. The *input arcs* connect places with transitions, while the *output arcs* start

at a transition and end at a place. Places can contain *tokens*; the concurrent state of the modeled system (*the marking*) is given by the number (and type if the tokens are distinguishable) of tokens in each place. Transitions are active components. They model activities which can occur (the transition *fires*), thus changing the state of the system. The transitions are only allowed to fire if they are *enabled*, which means that all the preconditions for the activity must be fulfilled (there are enough tokens available in the input places). When the transition fires, it removes tokens from its input places and adds some at all of its output places. The number of tokens removed/added depends on the cardinality of each arc.

If we are to think in terms of a factory, tokens are equivalent to work parts. Arcs are the paths the work will follow through the factory. Places are buffers where parts are stored temporatily, and transitions are equivalent to machine where the parts are used to make new parts.

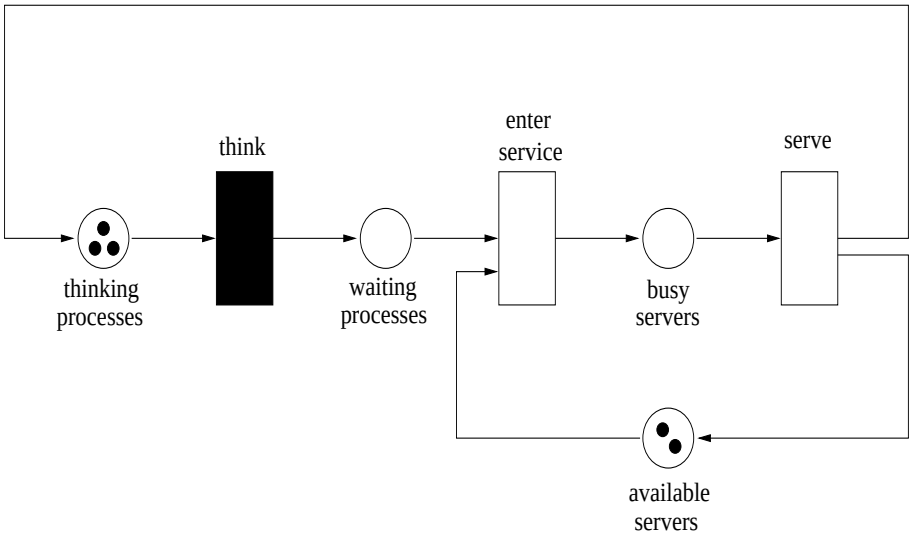


Figure 3.1: initial marking

The Petri Nets in figures 3.1 to 3.6 model a queueing system. The number of customers in the system is 3, while the number of the server is 2. In each net, the enabled transitions are shadowed. Figure 3.1 is the initial marking. In figure 3.2 (marking 1), the timed trainsition *think* is not fireable, because the immediate transition *enter service* has priority over any timed transition. In figure 3.3 (marking2), the transitions *think* and *serve* can be fired simultaneously. If transition *think* is fired, the system

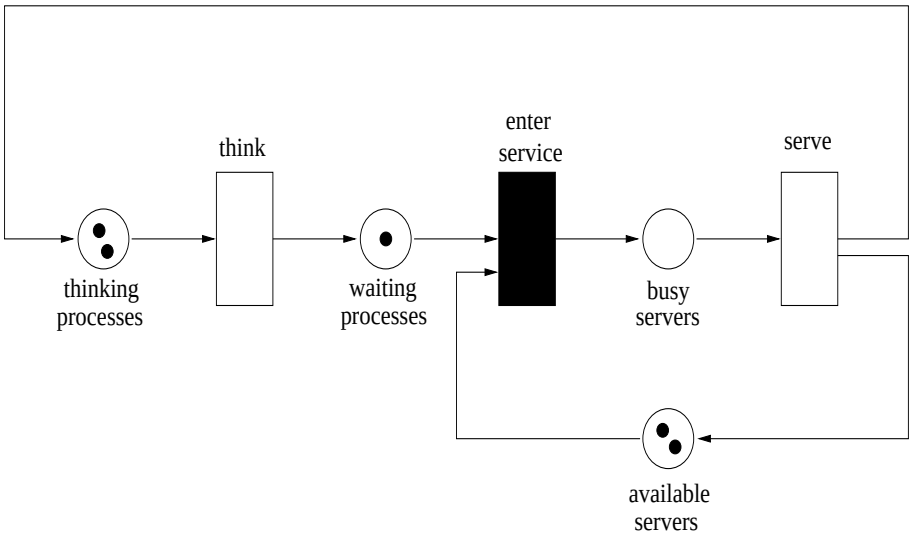


Figure 3.2: marking 1

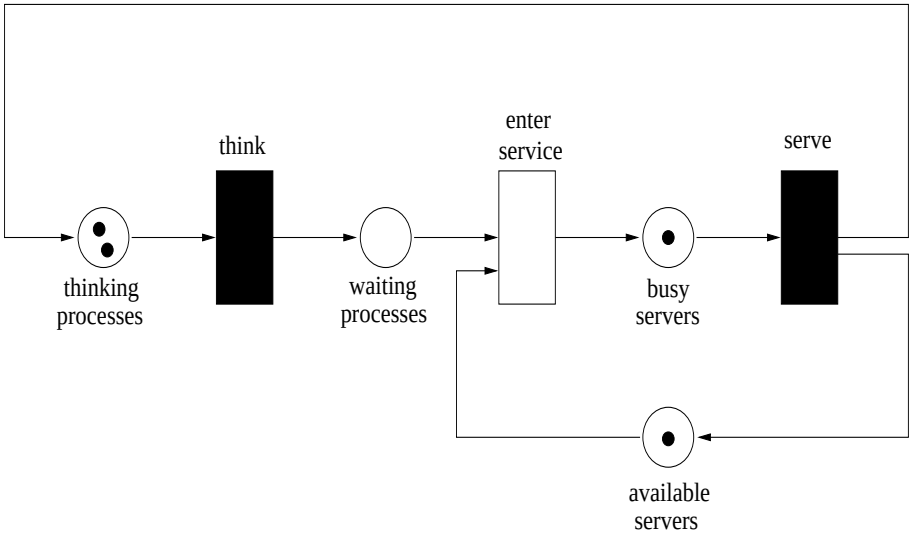


Figure 3.3: marking 2

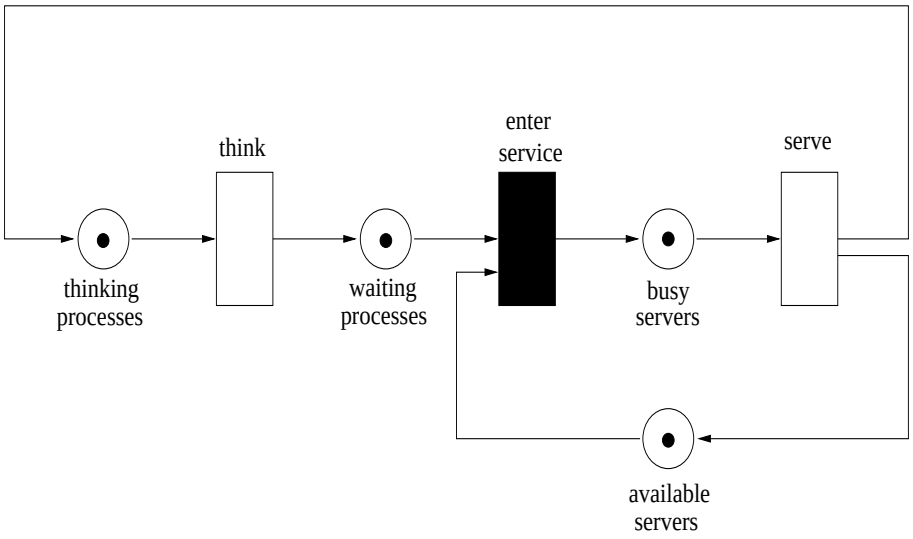


Figure 3.4: marking 3

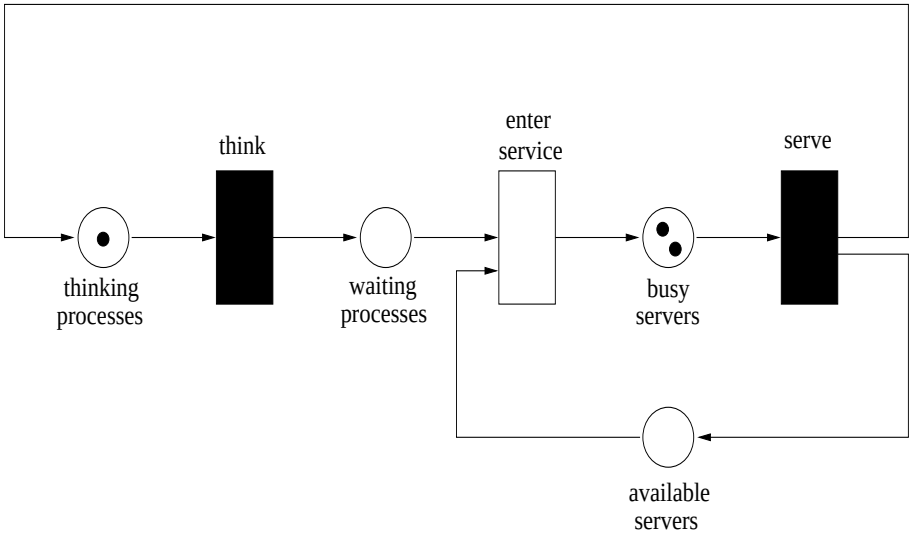


Figure 3.5: marking 4

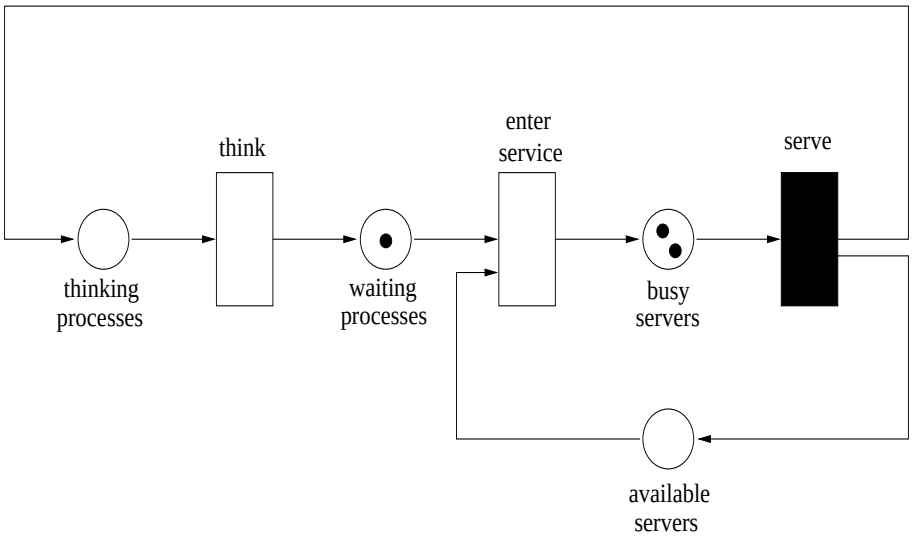


Figure 3.6: marking 5

will go to marking 3 (figure 3.4), while if transition *serve* is fired, the system will go back to initial marking. After transition *enter service* firing, the net will be updated to the marking 4 (figure 3.5). In the marking 4, the transitions *think* and *serve* can be fired simultaneously again. In this time, if transition *think* is fired, the system will go to marking 5 (figure 3.6), while transition *serve* is fired, the system will go back to marking 2 (figure 3.3). After marking 5, the system will go back to marking 3.

Although as a promising graphical description tool, Petri Nets have a formal semantics as well as an appropriate means to model concurrency. However, they also quickly yield a combinatorial explosion in the number of states and transitions. This drawback of Petri Nets makes them inapplicable to large complex systems.

3.4 Coloured Petri Nets

As we said, the state explosion is a major drawback of general Petri Nets, and such has clearly demonstrated a need for more powerful net types to describe complex systems in a manageable way. The development of Coloured Petri Nets (CPN) constitutes a very significant improvement in this respect. Here we present some fundamental concepts of CPN as defined by Jensen [10]

in the interest of demonstrating the continuity with the later definition of Object Oriented Coloured Peti Nets (OO-CPN).

CPN is a generation of Petri Nets, which describes the system in a more compact form by stressing the similarities and differences between two or more parts of a system. In CPN each token is attached a colour, indicating the identity of the token. Moreover, each place and each transition is attached a set of colours. A transition can fire with respect to each of its colours. By firing a transition, tokens are removed and added at the pre-condition and post-condition in the normal way.

3.4.1 Informal Introduction to Coloured Petri Nets

As an example, consider the standard synchronization problem consisting of five philosophers who alternatively think and eat. To eat, a philosopher needs two forks, but unfortunately there are only five forks on the circular table and each philosopher is only allowed to use the two forks nearest to him. Obviously two neighbors cannot eat at the same time.

The philosopher system can be described by a general Petri Net. Its graphical representation is show in Figure 3.7 (th, e, and ff are short for $think$, eat, and $freeforks$, respectively. $a_1, a_2, \ldots, a_5$ represent different philosopher $takeforks$, while $b_1, b_2, \ldots, b_5$ represent different philosopher $putdownforks$).

From Figure 3.7, we can see how large the net is. In practice, the system will be much larger than the philosopher problem, and if we work like this, we can not handle the complexity of the system. Now let us describe the same system by Coloured Peti Nets.

First we replace the five places $th_1, th_2, \ldots, th_5$ by a single place $think$, which can carry up to five tokens. To distinguish these tokens, we attach to $think$ a set of colours $PH = \{ph_1, ph_2, \ldots, ph5\}$. Analogously, the place $e_1, e_2, \ldots, e_5$ are replaced by a single place eat with PH as the set of possible colours, and the place $ff_1, ff_2, \ldots, ff_5$ are replaced by a single place $freeforks$ with $F = \{f_1, f_2, \ldots, f_5\}$ as the set of possible colours. We then get the Figure 3.8, where we use $i \bigoplus 1$ as shorthand for $(i \, mod \, 5)+1$.

In the next step we replace the five transitions $a_1, a_2, \ldots, a_5$ by a single transition $take\,forks$ which may fire in five different ways corresponding to the five philosopheres. To distinguish between the different ways of firing, we attach to the transition $take\,forks$ the set of colour, PH, representing the individual philosopher. We then get figure 3.9, where ID, $LEFT$ and $RIGHT$ are functions from the set of colour PH attached to $take\,forks$ into the sets of colours attached to its conditions: $think$, eat and $free\,forks$.

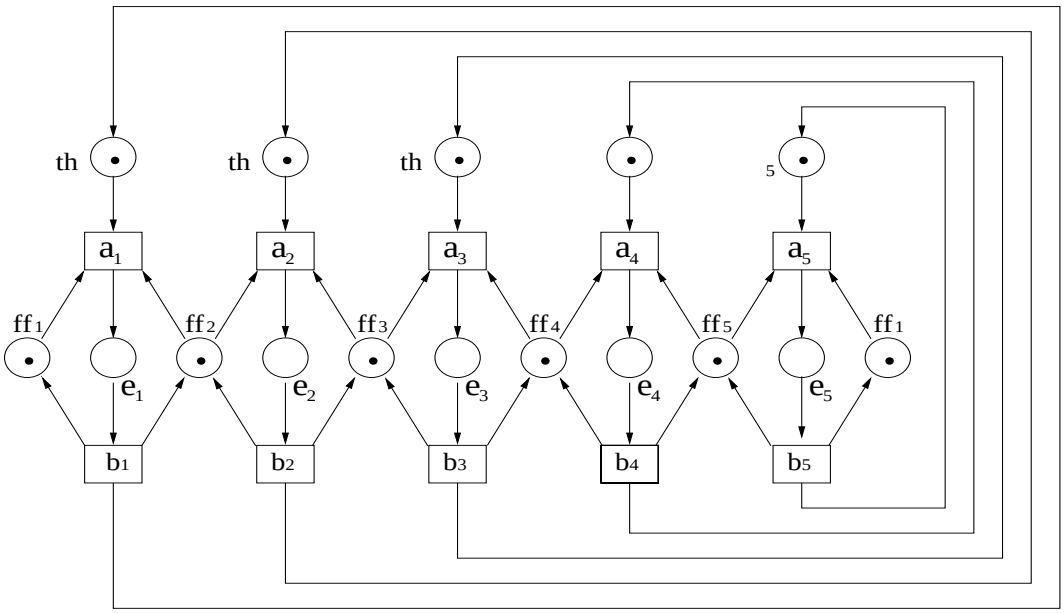


Figure 3.7: General Petri Net describing the philosopher system

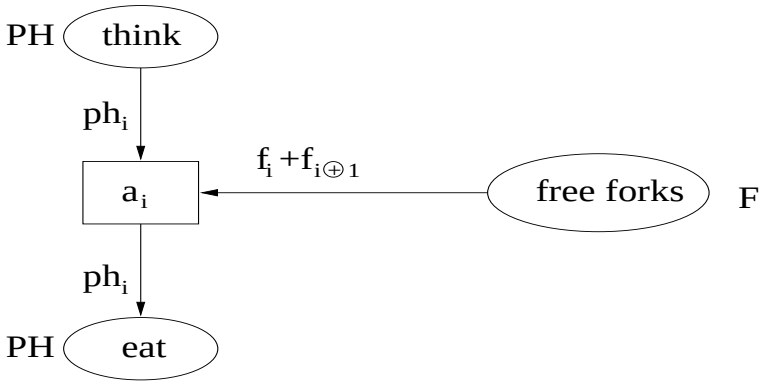


Figure 3.8: Part of the philosopher net after a folding where some places are unified.

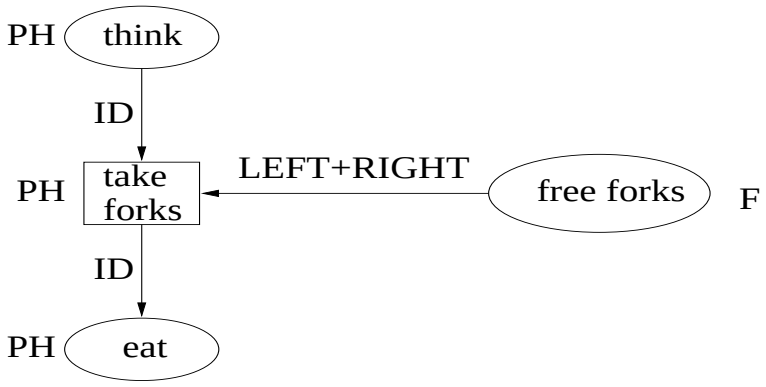


Figure 3.9: Part of the philosopher net after a folding where some places and some transitions are unified.

Analogously we replace the transitions b_1, b_2, ..., b_5 by a single transition *putdown forks* with PH as the set of possible firing colour. Now we get the Coloured Petri Nets in Figure. 3.10, which consists of three parts: the net structure (i.e., the places, transitions and arcs), the declarations and the net inscriptions (i.e., the various text strings which are attached to the elements of the net structure).

3.4.2 Formal Definition of Coloured Petri Nets

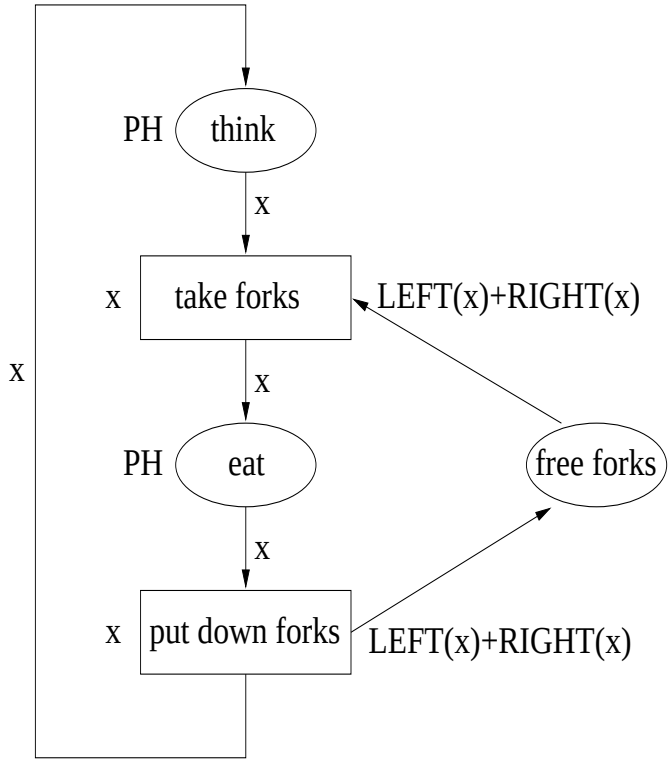
This section contains the formal definition of (non-hierarchical) CPN and their behaviour. We use Jensen's definition [10] for a multi-set, one of the fundamental concepts in Coloured Petri Nets. When we describe an entire CPN, we combine Jensen's definition with Lakos' definition [16] in the interest of demonstrating the consistency with the later formalism.

Definition 3.1: A **multi-set** m, over a non-empty set S, is a function $m \in [S \rightarrow N]$, which we represent as a formal sum:

$$\sum_{s \in S} m(s)`s$$

By S_{MS} we denote the set of all multi-sets over S. The non-negative intergers $\{m(s)|s \in S\}$ are the **coefficients** of the multiset. Note that $s \in m$ iff $m(s) \neq 0$. Here N denotes the set of all non-negative intergers and iff means *if and only if*.

Notes:



Colour PH=with ph_1 | ph_2 | ph_3 | ph_4 | ph_5

Colour FORK=with f_1 | f_2 | f_3 | f_4 | f_5

Var x: PH

fun RIGHT(x)=case x of ph_1 => f_2 | ph_2 => f_3 | ph_3 => f_4 | ph_4 => f_5 | ph_5 => f_1

fun RIGHT(x)=case x of ph_1 => f_1 | ph_2 => f_2 | ph_3 => f_3 | ph_4 => f_4 | ph_5 => f_5

Figure 3.10: Coloured Petri net describing the philosopher system

We use ϕ to denote the empty multi-set. There is one-to-one correspondence between sets over S and those multi-sets for which all coefficients are zero or one. Thus we shall, without any futher comments, use all $A \subseteq S$ to denote the multi-set which consists of at least one appearance of each element in A. Analogously, we use an element $s \in S$ to denote the multiset $1`s$.

Definition 3.2: Addition, scalar multiplication, comparison, and **size** of multi-sets are defined in the following way, for all $m, m_1, m_2 \in S_{MS}$ and all $n \in N$:

(a)

$$m_1 + m_2 = \sum_{s \in S} (m_1(s) + m_2(s))`s$$

(b)

$$n * m = \sum_{s \in S} (n * m(s))`s$$

(c) $m_1 \neq m_2 = \exists s \in S : m_1(s) \neq m_2(s)$
$m_1 \leq m_2 = \forall s \in S : m_1(s) \leq m_2(s)$

(d)

$$|m| = \sum_{s \in S} m(s)$$

when $|m| = \infty$ we say that m is **infinite**. Otherwise m is **finite**.

when $m_1 \leq m_2$, we also define **subtraction**:

(e)

$$m_2 - m_1 = \sum_{s \in S} (m_2(s) - m_1(s))`s$$

The following are assumed to be well-defined:

(a) the **set of elements of type** T, which can be denoted by T.

(b) the **type of a variable** v, denoted $Type(v)$.

(c) the **type of an expression** $expr$, denoted $Type(expr)$.

(d) the **set of variables in expression** $expr$, denoted $Var(expr)$.

(e) A **binding,** b**, of a set of variables** V, where $\forall v \in V : b(v) \in Type(v)$.

(f) the **value obtained by evaluation the expression** $expr$ **in binding** b,

denoted $expr < b >$. ($Var(expr)$ must be a subset of thevariables of b.)

Definition 3.3: A Coloured Petri Net is a tuple CPN $= (\Sigma, P, T, A, C, G, E, I)$ where:

(a) Σ is a finite set of non-empty types, called **colour set**.

(b) P is a finite set of **places**.
(c) T is a finite set of **transitions** with $P \cap T = \phi$
(d) A is a finite set of **arcs** such that $A \subseteq (P \times T) \cup (T \times P)$
(e) C is a **colour** function, $C : P \rightarrow \Sigma$
(f) G is a guard function, $G : T \rightarrow expr$
where $\forall t \in T : [Type(G(t)) = bool \wedge Type(Var(G(t))) \subseteq \Sigma]$.
(g) E is an **arc expression** function, It is defined from A into expressions such that:
$\forall a \in A : [Type(E(a)) = C(p(a)) \wedge Type(Var(E(a))) \subseteq \Sigma]$
where $p(a)$ is the place of arc a.
(h) I is an **initialization** function, $I : P \rightarrow expr$
where $I(p)$ is a closed expression (i.e. it contains no free variables)
and $\forall p \in P : [Type(I(p)) = C(p)]$.
Notes:

- The set of types determines the data values and the operations and functions that can be in the net expressions (i.e., arc expressions, guards and initializations expressions). We assume that each type has at least one element.

- Places serve to hold tokens, with the distribution of tokens determining the state of the net.

- Transitions serve to effect changes of state, and hence are distinct from places.

- Arcs have been restricted so that each transition has at most one input and one output arc to each place. This has been done merely to reduce the number of definitions and thereby simplify the presentation.

- The colour or type function C gives the (multiset) type of values associate with each place. This means that each token in p must have a data value that belongs to $C(p)$.

- The guard associates a boolean expression with each transition, which must evaluate to *true* for the transition to be enabled.

- The arc expression is only non-empty for arcs and its expression must have a type matching the type of the place to which the arc is connected.

- The initialization of a data field must specify an appropriately typed value.

Definition 3.4: A **token element** is a pair (p, c) where $p \in P$, $c \in C(p)$. The set of all token elements is denoted by TE.

A **marking** M is a multiset over TE. A marking M can interchargeably be treated as a multiset and as a fuction on P with $\forall p \in P : M(p) \in C(p)$. The **initial markng** M_0 is the marking obtained by evaluating the initialization expressins, i.e $\forall p \in P : M_0(p) = I(p)$. The set of all marking is denoted by M.

Definition 3.5: A **binding b** of transition t is a binding of the variables $Var(t)$ satisfying $G(t) < b >$. $B(t)$ is used to denote the set of all bindings for t. A **binding element** is a pair (t, b) where $t \in T$ and $b \in B(t)$. The set of all binding elements is denoted by BE. A step Y is a non-empty, finite multi-set over BE. A **step** is a non-empty and finite multiset over BE.

Definition 3.6: The multiset of tokens removed from (added to) places by step Y is denoted $get_Y(put_Y)$, and is defined as:

$$get_Y = \forall p \in P : \sum_{(t,b) \in Y} E(p,t) < b >$$

$$put_Y = \forall p \in P : \sum_{(t,b) \in Y} E(t,p) < b >$$

Definition 3.7: A step Y is **enable** in a marking M of net CPN iff: $M(p) \geq get_Y$.

Definition 3.8: When a step Y is enabled in a marking M of net CPN, it may **occur**, changing the marking M to another marking M', defined by:

$\forall p \in P : M'(p) = (M(p) - get_Y) + put_Y$.

Notes:

The expression evaluation $E(t,p) < b >$ gives us the tokens, 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 multiset 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 Y.

3.5 Hierarchical Coloured Petri Nets

3.5.1 Informal Definition of Hierarchical Coloured Petri Nets

In order to cope with large systems we need to develop strong structuring and abstraction concepts that allow us to work with selected part of the

model without being distract by the low-level details of the remaining parts. This motivation has led to the development of Hierarchical Coloured Petri Nets (HCPN) [10]. The basic idea behind HCPN is to allow the modeller to construct a large model by using a number of small CPN which are related to each other in a well-defined way. This is similar to the situation in which a programmer constructs a large program by means of a set of modules. Many CPN models consist of more than one hundred individual CPNs with a total of many hundred places and transitions. Without hierarchical structuring facilities, such a model would have to be drawn as a single (very large) CPN, and it would become totally incomprehensible. In this section we will briefly present some major concepts in Hierarchical Coloured Petri Nets.

(1)Substitution of transitions

The intuitive idea behind substitution of transitions is to allow the user to relate a transition (and its surrounding arcs) to a more complex CPN, which usually gives a more precise and detailed description of the activity represented by the substituted transition. In HCPN we relate individual CPN to a node, that is a member of another CPNs, and this means that our description will contain a *set* of non-hierarchical CPNs, which we shall call **pages**. Meanwhile, in order to make it possible for human beings and computer systems to refer to the page, we give every page a *page name* or *page number*.

In the net hierarchy, each substitution transition is said to be a *supernode* (of the corresponding subpage) while the page of a substitution transition is a *superpage* (of the corresponding *subpage* which contains the detailed description of the activity modelled by the corresponding substitution transition. The basic idea behind substitution transitions is that it should be possible to translate a hierarchical CPN into a behaviourally equivalent non-hierarchical net by replacing each substitution node (and its surrounding arcs) with a copy of its subpage. However, to do this we need to know how the subpage should be glued together with the surrounding of the supernode (i.e., the rest of the superpage). This information is provided by the *port assignmet*, which describes the interface between the superpage and the subpage.

(2) Page instance

In order to model a large system containing a lot of pages, we need a way to specify the number of page instances a system has. The situation is

analogous to a program, where we also need to specify a main program. To specify the number of page instances we shall specify a set of *prime page instances* which are analogous to the main program, and a number of *secondary page instances*, which are analogous to the procedures.

(3) Fusion of places

The intuitive idea behind fusion places is to allow the user to specify that a set of places are considered to be identical, i.e., they all represent a single conceptual place even though they are drawn as individual places. This means that when a token is added/removed at one of the places, an identical token will be added/removed at all the others. The relationship between the members of a fusion set is, in some respects, similar to the relationship between two places which are assigned to each other by a port assignment.

The places that participate in such a *fusion set* may belong to a single page or to several different pages. When all members of a fusion set belong to a single page and that page only has one page instance, place fusion is nothing other than a drawing convenience that allows the user to avoid too many crossing arcs. However, things become much more interesting when the members of a fusion set belong to several different pages. In that case fusion sets allow the user to specify a behaviour which may be cumbersome to describe without fusion.

There are three different kinds of fusion sets: *global fusion sets* are allowed to have members from many different pages, while *page fusion sets* and *instance fusion sets* only have members from single page. The difference between the last two is the following: A page fusion unifies all the instances of its places (independently of the page instance at which the place instance appears), and this means that the fusion set only has one *resulting place* which is *shared* by all instances of the corresponding page. In contrast, an instance fusion set only identifies place instances that belong to the *same* page instance, and this means that the fusion set has a *resulting place* for each page instance.

Chapter 4

Object Oriented Coloured Petri Nets

This chapter plays the most important role in the thesis. In this chapter, we introduce the Object Oriented Coloured Petri Nets (OO-CPN) formalism, which try to solve both thelack of modularity in Petri Nets and the lack of formality in the object oriented paradigm. OO-CPN is based on Hierarchical Coloured Petri Nets and emphasizes on the simplicity and clarity of Object Oriented Coloured Petri Nets definition.

The organization of this chapter is as follows. Section 4.1 presents the concept of objects in the OO-CPN. Section 4.2 introduces communication in OO-CPN. Section 4.3 discusses how to deal with the problem of the inheritance anomaly in the OO-CPN. Section 4.4 describes the page hierarchy of OO-CPN. Section 4.5 gives an example of our approach by taking dining philosophers problem. Section 4.6 discusses the same problem from the component point of the view, Section 4.7 formally defines the OO-CPN. Section 4.8 is a brief summary.

4.1 Object and Petri Nets

The major structural unit of an OO-CPN is an *object*. In general, an object in a multi-level Petri Nets is an instance of a *class net*. Different instances of the same class are differentiated by *object identifiers*. The *marking* of a place in such a class net can be specified by the tokens resident in the place, plus the object identifier of the class net instance. Furthermore, if tokens can be object identifiers, they can represent nested class instances.

Therefore, in a similar manner to the concept of an object in object-oriented language, in the OO-CPN, an object is any thing, real or abstract, about which we store data and operations that manipulate the data. To execute an OO-CPN, instances of the class nets are created. These instances are called *object nets*. Transitions associated with a place provides the methods (or the services that a class can offer) which may operate on the data within the object. The net structure determines the ordering of these methods, hence defining the systems functionality. Object nets are potentially concurrent, and communicate by passing messages through a certain channel.

4.2 Communication in OO-CPN

In the real world, two units communicate and interacte with each others by sending messages. Each unit has a set of messages to which it will respond. Moreover, a successful communication can happene only when the message sender is ready to send and the message receiver is ready to receive. In other words, communication is processed by message request and message reply through a certain channel. Taking this nature into account, in this thesis, we will bring together the concept of channel and message request-reply mechanism.

In the OO-CPN, objects communicate through token exchange, which can be simply treated as message exchange. An event means the arrival of a message, namely, a certain kind of token. A transition is an action triggered by an event, which will move the object into a new place. A place represents a static situation in which an object finds itself.

We adopt the CSP ?! notation to describe message receiving and sending. The expression $c?v$ denotes that an object receives message v from a channel c; The expression $c!v$ denotes a object sends message v through a channel c. In the OO-CPN, if the event $c?v$ triggers a transition, we will call this transition $?v$- tansition; if the event $c!v$ triggers a certain transition, we will call this transition $!v$-tansition. A communication between two transitions is only possible if one of the transitions is a $?v$-tansition and the other is a $!v$-tansition.

4.3 Inheritance

One of the main advantage of the object oriented approach is inheritance. Objects are organised in a hierarchy by an object inherits properties from its ancesters. Inheritance increases code sharing by allowing the reuse code

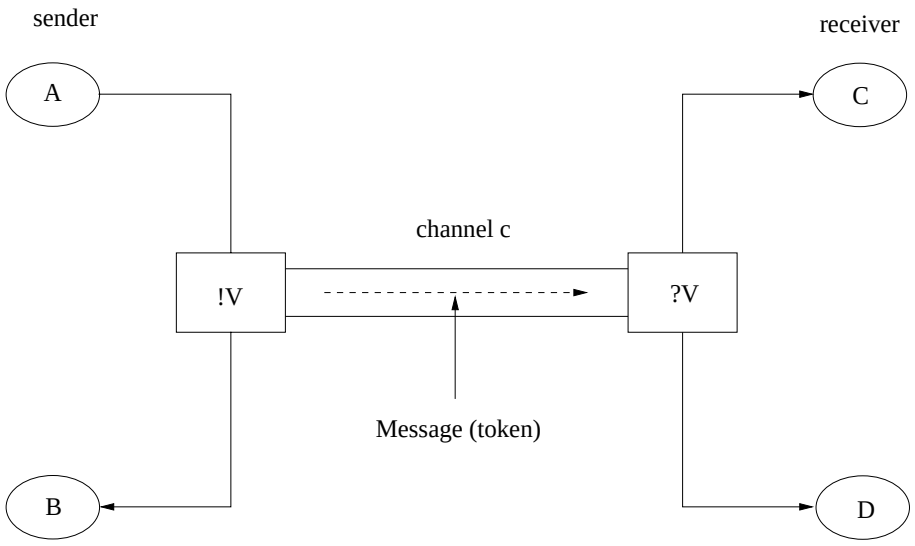


Figure 4.1: communication in OO-CPN

from one class in another class. In the OO-CPN a class inherits another class using an *inherit* declaration. A class either inherits a particular section of another class using the *partially inherit* declaration or inherits a complete class by declaring *globally inherit*. Therefore, inheritance of behaviour and inheritance of code are independently supported. A class may contain more than one inherit declaration, allowing multipe inheritance. In the next section, we will explain inheritance in more detail using dining philosopher problem.

4.4 Page Hierarchy of OO-CPN

The idea behind page hierarchy of OO-CPN is to reduce the apparent complexity of an OO-CPN by allowing the behavior to be subdivided into multiple levels. Formally, a hierarchical page is a page in which one or more transitions can contain its own internal OO-CPN. This allows the behavior of the OO-CPN to be viewed at different levels of abstraction, that is, complex detailed behavior can be lumped into a single stable transition at the higher level of abstraction. Page hierarchy can be used to encapsulate entire event handling sequences involving multiple places and transitions in the OO-CPN. The component pages of a hierarchical page are called *subpages*

(of the corresponding *superpage*). The highest level page in a OO-CPN is called *top page*, while a page which does not have further decomposition is called *leaf page*. In the example of the dining philosopher problem, we will present the page hierarchy in more detail.

4.5 An Informal Example

In order to present OO-CPN, in this section we have chosen the well known dining philosopher problem as an example. The context is still one of philosophers sitting around a table with one fork lying between two philosophers. The philosophers are in the thinking state and to move into the eating state, each of them needs to hold their left and right forks.

To express object oriented features, we have introduced three hierarchically organized kinds of philosophers. Philosophers at the top of the hierarchy belong to the Dijkstra philosopher club and we simply call them *Philosopher(PH)*. In this club there are two different trends which do not share the same idea, namely, *Philosopher_s(PH_s)* and *Philosopher_d(PH_d)*. In addition to the traditional behaivour, *Philosopher_s* have one more habit, sleeping for a while then thinking again. However *Philosopher_d* like drinking some wine after thinking, then go back to the traditional loop. The inheritance relationship in the philosopher club is illustrated in the Figure 4.2.

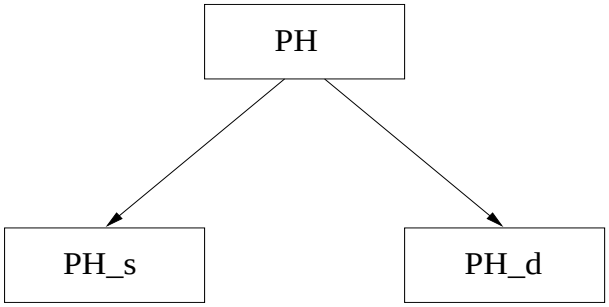


Figure 4.2: Inheritance relationship in the philosopher club

4.5.1 Structure of OO-CPN

In our OO-CPN, we model three kinds of philosophers in Dijkstra philosopher club as three class, class PH as superclass which has two subclass, class

PH_s and PH_d. In order to process their methods, all these three classes need to communicate with the class $Fork$ through a certain medium, class *channel*. The corresponding OO-CPN is illustrated as Figure 4.3 to 4.9.

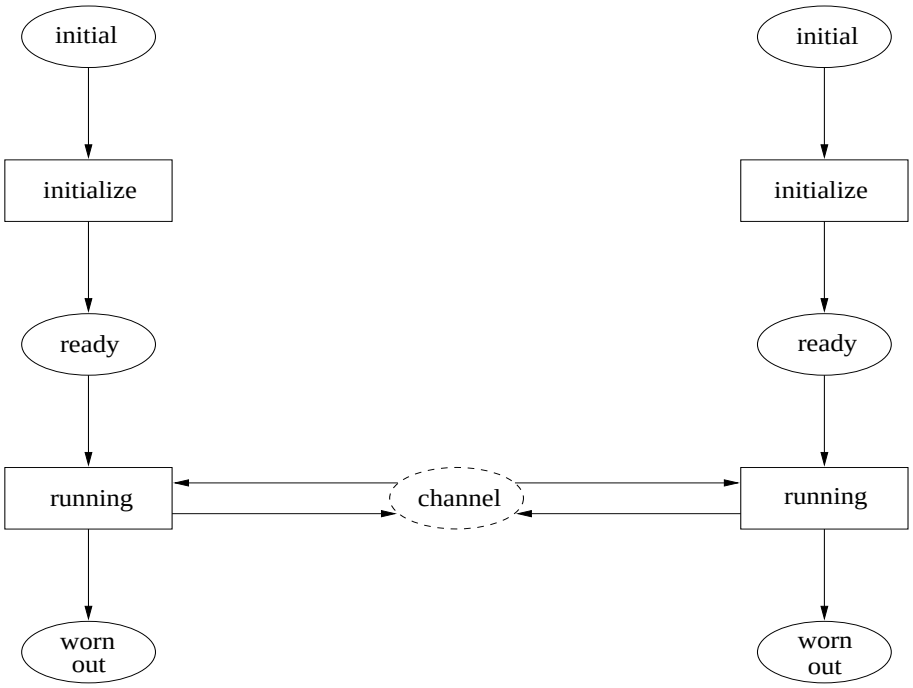


Figure 4.3: OO-CPN for dining philosopher system: top page

In this example, we only have two levels in the page hierarchy, the *top page* in Figure 4.3 and the subpage in Figure 4.4. The top page gives the overview of the OO-CPN for this problem: in the life time of philosophers and forks, they communicate with each other through a certain channel so that every philosopher can get the chance to pick up the two forks at the same time to eat and nobody will starve. The subpage describes the detail behavior of the three kinds of philosophers and the forks. In the figure of the subpage, we present the one superclass, PH and two subclasses, class PH_d, and class PH_s together. The solid arrows describe the tradition behaviour of class PH, while the dash arrows add some additional behaviour of the class PH_d and the class PH_s. The reason that we construct this system in this way is that firstly, in this example, our major concern is the interaction between the philosophers and the forks; secondly describing the behaviour

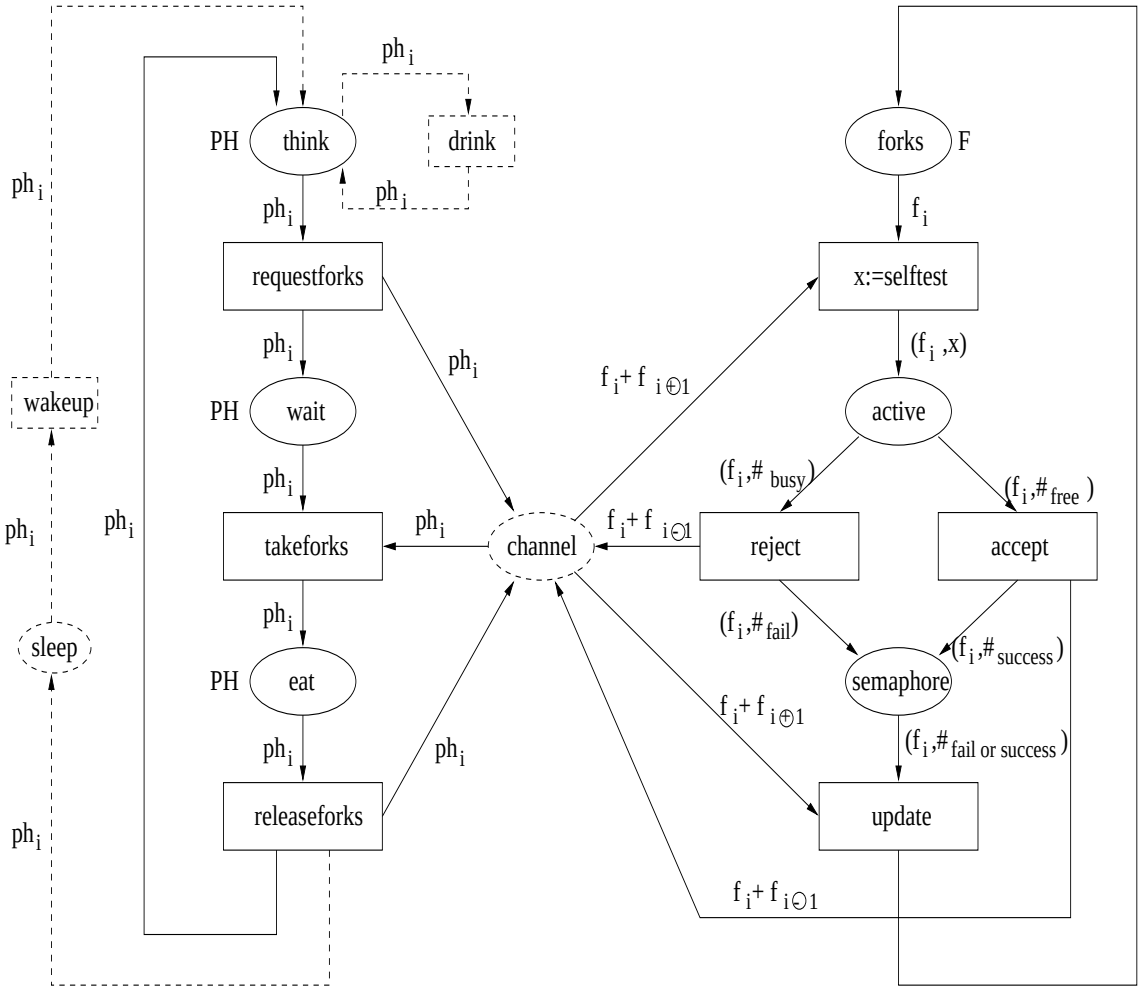


Figure 4.4: OO-CPN for philosopher system: running page

Class PH;
Initial
ph_i :=**new** $ph \longrightarrow think$;
Body
$ph_i.requestforks\,!\,f_i + f_{i\oplus 1}$ **is**
$think \longrightarrow wait$;
$ph_i.takeforks\,?\,f_i = f_{free}, f_{i\oplus 1} = f_{free}$ **is**
$wait \longrightarrow eat$;
$ph_i.releaseforks\,!\,f_i := f_{free}, f_{i\oplus 1} := f_{free}$ **is**
$eat \longrightarrow think$;
Constraints
$think < wait < eat < think+$;
Where
$i = 1, 2, 3, 4, 5$;
End PH;

Figure 4.5: Class PH

Class PH_s **partially inherit** class PH;
Initial
ph_s_i :=**new** $ph\_s \longrightarrow think$;
Body
$ph\_s_i.releaseforks\,!\,f_i := f_{free},\ f_{i\oplus 1} := f_{free}$ **is**
$eat \longrightarrow sleep$;
$ph\_s_i.wakeup$ **is**
$sleep \longrightarrow think$;
Constraints
$think < wait < eat < sleep < think+$;
where
$i = 1, 2, 3, 4, 5$;
End PH_s;

Figure 4.6: Class PH_s

Class PH_d **globally inherit** class PH;
Initial
$ph\_d_i :=$ **new** $ph\_d \longrightarrow think$;
Body
$ph\_d_i.drink$ **is**
$think \longrightarrow think$;
Constraints
$think < wait < eat < think+$;
Where
$i = 1, 2, 3, 4, 5$;
End PH_d;

Figure 4.7: Class PH_d

Class $Fork$;
Initial
$f_i :=$ **new** $fork \longrightarrow forks$;
Body
$f_i.selftest\,?\,f_i + f_{i\ominus 1}$ **is**
$forks \longrightarrow active$;
$f_i.reject\,!\,f_i = f_{busy}, f_{i\ominus 1} = f_{busy}$ **is**
$active \longrightarrow semaphore$;
$f_i.accept\,!\,f_i = f_{free}, f_{i\ominus 1} = f_{free}$ **is**
$active \longrightarrow semaphore$;
$f_i.update\,?\,f_i := f_{free}, f_{i\ominus 1} := f_{free}$ **is**
$semaphore \longrightarrow forks$;
Constraints
$forks < active < semaphore < forks+$;
Where
$i = 1, 2, 3, 4, 5$;
$x = int$;
End $Fork$;

Figure 4.8: Class $Fork$

Class *Channel*;
Initial *channel* **new** *channel*;
End *channel*;

Figure 4.9: Class *Channel*

of the different philosophers in the same figure, it is easier for us to tell the similarities and differences of the action loops of them.If we take the interaction between different philosophers into acount, we will need more levels of page hierarchy.

From the Figure 4.3 and 4.4, we can see that in the OO-CPN, the net structure (i.e, the places, transitions and arcs) is similar to the control structure of a Coloured Petri Net, while net inscriptions are used for the detailed class declaration.

The key word *Class* is used to declare every new class, while *Initial* completes new object creation. *Body* is the major part of the class declaration, which comprises transitions and places of a class. In the *Body*, we associate every event with corresponding action. For example, $requestforks$ changes the state of philosopher from *think* to *wait*. Moreover, in order to descibe communication between the two objects, we introduce CSP notaion $''?''$ and $''!''$. The expression $ph_i.requestforks\,!\,f_i + f_{i\oplus 1}$ means the object philosopher ph_i request forks, meanwhile he sends the message that forks f_i and $f_{i\oplus 1}$ are requested through the channel to the object forks. On the other hand, the expression $f_i.selftest\,f_i + f_{i\ominus 1}$ means that the object forks after receiving the message that f_i and $f_{i\ominus 1}$ are requested, the relevant forks selftest to check whether they are aviliable. Here we use the notation $=$ to express *equal* and $:=$ to expresses *assignment*. The part *Constraints* determines the object's behaviour precedence as well as the inheritance type. The subclass PH_d has completely the same constraints as that of the superclass PH, and therefore we declare it *globally inherit* class PH. By the contrast, because the subclass PH_s only has the part of same constraints as that of the superclass PH, we only declare it *partially inherit* class PH.

Class PH

This is the general philosopher class. Object philosopher is created by expression $ph_i := new\ \ ph$. As thinking is the philosopher's characteristic, initially, they start in the state *think* immediately. In order to eat, each philosopher needs to $requestforks$. If he is lucky, from the channel he gets

the information that the forks of his two sides, f_i and $f_{i\oplus 1}$ are all free, he can $takeforks$ and go into the state eat. After eating, the philosopher should $releaseforks$ through the same channel so that object forks can update their database and his neighbour philosopher can get the forks and eat.

Class PH_s and class PH_d

As we have known, class PH_s and class PH_d are both heirs of general philosopher class PH. From Figure 4.4, we can see philosophers ph_d has the identical behaviour to that of the general philosopher ph except they have one more habit, after thinking they like drinking some wine then thinking again. As so, from the class decalaration (Figure 4.7), we find class PH_d has the same constraints as that of the class PH, and therefore we declare it *globally inherit* class PH. Class PH_d can reuse all the *Body* part of class PH.

As a heir of general philosopher class PH, class PH_s is greatly different from class PH_d in respect to inheritance. From Figure 4.4, we can see that philosopher ph_s must sleep for a while after eating. *sleep*, this one more event makes the traditional philosopher's behaviour loop critically be changed so that the corresponding **Constraints** part in class declaration have to be partially modified (Figure 4.6); therefore we only declar class PH_s **partially inherit** class PH.

Class Fork

Class $Fork$ (Figure 4.8) has four transitions $selftest$, $reject$, $accepte$, and $update$. When forks receive the information that two of them f_i and $f_{i\ominus 1}$have been requested, they immediately $selftest$ these two forks. If the corresponding forks are $free$ (or $busy$), they accept (or reject) the request and send back the relevant information to the philosopher through the channel. Futhermore, the $success$ and $fail$ information must be returned to $update$. These two processes are synchronized by $semaphore$. Meanwhile, the place $update$ also receives the information from the philosopher through the channel to update the class' datebase.

Class channel

The simplest class of this example is channel. Since it does it do not have any behaviour of its own, and therefore could be represented by ADT. The class declaration is illustrated as Figure 4.9.

4.5.2 Componenet in OO-CPN

In the above section, we focused on the concept of interaction between objects and classes. Now, let us conside the same problem in the different way: the concept of component. In our example, Philosopher and Fork are two different classes which can communicate with each other through a certain channel. However, from the component point of view, we can also treat Philosopher and Fork as two software components. Both of them represent real-world entities, which model concrete entities that exist in the application domain. In addition to creat components that correspond directly to things in the real -world application domain, when we analyse and design a system, we can also creat some other software components which may represent non-physical but virtual or abstract entities such as: an algorithm or data structure (for example: a queue), or a piece of software that performs a special function, which is somehow analogous to the transition in the OO-CPN. Based on this idea, we can model the component view of dining philosopher problem in the Figure 4.10.

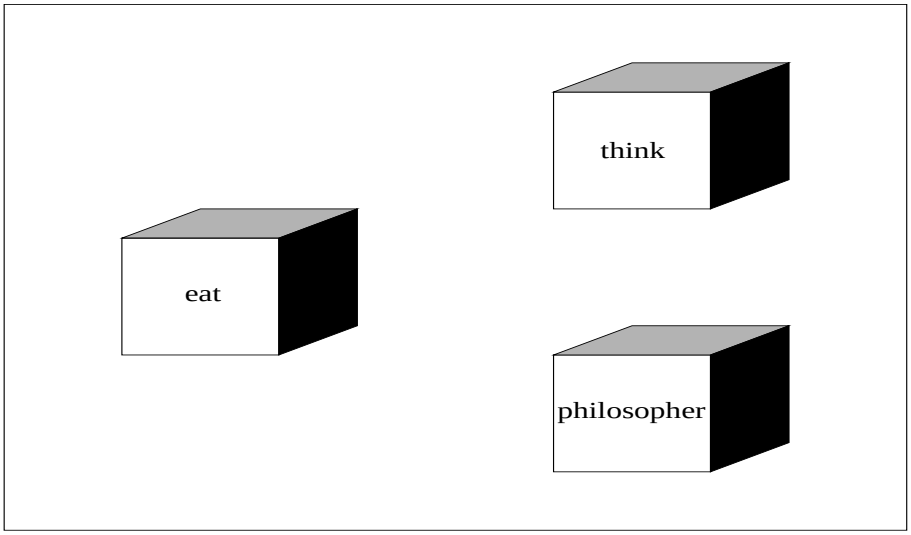


Figure 4.10: Component-based modeling dining philosopher problem

A philosopher is an real-world entity that can be modeled explicitly, while the feature eat is a virtual entity. As an example, in the Figure 4.11, *eat* is represented by a component, containing other three components: two real-world entities namely *hand* and *forks* as well as another virtual entity

takeforks.

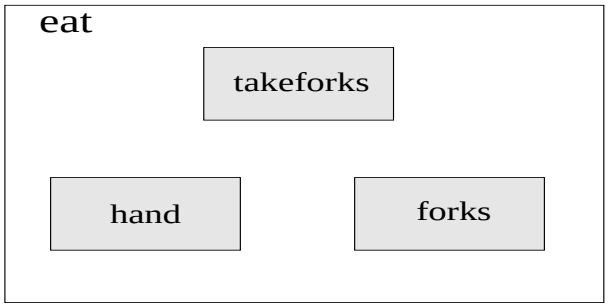


Figure 4.11: Component-based modeling virtual entity *eat*

Software components can be stored in libraries, so they can be reused in the different designs. This reduces the overall effort required to develop and maintain software.

4.6 Formal Definition of OO-CPN

Definition 4.1 An Object-oriented Petri Nets class c, in the context of a set of pages S, is a tuple (OID, Σ_s, K_s, P_s, PP_s, T_s, A_s, τ_s, σ_s, G_s, E_s, I_s) with:

(a) $OID = \{OID_{s'} \ | s' \in S\}$ where each $OID_{s'}$ is a set of distinct object identifiers associated with each page s' in S.
(b) Σ_s is a finite set of colour set with $OID \subseteq \Sigma_s$
where $\forall OID_{s'} \in OID : \forall C \in \Sigma_s : C = OID_{s'} \vee C \cap OID_{s'} = \phi$
(c) K_s is a finite set of constants
(d) P_s is a finite set of places with $K_s \cap P_s = \phi$
(e) PP_s is a finite set of port places $PP_s \subseteq P_s$ possibly with $self \in PP_s$
(f) T_s is a finite set of transitions with $(K_s \cup P_s) \cap T_s = \phi$
(g) A_s is a finite set of arcs such that $A_s \subseteq P_s \times T_s \cup T_s \times P_s$
(h) τ_s is a colour function, $\tau_s : K_s \cup P_s \cup T_s \longrightarrow \Sigma_s$
where $\forall k \in K_s : \tau(k) \neq \phi$
and $\forall p \in P_s : \tau(p) = C^* \vee \tau(p) = OID_{s'}$ for some s'
and $\forall t \in T_s : \tau(t) = \phi \vee \tau(t) = OID_{s'}$ for some s'
(i) σ_s is a token interface function, $\sigma_s : P_s \longrightarrow \Sigma_s$
where $\sigma_s(p) = C_{MS}$ for $p \in P_s$
(j) G_s is a guard function, $G_s : T_s \longrightarrow expr$

where $\forall t \in T : [Type(G_s(t)) = bool \wedge Type(Var(G_s(t))) \subseteq \Sigma_s]$
(k) E_s is an arc expression function, $E_s : P_s \times T_s \cup \times T_s \times P_s \longrightarrow expr$
$\forall a \in A_s : [Type(E_s(a)) \leq \sigma_s(p(a)) \wedge Type(Var(E_s(a))) \subseteq \Sigma_s]$
where $p(a)$ is the place of arc a.
(l) I_s is a partial initialization function, $I_s : K_s \cup P_s \cup T_s - PP_s \longrightarrow expr$
where $\forall x \in K_s \cup P_s \cup T_s - PP_s : I_s(x)$ is a closed expression and is defined
if $\tau(x) \neq \phi$
and $\forall x \in K_s \cup P_s \cup T_s - PP_s : [\tau_s(x) = \phi \vee Type(I_s(x)) \leq \tau_s(x)]$
and $\forall s' \in S : \sum_{x \in K_s \cup P_s \cup T_s - PP_s \ \tau(x) \in OID} I_s(x) \leq OID'_s$

Notes:

- The set OID includes multiset of object identifiers.
- Each page has a set of colour sets including the sets of object identifiers in OID. The object identifiers do not occur otherwise in Σ_s.
- It is desirable to allow constants to be associated with each page instance.
- Places are allowed as HCPN, and are distict from the constants.
- Some places can be identified as port places. These places do not hold tokens in their own right, but are used as aliases (possibly indirect) to existing places. Port places are used to specified the interaction between a super transition and its environment, by fusing the internal port places with places with places in the super transition's environment. The reserved symbol *self* may be a port place and is used by a super place to pass tokens to its environment.
- As is ususal, the set of places and transitions are disjoint. since these sets include super places and super transitions, this implies that pages or subsets normally interact with their environment in one of these two patterns.
- As in definition 3.3, there is at most one arc from each place to a transition and from each transition to a place.
- The colour function is extented to constants and transitions. This allow us to capture the type of page that a super transition instantiates. The empty type is then associated with simple transitions (which have no associated data).

- The token interface function gives the colour or type of tokens which can be offered to or accepted from places. Moreover, A token interface can include object identifiers.

- The guard function is the same as before.

- The arc expression function gives the arc inscriptions as in definition 3.3 but the type of the arc expressions is now related with token interface function , and this implies that arc can select tokens according to their class.

- The initialisation function associates an initial value with every component having a non-empty type. Each super place and super transition must be assigned a unique object identifier. Moreover, the value associate with super places, super transitions may be subclass instances.

Definitin 4.2: The places and transitions of a OO-CPN can be subdivided according to their type:

(a) The simple places $P_s^0 = \{p \in P_s | \tau_s(p) = \sigma_s(p)\}$

(b) The super places $P_{s'} = \{p \in P_s | \tau_s(p) = OID_{s'} \quad \text{for some } s' \in S\} = P_s - P_s^0$

(c) The simple transitions $T_s^0 = \{t \in T_s | \tau_s(t) = \phi\}$

(d) The super transitions $T_{s'} = \{t \in T_s | \tau_s(t) = OID_{s'}$
for some $s' \in S\} = T_s - T_s^0$

(e) The data fields $D_s = K_s \cup P_s \cup T_{s'} - PP_s$

Notes:

- The set of places includes simple places, which are the traditional CPN places. These have a type identical to the token interface type. Hence, simple places cannot hold object identifiers.

- The set of places also includes super places, which are instances of pages or subsets which offer and accept tokens via synchronous channels.

- The set of transitions includes simple transitions, which are the traditional CPN transitions.

- The set of transitions include super transitions, which are instances of pages or subnets which interact with their environment through port places.

- The data fields are all the net components which are assigned a value by the initialisation function, and therefore contribute to the marking of the net.

Definition 4.3: An OO-CPN page s_1 is a **subclass** of page s_2, written $s_1 \leq s_2$ iff:

(a) $\forall d \in D_{s_2} : d \in D_{s_1} \wedge \tau_{s_1}(d) \leq: \tau_{s_2}(d) \wedge \sigma_{s_1}(d) \leq: \sigma_{s_2}(d) \wedge Type(I_{s_1}(d)) \leq: Type(I_{s_2}(d))$

(b) $\forall a \in A_{s_2} : a \in A_{s_1} \wedge Type(E_{s_1}(a)) \leq: Type(E_{s_2}(a))$

where page s is a subclass of page s', i.e. $s \leq s'$, we also write $OID_s \leq OID_{s'}$. The relationship $\leq:$ can be extened to other colour set in some meaningful way, or else we can assert that the relationship $C_1 \leq C_2$ only holds when colour set C_1 and C_2 are identical.

Definition 4.4: An OO-CPN consisting of instances $PI_{s,M}$ of pages $s \in S$ in marking M, is a tuple

$OO{-}CPN = (OID,\ S,\ F,\ s_0,\ oid_0)$ where the following conditions hold and the following collected components are defined:

(a) each $s \in S$ is a page $(OID,\ \Sigma_s,\ K_s,\ P_s,\ PP_s,\ T_s,\ A_s,\ \tau_s,\ \sigma_s,\ G_s,\ E_s,\ I_s,)$

(b) $\Sigma = \bigcup_{s \in S} \sigma_s$

(c) $K = \bigcup_{s \in S} K_s \times PI_{s,M}$ and similarly for $P,\ PP,\ T,\ A,\ P^0,\ P',\ T^0,\ T',\ D$

(d) $\tau : K \cup P \cup T \longrightarrow \Sigma$ with $\tau((x, oid)) = \tau_s(x)$ for $x \in K_s \cup P_s \cup T_s,\ oid \in PI_s$

and similarly for σ, G

(e) $E : P \times T \cup T \times P \longrightarrow expr$ with $E(x_1, x_2) = \phi$ if $(x_1, x_2) \notin A$

and $E((x_1, oid), (x_2, oid)) = E(x_1, x_2)$ if $oid \in PI_s$

(f) $I : D \longrightarrow expr$ with $I((d, oid)) = I_s(d)$ for $d \in D_s,\ oid \in PI_s$

and $\forall s' \in S : \sum_{d \in D\ \tau(d) \in OID} I(d) \leq OID_{s'}$

(g) F is a place fusion function, $F : PP \longrightarrow expr$

where $\forall pp \in PP : [Type(F(pp)) \leq: \tau(pp)]$

(h) s_0 is the root page whose instance with object identifier oid_0 contain all other

net components.

Notes:

- The colour sets for entire OO-CPN can be collected together.
- The net components can be collected together into the sets $K, P, PP, T, A, P^0, P', T^0, T', D$.
- The net function τ, σ, G, E can be defined across the whole net.

- The initialisation function that can be defined across the whole net. It is nessary to ensure that the object identifiers uniquely identify page instances across the entire OO-CPN.
- The place fusion function is now specified by an expression which will be determined by a particular binding element. As here we introduce inheritance, a subclass may inherit from parent and argument the class components or override them within the constraint indicated.

4.7 Summary

In this chapter, we have described our OO-CPN, one of the object-oriented concurrent system, the main technical points of this system we have presented here are:

- An OO-CPN is made of a set of class nets. The instances of the class nets are called object nets.
- Precedence constraints is used to describe timing dependence.
- Inheritance mechanism, partially inherit and gloabally inherit.
- Communication between objects is achieved by tokens(messages) exchange through a certain channel.
- Different levels of page hierarchy describe the OO-CPN from the different aspects.
- The concept of component is supported in our object-oriented concurrent system; Moreover, physical and virtual or abstract components can be in a same model.

Chapter 5

Conclusions and Future Work

This chapter states the contributions of this thesis, suggests future research areas.

5.1 Contribution

State Explosion is a big drawback of Petri Nets, which makes it inapplicable to large, complex concurrent system. In order to overcome this problem, this thesis introduce a graphical formalism, object oriented Coloured Petri Nets (OO-CPN), which combines the power of modularity in object-orientated paradigm and the capability of formally modeling concurrency in Petri Nets. OO-CPN is based on Hierarchical Coloured Petri Nets [10] and supports the concept of object, class and inheritance.

In recent years, when we try to merge object-orientation paradigm with concurrent computation, we meet conflicts between inheritance and concurrency in object oriented, so called *Inheritance Anomaly* [2]. In order to solve this problem, in our OO-CPN, concurrency issues are declaratively abstracted by temporal constraints among events. In class declaration, we separate the constraint part from the part of event-related places and transitions. When we declare a subclass, it can inherit all the same places and transitions of its superclass, if and only if it have the same corresponding precedence constraints as its superclass. Moreover, such seperation of constraint part from places and transitions minimizes the dependence between application functionality and concurrency control so that allows using Petri

Net to model concurrecy issues more flexible and powerful.

As we know, interaction between the different service components is a major issue in real communication network. In the OO-CPN, communicationbetween the objects is achieved by the token(message) exchange through a certain channel. A transition in OO-CPN is an action triggered by the arrival of the tokens(message), which will move the object into a new place. A place presents a static situation in which an object finds itself and during which it is receptive to the arrival of new tokens(messages).The idea of this is based on the client-sever protocol [11] and communication channel [12]. We believe in this way, we can make Petri Net continue to be a popualr tool for describing and modeling concurrent systems.

5.2 Future Work

The work presented in this thesis can be future extended in several aspects. Inheritance anomaly is a major issue in merging object-orientation paradigm with concurrent computation. In this thesis we present our solution to this problem. We think that it constitutes a good starting point for further research and practice. However, we do not claim that it will be the final one. We hope there will be some new and better solutions which will make inheritance more applicable and flexible in concurrect system.

In this thesis, we only provide the general idea of the concept of component in OO-CPN, but not any detail extension, so the futher work in this aera would be significant for improving software reuse.

All Petri nets in this thesis were drew by hand which is rather easy but labor consuming. A computer tool that can do it in an automatic way would be a great help and make OO-CPN could be used in real large concurrent systems.

Bibliography

[1] Peter Wegner. Concepts and Paradigms of Object-Oriented Programming. *Oops Messenger*,1(1):7-87, June 1990.

[2] S.Matsuoka, K.Wakita, and A.Yonezawa. Synchronisation Constraints with Inheritancel; What is not possible? So waht is? *Technical report*, Department of Information Science, University of Tokyo, 1990.

[3] Peter Wegner. Inheritance as an Incremental Modification Mechanism, or What Like is and Isn't Like. *Proceedings of ECOOP' 88-European Conference on Object Oriented Programming*, Lecture Notes in Computer Science 322, Oslo, Norway (Eds), Springer-Verlag 1988.

[4] Lobel Crnogorac, Anands.Rao, and Kotagiri Romamohanarao. Classifying Inheritance Mechanisms in Concurrent Object-orinted Programming. *ECOOP'98 - European Conference on Object Oriented Programming*, Lecture Notes in Computer Science 1445, Eric Jul (Ed.), Springer 1988.

[5] Ciaran McHale. Synchronisation in Concurrent, Object-orientd languages: Expressive Power, Genericity and Inheritanc. PhD thesis, University of Dublin, Trinity College, 1994.

[6] D.J. Floreani, J.Bilington, A. Dadej: Designing and Verifying a Communications Gateway Using Coloured Petri Nets and Design/CPN. *Application and Theory of Petri Nets 1996*, Lecture Notes in Computer Science 1091, Jonathan Billington, Wolfgang Reisig (Eds),Springer-Verlag 1996.

[7] V.O. Pinci: The Shawmut Project. *Case Study Proceedings of the 14th International Conference on Application and Theory of Petri Nets, Chicago 1993.*

[8] H. Clausen, P.R. Jensen: Validation and Performance Analysis of Network Algorithms by Coloured Petri Nets. *Proceedings of the 5th International Workship, Toulouse, France 1993, IEEE Computer Society Press.*

[9] K.H. Mortensen, V.O.Pinci: Modelling the Work Flow of a Nuclear Waste Management Program. *Application and Theory of Petri Nets 1994*, Lecture Notes in Computer Science 815, Robert Valette (Ed.), Springer-Verlag 1994.

[10] Kurt Jensen. Coloured Petri Nets, Basic Concepts, Analysis Methods and Practical Use. Volume 1, Second Edition, Springer-Verlag 1996.

[11] C.Sibertin-Blanc. A Client-Server Protocol for the Composition of Petri Nets. *Proceedings of 14th International Conference on the Application and Theory of Petri Nets*, Lecture Notes in computer Science 691, Marco Ajmone Marson (Ed). Springer-Verlag 1993.

[12] Christensen, S., Damagaard Hansen, N. Coloured Petri Nets Extended with Channels for Synchronous Communication. *International Conference on the Application and Theory of Petri Nets.* Lecture Notes in Computer Science 815, Robert Valette (Ed.), Springer-Verlag 1994.

[13] S.Matsuoka and A.Yonezawa. Analysis of inheritance anomaly in concurrent object-oriented languages. In G.Agha, P. wegner, and A.Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107-150, MIT Press, 1993.

[14] Lakos C., Keen C.D. LOOP++: A New Language for Object-Orriented Petri Nets, Department of Computer Science, University Of Tasmania, 1994.

[15] Johan Lilius, OOB$(PN)_2$: An Object Oriented Petri Net Programming Notation (A Status Report), Digital Systems Laboratory, Helsinki University of Technology, 1996

[16] Lakos, C, Definition and Relationship to Coloured Nets. Technical Report TR94-3, Department of Computer Science, University of Tasmania, 1994.

[17] Janousek, V, PNtalk: Object Orientation in Petri Nets. In: Proceedings of European Simulation Multiconference ESM'95, Prague Technical University, Prague, 1995.