INTEGRATION OF OBJECT ORIENTED DESIGN AND COLORED PETRI NETS USING ABSTRACT NODE APPROACH

A Thesis in

Computer Science

by

Bhushan Bauskar

Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

May 2004
I grant the University of Massachusetts Dartmouth the non-exclusive right to use the work for the purpose of making single copies of the work available to the public on a not-for-profit basis if the University’s circulating copy is lost or destroyed.

________________________________________

Bhushan Bauskar

Date__________________________
We approve the thesis of Bhushan Bauskar

___________________________________    __________________
Boleslaw Mikołajczak
Professor and Chairperson, Computer and Information Science Department
Thesis Advisor

___________________________________    __________________
Iren Valova
Associate Professor, Computer and Information Science Department
Thesis Committee

___________________________________    __________________
Haiping Xu
Assistant Professor, Computer and Information Science Department
Thesis Committee

___________________________________    __________________
Jan Bergandy
Professor and Graduate Program Director,
Computer and Information Science Department

___________________________________    __________________
Farhad Azadivar
Dean, College of Engineering

___________________________________    __________________
Richard J. Panofsky
Associate Vice Chancellor for Academic Affairs and Graduate Studies
Abstract

Integration of Object Oriented Design and Colored Petri Nets using Abstract Node approach

by Bhushan Bauskar

Object Oriented Methodology is a well-established technique for structured software development. However, in concurrent systems, the main benefits of Object Oriented Design, i.e. inheritance and encapsulation are lost due to disagreement in inheritance specification and the synchronization constraints of concurrent operations that is known as Inheritance Anomaly. Inheritance Anomaly requires redefinition of inherited methods in order to maintain integrity among concurrent objects.

In order to verify the designed system, formal mathematical and executable model Petri Nets can be used. Petri Nets provide graphical representation, incorporate concurrency and parallelism. In Colored Petri Nets, objects and object attributes can be modeled with the token colors. The hierarchical structure of Colored Petri Nets is useful in representing Class Inheritance and to describe dynamics of objects.

Hence, to check the correctness of the designed system, there is a need to integrate Object Oriented techniques at design level and use of Colored Petri Nets at the Verification and Validation level of the software system development. Our thesis studies the Inheritance Anomaly in Concurrent Object Oriented Systems and provides a modeling technique to verify and validate (V&V) the Designed Object Oriented System with the Abstract Node approach in Colored Petri Nets.
# Table of Contents

## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER 1: COMBINING CONCURRENCY AND OBJECT ORIENTED DESIGN WITH COLORED PETRI NETS</td>
<td>9</td>
</tr>
<tr>
<td>1.1 Problem Statement</td>
<td>9</td>
</tr>
<tr>
<td>1.2 Objective of this Thesis</td>
<td>10</td>
</tr>
<tr>
<td>1.3 Inheritance Anomaly in Concurrent Object Oriented Systems</td>
<td>10</td>
</tr>
<tr>
<td>1.4 Approaches to solve Inheritance Anomaly</td>
<td>11</td>
</tr>
<tr>
<td>1.5 Integration of Object Orientedness with Petri Nets</td>
<td>12</td>
</tr>
<tr>
<td>1.6 Petri Nets and Verification and Validation of Object Oriented Design</td>
<td>14</td>
</tr>
<tr>
<td>CHAPTER 2: INHERITANCE SPECIFICATION WITH COLORED PETRI NETS</td>
<td>16</td>
</tr>
<tr>
<td>2.1 Introduction</td>
<td>16</td>
</tr>
<tr>
<td>2.2 State Partitioning</td>
<td>17</td>
</tr>
<tr>
<td>2.3 History only Sensitiveness</td>
<td>28</td>
</tr>
<tr>
<td>2.4 State Modification</td>
<td>37</td>
</tr>
<tr>
<td>CHAPTER 3 FORMALIZATION OF ABSTRACT NODE APPROACH</td>
<td>44</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>44</td>
</tr>
<tr>
<td>3.2 Transformation of Object Oriented Designed System to the Abstract Node Model</td>
<td>45</td>
</tr>
<tr>
<td>3.3 Steps to transform the Object Oriented Design to the Abstract Node Model</td>
<td>46</td>
</tr>
<tr>
<td>3.3.1 Input and Output are places for Abstract Node</td>
<td>47</td>
</tr>
<tr>
<td>3.3.2 Input is a place and Output is a transition for Abstract Node</td>
<td>47</td>
</tr>
<tr>
<td>3.3.3 Input is a transition and output is a place to Abstract Node</td>
<td>48</td>
</tr>
</tbody>
</table>
3.3.4 Input and Output are transitions for Abstract Node
3.3 Further refinements of Abstract Node model
3.4.1 First Level Refinement
3.4.2 Second Level Refinement

CHAPTER 4 ATM SYSTEM DESIGN, VERIFICATION AND VALIDATION

4.1 ATM Requirement Specifications
4.2 ATM Representation in Unified Modeling Language (UML)
4.3 ATM Representation in Colored Petri Nets

CONCLUSIONS
REFERENCES
List of Figures

FIGURE 2.2.1: CLASS BUFFER AND ITS SUBCLASS NEWBUFFER 17
FIGURE 2.2.2: STATE PARTITIONING ANOMALY REPRESENTATION 18
FIGURE 2.2.3: REDEFINITION OF METHOD IN PARENT BUFFER CLASS 18
FIGURE 2.2.4: STATE DIAGRAM OF CLASS BUFFER WITH CAPACITY =1 19
FIGURE 2.2.5: STATE DIAGRAM OF CLASS BUFFER WITH CAPACITY =2 19
FIGURE 2.2.6: STATE DIAGRAM OF CLASS BUFFER WITH CAPACITY >2 20
FIGURE 2.2.7: STATE DIAGRAM OF CLASS NEWBUFFER WITH CAPACITY =2 20
FIGURE 2.2.8: STATE DIAGRAM OF CLASS NEWBUFFER WITH CAPACITY >2 21
FIGURE 2.2.9: STATE DIAGRAM AFTER PARTITIONING OF CLASS NEWBUFFER WITH CAPACITY >2 21
FIGURE 2.2.10: SOLUTION TO STATE PARTITIONING 22
FIGURE 2.2.11: HIERARCHICAL REPRESENTATION OF CLASS NEW BUFFER 23
FIGURE 2.2.12: CLASS NEWBUFFER AS COLORED PETRI NETS MODEL 24
FIGURE 2.2.13: CLASS BUFFER AS COLORED PETRI NETS MODEL 25
FIGURE 2.2.14: GET2 METHOD AS COLORED PETRI NETS MODEL 27
FIGURE 2.3.1 CLASS BUFFER AND ITS SUBCLASS HISTORYBUFFER 28
FIGURE 2.3.2 STATE DIAGRAM OF CLASS BUFFER WITH CAPACITY =1 29
FIGURE 2.3.3 STATE DIAGRAM OF CLASS BUFFER WITH CAPACITY =1 30
FIGURE 2.3.4 STATE DIAGRAM AFTER PARTITIONING OF STATES OF CLASS HISTORYBUFFER WITH CAPACITY >= 3 31
FIGURE 2.3.5 CLASS HISTORYBUFFER WITH EVENT QUEUE 32
FIGURE 2.3.6 CLASS HISTORYBUFFER WITH POST ACTIONS 32
FIGURE 2.3.7 HIERARCHICAL MODEL OF CLASS HISTORYBUFFER IN COLORED PETRI NETS 33
FIGURE 2.3.8 CLASS HISTORYBUFFER COLORED PETRI NETS MODEL 34
FIGURE 2.3.9: CLASS BUFFER COLORED PETRI NETS MODEL WITH INCREMENTAL UPDATION 35
FIGURE 2.3.10 GGET METHOD IN COLORED PETRI NETS MODEL 36
FIGURE 2.4.1 CLASS LOCK, MIXED-IN LOCKABLEBUFFER CLASS 38
Chapter 1: Combining Concurrency and Object Oriented Design with Colored Petri Nets

1.1 Problem Statement

Object Oriented Methodology is a well-established technique for structured software design. Object Orientedness supports Inheritance, Polymorphism and Dynamic Binding. Object Oriented Methodology is useful in order to design software that is comprehensible, flexible and maintainable [LC]. It supports incremental development. However, Object Oriented methodology lacks analysis and verification methods of designed systems. It does not support verification and the validation of concurrency in the designed system [MM].

Petri Nets, on the other hand, is a formal mathematical model of computation that is executable and incorporates concurrency, resolves conflict and supports resource sharing. The graphical representation, simplicity and the most significant, the executable nature of Petri Nets model makes Petri Nets suitable for simulation, rapid prototyping and verification of designed systems [MM]. However, Petri Nets lack clear and effective technique to specify design of a system. Petri Nets do not comment on how to initiate modeling of the system. Object in the Petri Nets can be represented as a place or as a token inside the place. Hence, a technique may be required to analyze the system before modeling in Petri Nets.

While designing Concurrent Object Oriented systems, we encounter the problem of Inheritance Anomaly. Inheritance Anomaly is the incongruity between concurrency and inheritance where extensive re-definitions of inherited methods are necessary in order to maintain the synchronization constraints of concurrent objects [CRR].

There is a need to integrate both Object Oriented methodology and Petri Nets modeling in order to verify and to represent a Concurrent Object Oriented system. In this thesis, we are considering an approach of Abstract Node stated in [MM] to unite both the above-mentioned methodologies by using Object Orientedness at design stage and Petri Nets at verification and validation stage. We say that Abstract Node is a method of designing large concurrent systems, which is based on the combination of Object Oriented Design and High-Level Petri Nets [MM].
In this thesis, we will also revise the problem of Inheritance Anomaly in Concurrent Object Oriented systems [CRR]. We will state how Inheritance Anomaly can be implemented, verified and validated with the Abstract Node approach [MM].

1.2 Objective of this thesis

Our thesis will first look at the various approaches to handle Inheritance Anomaly that arises due to incorporation of Object Orientedness with Concurrency. We will also study the various approaches to integrate Object Orientedness and Petri Nets. Later, we will discuss an approach of Abstract Node to integrate Object Oriented Design with Petri Nets. With an example of academic interest, we will present a Petri Nets model with Abstract Node approach for an Object Oriented Designed system. We will also present this model to show Inheritance relationship, to show various Objects and inter-object concurrency and intra-object concurrency and to verify and validate the designed model. We will present a technique to model an Object Oriented Designed system with Abstract Node approach and thus to show how an object structure, inheritance relationship and the concurrency can be modeled with Colored Petri Nets.

The goal of our thesis is to present how an Object Oriented Designed system can be modeled with Abstract Node method to Colored Petri Nets model.

1.3 Inheritance Anomaly in Concurrent Object Oriented Systems

Inheritance is a prime language feature of Object Oriented languages and it is important for code reuse. Most of the current systems require concurrency for better performance or the systems are concurrent in nature. In such concurrent object oriented system we often see the breakage of encapsulation property. This is because Inheritance and Concurrency show disagreement in their characteristics and hence we need to redefine the synchronization constraints [CRR, MY].

Following can be the situations where the benefits of inheritance are lost in concurrent systems:

- Definition of new subclass K’ of class K requires redefinition of methods in class K and also in parent class of K.
• Modification of new method ‘m’ of class K within the inheritance hierarchy requires the modification of methods in both parent class (superclass) and subclass of K.

• Definition of method ‘m’ might force the other methods (including those defined in subclass) to follow a specific protocol which may not be required if this method ‘m’ is not defined. This is one of the reasons for encapsulation in mixed-in classes to be difficult [MY].

1.4 Approaches to solve Inheritance Anomaly

1.4.1 Shibayama’s Proposal

Shibayama [MY] proposes a scheme to minimize the redefinition of code by classifying methods as primary, constraint, and transition methods. A method of one category may have its counterparts with similar keys in other categories and each of them can be separately defined or inherited. The methods are categorized as follows:

• A primary method will only perform some task rather than providing object wise synchronization.

• A constraint method acts as a method guard. These methods can be redefined independent of primary methods; hence the functionality remains unaffected in primary methods.

• A transition method determines how the messages are passed. Its redefinition allows the dynamic modification of the delegation path.

In this approach, by separating the synchronization code from other parts of method definitions, the amount of re-definitions is minimized [MY].

1.4.2 Meseguer’s Proposal

Meseguer [MJ] proposes a new formalism for modeling concurrent systems in an Object Oriented Concurrent Programming language called Maude. Meseguer’s states a rewriting logic, which
models concurrent computation as its special instantiation. There are two types of modules defined: system and functional. The Maude language provides object oriented modules to provide ease in writing concurrent object oriented programs. While execution these object oriented modules are first translated into system modules and then computation proceeds with concurrent rewriting according to the rewrite rules of translated modules. Inheritance anomaly is avoided by placing conditions in rewritable codes, which serves as guards [MJ].

1.4.3 FrΦlund’s Proposal

FrΦlund [FSv] proposes a framework, which mostly concentrates on synchronization code for the derived methods. He proposes a design in which synchronization constraints get increasingly restrictive in subclasses. Basically, one specifies the guard that gives a condition, under which a method cannot be accepted, i.e. a negative guard. The reuse works in restrictive way where guard conditions are also inherited in the subclasses. So, if ‘m’ is the method in the subclass then for all the methods with name ‘m’ in the parent class, the guard will be restrictive therefore it will evaluate false to accept a method ‘m’ of subclass. This work is helpful to understand how operations are derived and it solves the problem of inheritance anomaly only in derived operations. But the problem of inheritance anomaly can occur in unrelated methods that might require redefinition [FSv].

1.4.4 Our approach

We will be using the approach of preconditions and the post actions (described in detail in chapter 2) to solve the problem of Inheritance Anomaly [CRR]. Every object is defined with accept set based terminology. Object can only accept the subset of their messages at a particular time. When a concurrent object is in certain state, it can accept only subset of its entire set of messages in order to maintain its internal integrity. This is synchronization constraint of a concurrent object [MY]. This synchronization constraint will be specified by preconditions and post actions, and hence the Inheritance anomaly can be avoided [CRR].

1.5 Integration of Object Orientedness with Petri Nets

Petri Nets are useful in describing and studying information systems that are characterized as being concurrent, asynchronous, distributed, parallel, nondeterministic and/or stochastic [MT].
In the past, various researchers have presented a methodology to integrate Petri Nets with Object Orientedness. The approaches are useful to represent concurrent nature of objects. The real world systems are concurrent in nature; hence these approaches are helpful to show concurrent object oriented systems. The combination of Object Oriented technique which provides inheritance (code reusability), polymorphism, dynamic binding and Petri Nets which provides parallel system modeling, verification and analysis including conflict resolution, resource allocation will give the above advantages.

There are various approaches put forth by researchers to combine Object Orientedness and Petri Nets. They are described below.

1.5.1 CO-Operative Objects (COO)

This approach is stated and developed by Sibertin-Blanc and Bastide. COO defines a system as a collection of Objects and Cooperation among them. COO is an object-oriented language, which retain features like encapsulation, dynamic binding, inheritance, and client/server communication. System can be looked at as Objects and the way these Objects will communicate i.e. Cooperation. Therefore,

\[
\text{System} = \text{Objects} + \text{Cooperation} \\
\text{Object} = \text{Data Structure} + \text{Operations} + \text{Behavior}
\]

Thus, system is a collection of high-level objects that encapsulates the control structure and communicates through the request reply protocol.

Tool SYROCO is the C++ implementation of the COO formalism [SB].

1.5.2 Object Petri Nets (OPN)

Lakos developed OPN from the Colored Petri Nets (CPN) through a series of formal transformations. Object contains the token that encapsulates the new object, which is a Petri Nets (PN) structure. It presents the nested or the hierarchical view of the system. Thus, the system has a single class hierarchy, which includes both the token types and the subnet types, providing the multiple levels of the activity in the net.
Tool LOOPN++ (Language for Object Oriented Petri Nets ++) is an implementation of OPN using C++ [LCh].

1.5.3 Concurrent Object Oriented Petri Nets (CO-OPN)

This approach is developed by Buchs. CO-OPN incorporates both concurrency and data structuring features. It is a structured extension of PN and algebraic abstract data types. CO-OPN specification is composed of set of objects and synchronous communication between them. Objects are algebraic abstract data types and PN.

CO-OPN specification consists of set of objects described by algebraic Petri Nets that are connected over a communication integrating sequential and concurrent conditions.

Tool supported is CO-OPN [WS1].

1.5.4 Object Oriented Petri Nets (OOPN)

OOPN is developed by Vojnar, Milan Ceska and Vladimir Janousek. OOPN are based on CPN enriched by object-oriented paradigm. OOPN formalism is characterized by Smalltalk-based Object-orientation enriched with concurrency and polymorphic transitions execution. OOPN allows message sending, receiving and creating new objects and performing primitive computations. A class consists of an object net, method nets and predicates.

Tool Supported PNTALK [VCJ, WS2]

1.6 Petri Nets and Verification and Validation of Object Oriented Design

Object Oriented Design (OOD) defines the Object structure and relationships between objects. Petri Nets model object behavior, object communication with internal and external object concurrency. Petri Nets are also suitable for rapid prototyping and performance evaluation. Hence, Petri Nets can be useful to verify and validate the designed model.

Object oriented design is a proper stage of system development to introduce concurrency and formal specification into software development as
• Petri Nets and Object Oriented Methodology provide a unified framework for development of software systems as both support incremental design.
• System designed as a result of this approach could be characterized as loosely coupled components with high cohesion.
• Classes and objects represented by means of Petri Nets support reusability of software with its structure and behavior encapsulated into a single graphical document that helps software designer to understand functionality of the class.
• Classes and objects whose inheritance structure is defined as a form of class classification and behavior described by means of Petri Nets allow organized browsing of class libraries [MM].

Since, Petri Nets models are executable. The Object Oriented designed models can be made executable by transforming them into Petri Nets models. Thus, the designed system can be validated by simulation. The formal specification of a system can be validated against its informal initial requirements while involving its end-users and owners. Specific scenarios can be tested on the verification and validation model. The model describes the interactions among objects and these objects can be grouped together to compose into (sub-) systems. That is software can be developed with “the bottom-up” approach. The specification is then transferred into a modular Petri Nets standing with an Object-Oriented semantics [DE].
Chapter 2: Inheritance Specification with Colored Petri Nets

2.1 Introduction

Inheritance is one of the important features of Object Oriented Methodologies. Inheritance is mainly useful for source code reuse. Concurrency issues are the prime concerns for the development of software systems. In designing of concurrent systems, incompatibility arises in specifying inheritance and concurrency, as there is no facility provided in the programming languages for the implementation of synchronization constraints of concurrent objects.

Thus, Inheritance Anomaly can be stated as the inconsistency between concurrency and inheritance where extensive re-definitions of inherited methods may be necessary in order to maintain the synchronization constraints of concurrent objects. Inheritance anomaly can be classified into the following types: State Partitioning, History-Only-Sensitiveness and State Modification [CRR]. This classification is based on the state of an object, which is dependent on the messages that object accepts. The state of that object changes when these messages get modified due to their redefinitions in subclasses.

The Inheritance Anomaly can be tackled with the use of pre-conditions and post-actions in which synchronization constraints of objects are specified. Every method definition of an object consists of a block of pre-conditions, a method body and a block of post-actions. The synchronization code is stated in pre-conditions and post-actions and is clearly separated from functional code of the method, which solves the anomaly. These blocks of pre-condition and post-action can be updated in the subclass. Incremental modification of pre-condition and post-action blocks is supported when inherited methods are specialized and generalized. [CRR]

Incremental modification of the pre-condition and post-action blocks is supported by three operators: $\Lambda$, $V$, $\mid$. The composition operator ‘$\Lambda$’ enables us to add new conditions to already existing set of preconditions. This operator stands for the conjunction of conditions. The operator ‘$V$’ stands for a disjunction of two conditions. The operator ‘$\mid$’ is a restriction operator. Incremental modification of the post actions block is supported by the operator ‘$;$’ which denotes sequential execution. The operator ‘$*$’ denotes 0 or more occurrences of the stated expression [CRR].
2.2 State Partitioning

2.2.1 State partitioning Definition

In Object-Oriented methodology, an object has a state and the state of an object changes as per the behavior of the object during its execution. The object is said to have a set of acceptable states i.e. a possible set of states in which an object can exist during its execution. The set of acceptable states is defined as per the synchronization mechanism for that object. This set can be partitioned into disjoint subsets according to the synchronization constraints of the object. For example, consider buffer example [CRR] shown in Figure 2.2.1. An object Buffer with methods put() and get(), is defined with the accept set based synchronization mechanism. An accept set is a subset of acceptable methods for a particular object in its particular state. The synchronization mechanism uses the keyword ‘accept’ to specify the set of methods acceptable to the object. Method put() is acceptable unless buffer is full and method get() is acceptable if buffer is not empty. The variable numberOfElements represents the current value of the number of elements in a buffer and the variable Maxsize represents the maximum capacity for the buffer.

```java
class Buffer {
    body:
    If (numberOfElements==0) accept {put};
    else if (numberOfElements < Maxsize) accept {put, get};
    else if (numberOfElements==Maxsize) accept {get};
    methods:
    put {...}
    get {...}
}
class NewBuffer:Buffer {
    methods:
    get2 {
        code for get2
    }
}
```

**FIGURE 2.2.1: CLASS BUFFER AND ITS SUBCLASS NEWBUFFER**

Thus, there are three disjoint set of states, under which Buffer can accept a message: Empty, Partial and Full. This is shown in Figure 2.2.2 below. Now, when a new method is added in the subclass, the splitting up of the set of states in the parent class may need to be further divided in the subclass. This is because the synchronization constraint of the new method may not be properly defined in order to account for partitioning of parent class. In our example, we define a subclass of Buffer, NewBuffer, with an additional method get2(), which gets two elements from the object. This is shown in Figure 2.2.1.
Method `get2()` can be used if `NewBuffer` contains more than one element. Thus, when the `get2()` method was added in the `NewBuffer` subclass, the partitioning the `Partial` state into `one_element` and `more_than_one_element` is necessary in order to distinguish the state where buffer can contain only one element. This is shown in Figure 2.2.2.

Thus, with the addition of `get2()` method there are four possible states of `Buffer` `{Empty, one_element, more_than_one_element, Full}` instead of original three. So, redefinition of method code is necessary to account for newly partitioned state space in all the methods (as shown in Figure 2.2.3) and hence the anomaly appears. The State Partitioning occurs as addition of new method further partitions the state space of an Object.

```java
class Buffer {
    body:
    if (numberOfElements==0) accept {put};
    else if (numberOfElements == 1) accept {put, get};
    else if (2<= numberOfElements < Maxsize) accept {put, get, get2};
    else if (numberOfElements==Maxsize) accept {get, get2};
    methods:
    put {…}
    get {…}
}
```

**Figure 2.2.2: State Partitioning Anomaly Representation**

**Figure 2.2.3: Redefinition of Method in Parent Buffer Class**
2.2.2 Partitioning of the states due to get2() method

Consider the case when buffer capacity = 1. There are only two possible states of buffer i.e. \{Empty, Full\}. A state diagram representing method calls `put()` and `get()` is shown below in Figure 2.2.4. (p/a meaning there is a call for method `put()` and that method `put()` is accepted. g/a meaning call for method `get()` and method `get()` is accepted. p/r call for method `put()` and it is rejected and g/r meaning call for method `get()` and it is rejected.)

Implementation of method `get2()` is of no use for this case as buffer capacity is one and so we cannot get two elements from the buffer. Hence let’s not consider this case.

![Figure 2.2.4: State Diagram of Class Buffer with Capacity = 1](image)

Consider the case when buffer capacity = 2. In this case buffer can be Empty, having no element or buffer is not empty and not full i.e. Partial state or buffer is Full by having number of elements equal to buffer capacity. So, there are three possible states of the buffer, i.e. \{Empty, Partial, Full\}. A state diagram is shown in Figure 2.2.5 with the method calls of `put()` and `get()` on this buffer.

![Figure 2.2.5: State Diagram of Class Buffer with Capacity = 2](image)
Similarly, we can show the state diagrams for the buffer capacity = 3 and buffer capacity = 4 and so on. We notice that for all these cases (buffer capacity > 2) the state diagram remains the same as shown in Figure 2.2.6. (If the same set of methods are acting on the buffer i.e. put() and get() in this case).

![Figure 2.2.6: State Diagram of Class Buffer with Capacity > 2](image)

Now, consider a case when a new method is added to the above buffer with buffer capacity >= 2. (i.e. get2() method is added). Now, buffer can have three method calls {put(), get(), get2()}. If buffer capacity = 2 then the state diagram will be as shown below in Figure 2.2.7. (G2/a meaning method get2() is accepted and G2/r meaning method get2() is rejected.) We can see that method get2() is rejected in partial state of the buffer with capacity = 2 as meaning of partial state for buffer capacity = 2 is buffer having one element.

![Figure 2.2.7: State Diagram of Class NewBuffer with Capacity = 2](image)
Consider the cases when buffer capacity > 2. The state diagram for all these cases will be as shown below in Figure 2.2.8.

**Figure 2.2.8: State Diagram of Class newbuffer with capacity >2**

State is defined as *Partial* state if buffer is neither empty nor full. We can clearly state that `get2()` can be accepted or `get2()` can be rejected in this *Partial* state. For example, consider the buffer currently having only one element so buffer is in *Partial* state. If there is a method call `get2()` then it will be rejected, but if we consider the fact that buffer is having more than one element and in *Partial* state then the same method call `get2()` is accepted in this state.

**Figure 2.2.9: State Diagram after partitioning of Class newbuffer with capacity >2**
That is for the same state of buffer Partial, method call get2() can be accepted or rejected. Thus, it shows nondeterminism and hence we need to partition this Partial state into buffer_with_one_element and Partial_buffer_but_more_than_one_element.

So, after addition of new method get2(), buffer will have four states {Empty, buffer_with_one_element, partial_buffer_but_more_than_one_element, Full}. The state diagram for this partitioned state buffer will be as shown below in Figure 2.2.9. We can state that after addition of a new method partitioning of state is required in order to distinguish between different states of buffer. The dotted arrow in the diagram represents the case that is applicable only for buffer capacity = 3. This dotted arrow states the method get2() is acceptable (G2/a) in this case.

### 2.2.3 Solution to State Partitioning

The partitioning of states can be avoided as shown in Figure 2.2.10. Every method has a pre-condition attached to it. A new method is added along with its pre-conditions, which maintains the synchronization constraints. Hence, if new methods were added then pre-condition code for the parent class methods need not be redefined. The separation of concurrency issues from the functionality code solves the problem. [CRR]

```java
class Buffer {
    method put {
        pre: If (numberOfElements < Maxsize)
        code for put
        post
    }
    method get {
        pre: (numberOfElements > 0)
        code for get
        post
    }
}

class NewBuffer:Buffer {
    method get2 {
        pre: (numberOfElements >= 2)
        code for get2
        post
    }
}
```

**FIGURE 2.2.10: SOLUTION TO STATE PARTITIONING**

Next, we have transformed the above Object Oriented Designed solution to Petri Nets model with our Abstract Node (AN) approach. We modeled the system in Petri Nets in Design/CPN, Colored Petri Nets tool.
2.2.4 Abstract Node (AN) representation for State Partitioning

Places, Transitions, AN and their meaning for class Buffer and Class NewBuffer:

The hierarchical representation of class NewBuffer is as shown in Figure 2.2.11.

The class NewBuffer is inherited from parent class Buffer and it has its own method get2(). If there is a call for the methods in parent class, i.e. Buffer and if it satisfies the pre-conditions then the methods from parent class will be executed. If the call is for its own method i.e. get2() and if it satisfies the pre-conditions then that method will be executed.

The Figure in 2.2.12 represents the Colored Petri Nets model of the hierarchical NewBuffer class that is shown in Figure 2.2.11.
FIGURE 2.2.12: CLASS NEWBUFFER AS COLORED PETRI NETS MODEL

1. **methodcall**: the place `methodcall` can store methods that are public methods to class `Buffer` and class `NewBuffer`. It is a place, which represents the pool of methods to be executed. All the methods are initialized here and later sent for execution in random order.

2. **Buffer Capacity**: it is a place that keeps track of the current number of elements in the buffer. This place is initialized to zero representing that initially buffer is empty i.e. there are no elements in the buffer. This place has the maximum capacity (in this case it is 5) showing that this is a bounded buffer. The maximum capacity is stated in the declaration of Color Set.

3. **Methods used**: methods that are successfully executed enter this place.

4. **Call Buffer Methods**: it is AN representation of the class `Buffer`. It takes an input from place `methodcall` and a place `Buffer Capacity` and the output from `Call Buffer Methods` transition goes to `methods used`. As the name suggests, this AN will accept only public methods of class `Buffer`, any other method call will not be accepted. In the Design/CPN, the Abstract Node can
be shown as a Hierarchical substitution transition. For every such hierarchical substitution transition we need to specify their input and output places. These places are represented with P In and P Out in the sub page for that hierarchical transition. These places are socket places, and we need to define if these socket places are input (In) or output (Out) or both for the defined hierarchical substitution transition. This transition will be explored as shown in Figure 2.2.13.

![Diagram](image)

**FIGURE 2.2.13: CLASS BUFFER AS COLORED PETRI NETS MODEL**

4.1. When a request is made to execute a public method from the class *Buffer*, then it is passed to the *Call Buffer Method* AN, which is divided into number of transitions as number of public methods present in the class *Buffer*. In this case it is divided into two distinct transitions, *callput* and *call get*, which represent call for method put and call for method get respectively as shown in Figure 2.2.13.

4.1.1. *callput*: when a call for method put is made, then it is guarded here with the precondition (stated in Figure 2.2.13) to check if the number of elements in the buffer are less than max size, if yes then the method is accepted and executed.

4.1.2. *call get*: when a call for method get is made then it is guarded here with the precondition to see if buffer is empty (as stated in Figure 2.2.13).
4.2. \textit{put accepted} and \textit{get accepted}: these are places, which say that if \textit{callput} or \textit{call get} transition is fired respectively.

4.3. \textit{put} and \textit{get}: it is a transition where the method body is implemented. The transition \textit{put} will add an element in the buffer and will increment \textit{Buffer Capacity} by one and transition \textit{get} will remove an element from the buffer and will decrement the \textit{Buffer Capacity} by one. When the method is executed that is when this transition fires a token goes to place \textit{methods used} showing the successful execution of that method.

5. \textit{Call for Get2}: it is AN representation for the method \textit{get2}. In our example class \textit{NewBuffer} has only one method of its own that is \textit{get2}. Hence, it is explored in to only one transition \textit{call get2}. If there were more of its own methods for class \textit{NewBuffer}, then this AN would have been explored into number of transitions equal to number of own methods of the class \textit{NewBuffer} as shown in Figure 2.2.14.

5.1. \textit{call get2}: when a request is made to execute \textit{get2} method then the transition \textit{call for Get2} is enabled. Since it is an AN, it is explored in to the transition \textit{call get2}, where it is guarded with the pre-condition to check if there are at least two elements in the buffer (Stated in Figure 2.2.14).

5.2. \textit{get2 accepted}: when the pre-condition for method \textit{get2} is evaluated true, a token is passed in this place stating method \textit{get2} is accepted and now can be executed.

5.3. \textit{get2}: it is a transition, which represents body of the method \textit{get2} and any post actions. When \textit{get2} is fired \textit{Buffer Capacity} is decremented by two and a token goes to place \textit{methods used} representing successful execution of the method \textit{get2}. 
FIGURE 2.2.14: GET2 METHOD AS COLORED PETRI NETS MODEL
2.3 History only Sensitiveness

2.3.1 History Only Sensitiveness Definition

Sometimes in Concurrent Object Oriented Programming (COOP), the synchronization constraints are designed depending on distinguishable states of an object, that is, these synchronization constraints of the method, defined in the subclass depend on the history of the method invocations in that object. The existing class variable cannot distinguish between the different historical states. This historical information needs to be recorded in an additional instance of a variable and thus all the methods referring to this variable will have to be modified [KY]. Hence, we need to introduce extra variables, which are updated as the object executes methods. The introduction of these variables causes redefinition of the code to represent historical conditions. This causes appearance of the anomaly, which is History Only Sensitiveness [CRR]. Consider the example of a finite Buffer with methods put() and get() stated earlier in 2.2.1. We define the subclass of Buffer, HistoryBuffer, with a new method gget() as shown in Figure 2.3.1. This method behaves exactly like get() except that it cannot be called immediately after put() method call [CRR].

```java
class HistoryBuffer: Buffer {
    int after_put = false;
    methods:
        gget when (!(after_put && (numberOfElements > 0))) {
            code for gget
            after_put = false;
        }
        put when (numberOfElements < Maxsize) {
            code for put
            after_put = true;
        }
        get when (numberOfElements > 0) {
            code for get
            after_put = false;
        }
}
```

**Figure 2.3.1 Class Buffer and its Subclass HistoryBuffer**

Inheritance anomaly appears here as the variable space of the objects of super class does not record the information when the put() operation is invoked last. For this a new variable has to be introduced: after_put, to record the traces of events. In order to record the history of method calls, variable after_put has to be changed in the inherited put() and get() operations. Thus, the synchronization constructs of new method gget() is built. That is methods, put() and get() of
superclass are to be redefined and so inheritance is lost and anomaly appears [FS]. Here, gget() method partitions the states as well it is History only Sensitive [MY].

2.3.2 Partitioning of State due to gget()

We define a class Buffer with methods put() and get(), and then we define a subclass HistoryBuffer with an additional method gget(). This method gget() is similar to get() but it can not be invoked immediately after put() method call. Here, gget() method partitions the state as well it causes History only Sensitive anomaly [MY].

Consider the case when buffer capacity = 1. There are only two possible states of buffer i.e. {Empty, Full}. A state diagram representing method calls put() and get() is shown below in Figure 2.3.2. (p/a meaning there is a call for method put() and that method put() is accepted. g/a meaning call for method get() and method get() is accepted. p/r call for method put() and it is rejected and g/r meaning call for method get and it is rejected).

Implementation of method gget() is of no use for this case as this can be seen in the state diagram. As buffer capacity is one, Buffer goes in Full state and Empty state after method calls of put() and get() respectively. When buffer is full that means the last method call is put() and hence gget() is not acceptable. Now, Buffer can accept only get(), and after invoking get(), Buffer goes to Empty state and hence it can not accept get() or gget(). Hence let us not consider this case.

![Figure 2.3.2 STATE DIAGRAM OF CLASS BUFFER WITH CAPACITY =1](image)

Consider the case when buffer capacity = 2. In this case buffer can be Empty, having no element or buffer is not empty and not full i.e. Partial state or buffer is Full by having number of elements equal to buffer capacity. So, there are three possible states for the buffer, i.e. {Empty, Partial,
A state diagram is shown in Figure 2.2.5 with the method calls of put() and get() on this buffer.

Similarly, we can show the state diagrams for the buffer capacity = 3 and buffer capacity = 4 and so on. We notice that for all these cases (buffer capacity >= 2) the state diagram remains the same as shown in Figure 2.2.6. (If the same set of methods are acting on the buffer i.e. put() and get() in this case).

Now, consider a case when a new method call is added to the above Buffer with buffer capacity >= 2. (i.e. gget() method is added). So now buffer has three method calls {put(), get(), gget()}. If buffer capacity >= 2 then the state diagram will be as shown below in Figure 2.3.3. (g/a meaning method gget() is accepted and g/g/r meaning method gget() is rejected when method gget() is been invoked.) The dotted arrow is not present for the buffer capacity = 2.

State is defined as Partial state if buffer is neither empty nor full. We can clearly state that gget() can be accepted or gget() can be rejected in this Partial state. For example, consider the buffer currently in partial state after a put() method call. So, if there is a method call gget(), then it will be rejected. But, if we consider the fact that buffer in partial state after a get() is called, then the same method call gget() will be accepted in this state.

That is, for the same state of buffer Partial, method call gget(), can be accepted or rejected. Thus, it shows nondeterminism. Hence, we need to partition this Partial state in to after_put and
after_get to show the distinct states of buffer or we need an extra variable to trace these history conditions.

So, after addition of new method gget(), buffer will have four states {Empty, after_put, after_get, Full}. The state diagram for this partitioned state buffer will be as shown below in Figure 2.3.4. We can state that partitioning of state is required in this case in order to distinguish between different states of buffer after addition of a new method. Hence we can state that gget() method causes state partitioning as well it can be history only sensitive.

![State Diagram](image)

**Figure 2.3.4 State Diagram After Partitioning of States of Class HistoryBuffer with Capacity >= 3**

### 2.3.3 Solution to History Only Sensitiveness

History only Sensitiveness can be avoided by maintaining an event queue which will keep track of method calls that are invoked so far. Thus, History only Sensitiveness can be avoided by adding pre-conditions that will monitor the state of this event queue, and will determine the invocation of the method.

Consider the code that defines the class HistoryBuffer with the method gget(). It is shown below in Figure 2.3.5. Accept[method name], suggest the method accepted by an object for execution.
The event(1) != Accept(put) will check whether the last event executed is Accept(put) and likewise we can check for any number of past method invocations by specifying the number in the call, event(number). The expression super(Buffer.get) in method gget() refers to the precondition block of method get() in the super class Buffer. Handling of events in such a way will introduce an extra cost to the system due to storage needed to maintain the queue and also computational effort required in case of event handling. Hence, the event queue can be implemented with use of post-actions. The pre-conditions will act as method guards and post-actions will monitor history only sensitive problems [CRR].

```java
class HistoryBuffer : Buffer {
    int after_put = false;
    method put {
        post: super(Buffer) ; (after_put = true;)
    }
    method get{
        post: super(Buffer) ; (after_put = false;)
    }
    method gget{
        pre: (!after_put) ^ super(Buffer.get)
        code for gget
        post
    }
}
```

**FIGURE 2.3.6 CLASS HISTORYBUFFER WITH POST ACTIONS**

The new variable after_put is introduced to keep track of history information. Here, the post-actions of method put() and get() are incrementally updated to keep track of history information [CRR]. Thus, method code remains unchanged and the post-actions are not redefined, and in this manner anomaly is solved.
2.3.4 Abstract Node representation of History Only Sensitiveness

Places, Transitions and AN, their meaning for class Buffer and HistoryBuffer:

![Hierarchical model of class HistoryBuffer in colored Petri nets](image)

The hierarchical representation of class HistoryBuffer is as shown in Figure 2.3.7. The class HistoryBuffer is inherited from parent class Buffer and it has its own method gget(). If there is a call for the methods in parent class, i.e. Buffer and if it satisfies the pre-conditions then the methods from parent class will be executed. If the call is for its own method i.e. gget() and if it satisfies the pre-conditions then that method will be executed.

The Figure in 2.3.8 represents the Petri Nets model of the hierarchical HistoryBuffer class that is shown in Figure 2.3.7. We have implemented our solution of History only Sensitiveness for the class HistoryBuffer with the post-actions approach. We have introduced a new variable after_put which is incrementally updated and hence anomaly is avoided. This new variable is represented with a place after put in our Petri Nets model in Figure 2.3.8.
The meaning of all the places and transition of class Buffer will remain the same as described earlier in section 2.2.4 except:

1. **after_put**: the place represents if the previously executed method is `put()` or any method other than put. When any method of class Buffer is executed then this place is modified (as post actions), depending on its value. When the method `gget()` is called then this place is checked in the pre-conditions to see if the previous method executed is put or not.
2. *call Buffer Methods*: the hierarchical transition will represent the AN representation of class *Buffer* as described earlier in section 2.2.4 except there is also an input from place *after_put* place. This is done in incremental fashion by adding this place in the Petri Nets model and then drawing the proper input and output arcs.
2.1. *put*: this transition represents the method *put*. When *put* method is executed it will update the place *after_put* to absent meaning last method executed is put method, as shown in Figure 2.3.9.

2.2. *get*: this transition represents the method *get*. When *get* method is executed it will update the place *after_put* to present meaning last method executed is not put method, as shown in Figure 2.3.9.

3. *Call for GGet*: it is an AN representation for the method *gget().* In our example class *HistoryBuffer* has only one method of its own that is *gget().* Hence, it is explored in to only one transition *call gget*. If there were more of its own methods for class *HistoryBuffer*, then this AN would have been divided into number of transitions equals to number of own methods of the class *HistoryBuffer* as shown in Figure 2.3.10.

![Figure 2.3.10 GGet method in Colored Petri Nets Model](image-url)
3.1. *call gget*: when a request is made to execute *gget* method then the transition *call for Gget* is enabled. Since it is an AN, it is explored in to the transition *call gget*, where it is guarded with the pre-condition to check if buffer is not empty and also the last method executed is not put (Stated in Figure 2.3.10).

3.2. *gget accepted*: when the pre-condition for method *get2* is evaluated true, a token is passed in this place stating method *get2* is accepted and now can be executed.

3.3. *gget*: it is a transition, which represents body of method *gget* and any post actions. When *gget* is fired *Buffer Capacity* is decremented by 1 and a token goes to place *methods used* representing successful execution of the method *gget*.

### 2.4 State Modification

#### 2.4.1 State Modification Definition

State modification appears as a result of multiple inheritance, that is, it occurs when a sub class is defined by inheritance from two or more superclasses. Anomaly appears as a result to distinguish for the set of states under which the methods from other class can be invoked [CRR].

The standard example involves adding a locking capability to our *Buffer* object defined earlier in 2.2.1. The operation of locking and unlocking is *idempotent*, i.e. if a locked object is locked again, then this second operation will be executed keeping the state of the object same. State Modification arises from the need to add new attributes to distinguish between states, *e.g.*, *locked* attribute in Figure 2.4.1. All states are affected by state-modification [CRR].
2.4.2 Partitioning of the states in Mixed-in Classes

The execution of methods in class Lock modifies the set of states under which the methods inherited from the parent could be invoked as shown in Figure 2.4.2 [MY].

2.4.3 Solution State Modification

The methods of class Buffer have new pre-conditions added to them to ensure that they can only be invoked if the object is unlocked i.e., object of this mixed class (LockableBuffer) will behave like parent Buffer class except that when it receives a lock() message. When LockableBuffer
accepts a \textit{lock()} message, it will not accept any other message except \textit{unlock()} message [FS]. The new preconditions are defined as the conjunctions of the old preconditions and the state variable locked. In this example, the code of methods \textit{put()} and \textit{get()} is not redefined and hence the anomaly is avoided. The precondition block is redefined but that has been done in a systematic, incremental way that allows only few simple operations (like conjunction) on those preconditions.

In the above example, new preconditions involve the state variable \textit{locked}. This means the user needs to know about state variables in class \textit{Lock}. It is not feasible to publicize the names of state variables for two reasons. First, we may have to use executable library files, which will require having their codes available. Second, apart from knowing the names of state variables, when inheriting from a superclass that changes the state variables, we need to understand how the superclass works and how it changes those state variables. Hence the \textit{Lock}, class can not be treated as a black box in this example. This problem can be solved with the use of events notation in much simpler way as shown in Figure 2.4.3.

\begin{verbatim}
class LockableBuffer: Lock, Buffer{
   method put {
      pre: super(Buffer) ^ (event(1)!= Accept(lock))
   }
   method get {
      pre: super(Buffer) ^ (event(1)!= Accept(lock))
   }
}
\end{verbatim}

\textbf{FIGURE 2.4.3 CLASS LOCKABLEBUFFER WITH EVENTS}

Therefore, the state modification can also be solved by using events. Now, the \textit{Lock} class can be treated as a black box. Once an object is locked nothing but the \textit{unlock()} method is acceptable to the object. Therefore, if we try to execute different method while the object is locked, the previously accepted method has to be \textit{lock()}. Hence, the state of the object can be checked just by looking back a single step [CRR].

\textbf{2.4.4 Abstract Node representation for State Modification}

Here, we implemented our solution by systematically and incrementally updating preconditions in the methods of class \textit{Buffer}. 
Places, Transitions and AN, their meaning for class Buffer and Lock, LockableBuffer:

![Diagram of class LockableBuffer hierarchy in colored Petri Nets](image)

**FIGURE 2.4.4 CLASS LOCKABLEBUFFER HIERARCHY IN COLORED PETRI NETS**

The hierarchical representation of class LockableBuffer is as shown in Figure 2.4.4. The class LockableBuffer is inherited from parent class Buffer and parent class Lock (Stated in the diagram as LockandUnlock). If there is a call for the methods in parent class, Buffer then the methods from parent class will be executed if it is not locked along with satisfying other preconditions else if the call is for the method in Lock class then the buffer will be locked or unlocked depending on the method call.

The Figure in 2.4.5 represents the Petri Nets model of the hierarchical LockableBuffer class that is shown in Figure 2.4.4. We implemented our solution by updating preconditions as discussed earlier in section 2.4.3. We have introduced a new variable Locks and in order to consider this variable we have incrementally updated the preconditions of the methods in the parent class and hence anomaly is avoided. This new variable is represented with a place Locks in our Petri Nets model in Figure 2.4.5.

The meaning of all the places and transition of class Buffer will remain the same as described earlier in section 2.2.4 except.

1. **Locks**: the place represents the status of Buffer if the buffer is locked or unlocked. This place is modified by the execution of methods in the class Lock.
2. **call Buffer Methods**: the hierarchical transition will represent the AN representation of class Buffer as described earlier in section 2.2.4 except there is also an input from place Locks. This is done in incremental fashion by adding this place in the Petri Nets model and then drawing the proper input and output arcs.

2.1. **call put**: this transition represents the precondition for method put. When put method call is made then it’ll be checked to see if buffer is not full and also it is not locked, as shown in Figure 2.4.6.
2.2. *call get*: this transition represents the precondition block of method *get*. When *get* method is called it will be checked to see if buffer is not empty and also it is not locked, as shown in Figure 2.4.6.

3. *Call Lock or Unlock*: it is an AN representation for the method *lock* or *unlock*. In our example class *Lock* has methods *lock()* and *unlock()*.

3.1. *call lock or unlock*: when a request is made to execute the method from class *Lock* then the transition *call for lock or unlock* is enabled. If the method call is *Lock* and the buffer is in a state unlocked then it'll be locked and hence a token locked will be passed to...
place \textit{Locks}, else if buffer is already locked then no action will be taken. If the buffer is in locked state and the call for method \textit{unlocked()}, then the place \textit{Locks} gets a token unlocked else no action is taken. (Stated in Figure 2.4.7).

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2.4.7.png}
\caption{Lock and Unlock Methods of Class LockableBuffer in Colored Petri Nets}
\end{figure}
Chapter 3  Formalization of Abstract Node Approach

3.1  Introduction

Object Oriented Methodology can specify concurrency issues of a Concurrent Object Oriented system (as in UML, sequence diagrams, Activity diagrams) but it lacks the verification and validation of the designed systems. Also, to specify different states of the objects in the system and the communication of object with entire system is cumbersome. However, Object Oriented methodologies have the properties such as the modularity, readability, and reusability of the software system. Petri Nets, on the other side, provides the well-defined formalism for verification of the system since, the model is executable and also provides the specification of the concurrency, resource sharing and conflict resolution. In order to take the advantages of both, Object Orientedness and Petri Nets, an approach of Abstract Node is stated to combine both these technologies [MM]. Here, we are describing in detail this Abstract Node approach stated in this [MM] paper.

The approach of Software Development uses the Object Oriented Methodology at the Design stage and the Petri Nets at the Verification and the Validation phase of the software system development as shown in Figure 3.1.1 below.

![Figure 3.1.1 OOD and PN Integration Approach](image)

OOD states the hierarchy of classes and object co-operations and then Petri Nets is used to represent those classes and object interactions. This integration will result in a Petri Nets inside
Object construct, which will allow us to verify and validate the given specified model by simulation. The use of Petri Nets model will allow us, to execute several choices of the same set of conditions and also several different cases concurrently. Also, each object can be created as a Petri Nets model and then can be verified separately and later it can be combined to form the designed system. Hence, the design/verification process has two levels: single object level, object interactions level [MM]. Thus the given designed system can be verified for the various conditions before its implementation.

3.2 Transformation of Object Oriented Designed System to the Abstract Node Model

The following section will present the step-wise approach on how objects are represented in a Petri Nets domain and what kind of abstraction constructs are used to support the above approach.

We assume that the OOD concepts are applied to the given system problem, resulting in a class hierarchy and class interaction model for the system.

OOD identifies the various classes present in the system and the relationships between them. This is achieved by repeating the following steps till the design model is complete [WWW, PR].

1. Identify the basic system requirements
2. Identify the classes (with data and methods)
3. Specify hierarchy of classes
4. State relationship among objects
5. Specify behavior of object

Thus, we can have a Use case diagram and Class diagram to represent the system. The communication among various classes and flow of events in the system is shown by Interaction diagrams, Sequence diagrams, and Activity diagrams [SF].

Then we need to transform Object Oriented designed system to a Petri Nets model. We propose a solution of a unified abstraction construct, Abstract Node that can be both abstract place and abstract transition.
Abstract Node is constructed by connecting AN-place and AN-transition by two arcs in a loop. If arcs 1 and 2 are used then Abstract Node behaves as a place to store information and if arcs 3 and 4 are used then the Abstract Node behaves as a transition that processes data or performs some function. This is shown in figure 3.1.2 [MM]. Abstract Node holds both data or can perform function hence the approach can be related to Object Oriented approach.

The interaction among classes can be shown with the help of input and output places or input and output transitions. Also, the classes belonging to the same hierarchy can be grouped together. Abstract Node encapsulates both data and the methods, hence can be used to represent objects. Objects can be combined to form aggregates and aggregate can be treated as an object. So, object aggregation can be represented with Abstract Node. Hence, the concept of Abstract Node can be used to verify the designed system [MM].

3.3 Steps to transform the Object Oriented Design to the Abstract Node Model

The class diagram and the sequence diagram can be represented with the concept of Abstract Node (AN), where every object will now represent an Abstract Node. Thus, it will represent an AN-interaction diagram. The AN interaction diagram then can be transformed into a Petri Nets model. Thus, at the high level the diagram will be very abstract representing the interaction of various ANs. Objects now are represented by ANs. Unification of abstract place and abstract transition is an Abstract Node. Abstract Node has an input and an output. Abstract Node encapsulates both data and the activity hence it can be used to represent objects [MM]. This is then refined to represent Petri Nets model of the system, which is used for verification and validation of the system.
We can represent following types of transformations of an Abstract Node to a Petri Nets model.

3.3.1 **Input and Output are places for Abstract Node**

The input to AN is from input place. So, there is an input transition within AN, which will transform the data and will put the data in an output place.

Consider the interaction with AN as shown in Figure 3.2.1 below. Input is from input place to AN and output from AN goes to output place. The output transition Transition-2 inside AN, that can be connected to various places outside the AN. Petri Nets model shown with the dotted line between transition-1 and transition-2 is dependent on the functionality of that AN (Figure 3.2.2). Similarly we can propose other possible types of transformations.

![Figure 3.2.1 AN Model](image1)

![Figure 3.2.2 explored AN Model](image2)

**FIGURE 3.2 EXPLORATION OF AN FOR CASE 3.3.1 (SHOWN BY DOTTED LINE)**

3.3.2 **Input is a place and Output is a transition for Abstract Node**

Consider the interaction with AN as shown in Figure 3.3.1 below. Input is from input place to AN and output from AN goes to output transition. Thus, there is an input transition Transition-1 and output place or places inside AN, that can be connected to various transitions outside the AN. Petri Nets model shown within the dotted line represents the various methods accepted by AN (Figure 3.3.2).
3.3.3 **Input is a transition and output is a place to Abstract Node.**

The input to AN is from input transition. So there is an input place within AN, which will store the data. Consider the interaction with AN as shown in Figure 3.4.1 below. Input is from input transition to AN and output from AN goes to output place. Output transition Transition-2 inside AN, can be connected to various places outside the AN. Petri Nets model shown within the dotted line between Place-1 and Transition-2 is dependent on the functionality of that AN (Figure 3.4.2).

**Figure 3.3 Exploration of AN for Case-3.3.2 (shown by dotted line)**

**Figure 3.4 Exploration of AN for Case-3.3.3 (shown by dotted line)**
3.3.4 Input and Output are transitions for Abstract Node

Consider the interaction with AN as shown in Figure 3.5.1 below. Here, Output from AN goes to output transition. The output place is connected to output transition (Figure 3.5.2).

![Figure 3.5.1 AN Model](image1)

![Figure 3.5.2 explored AN Model](image2)

**FIGURE 3.5 EXPLORATION OF AN FOR CASE-3.3.4 (SHOWN BY DOTTED LINE)**

3.3 Further refinements of Abstract Node model

3.4.1 First Level Refinement

An Abstract Node has an input and an output. So, first refinement will split the AN according to type of interaction. It will split abstract place in to two distinct places, message acceptor and message response. A transition can be split to as many transitions as the number of messages object accepts as shown in Figure 3.6. Place stores the data types as many as AN can accept. Here each transition represents one action the object can perform. So depending on type of data, methods are implemented [MM].
3.4.2 Second Level Refinement

Next level of refinement can be achieved by refining the methods. Method provides the set of actions performed by an object. Each method accepts set of arguments or method call in order to execute that method. So method has its implementation depending on set and type of arguments it accepts. It is also possible to have multiple representation of a single method (shown in the Figure 3.7, second implementation is shown by the dotted line). The binding can be static or dynamic and can be shown by arc inscriptions [MM].
Chapter 4  ATM System Design, Verification and Validation

To illustrate our approach of software design and its verification and validation, we considered an Automated Teller Machine (ATM) example. In our model, we are looking for

- Modeling an Object structure with Petri Nets
- Specification of Inheritance
- Identification of Concurrent operations – which operations can be done in concurrent and how will that affect the specified Object Oriented Design
- Simulating Petri Nets model for Verification and Validation of each object and its functionality in the system and also for object interactions together in the system
- Incrementally updating the Petri Nets model if system requirements are altered

The ATM example with its Object Oriented Design is taken from the book [WWW].

4.1  ATM Requirement Specifications

- An Automated Teller Machine (ATM) is a machine through which bank customers can perform a number of the most common financial transactions. The machine consists of a display screen, a bank card reader, numeric and special input keys, a money dispenser slot, a deposit slot and a receipt printer.

- When the machine is idle, a greeting message is displayed. The keys and deposit slot will remain inactive until a bank card has been entered.

- When a bank card is inserted, the card reader attempts to read it. If the card cannot be read, the user is informed that the card is unreadable, and the card is ejected.

- If the card is readable, the user is asked to enter a personal identification number (PIN). The user is given feedback as to the number of digits entered at the numeric keypad, but not the specific digits entered. If the PIN is entered correctly, the user is shown the main menu (described below). Otherwise, the user is given up to two additional chances to enter the PIN correctly. Failure to do so on the third try causes the machine to keep the
bank card. The user can retrieve the card only by dealing directly with the authorized bank employee.

- The main menu contains a list of transactions that can be performed. These transactions are:
  - A deposit funds to an account
  - Withdraw funds from an account
  - Transfer funds from one account to another
  - Query balance of any account

- The user can select a transaction and specify all relevant information. When a transaction has been completed, the system returns the main menu.

- At any time after reaching the main menu and before finishing a transaction (including before selecting a transaction), the user may press the cancel key. The transaction being specified (if there is one) is canceled, the user’s card is returned, the receipt of all transactions is printed, and machine once again becomes idle.

- If a deposit transaction is selected, the user is asked to specify the account to which funds are to be deposited and the amount of the deposit, and is asked to insert deposit envelope.

- If a withdrawal transaction is selected, the user is asked to specify the account from which funds are to be withdrawn and the amount of withdrawal. If the account contains sufficient funds, the funds are given to user through the cash dispenser.

### 4.2 ATM representation in Unified Modeling Language (UML)

For the OOD of ATM given in the book [WWW], we present the various model behavioral properties in UML.

UML supports Object Oriented software development and offers a number of diagram types to model various aspects of the system.
For the given requirements of ATM, following Classes and their responsibilities are identified. Class hierarchy diagram for ATM is shown in Figure 4.2.1. Class Diagram identifies structural and the architectural view of the system.

The collaborations among the objects are also identified in the book. We have listed those collaborations along with the classes, which will send the request (Clients) and the classes, which will fulfill those requests (Servers) for our ready reference.

<table>
<thead>
<tr>
<th>Clients</th>
<th>Contracts</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Balance Inquiry</td>
<td>Access and Modify Account Balance</td>
</tr>
<tr>
<td></td>
<td>Deposit Transaction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Funds Transfer</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Withdrawal Transaction</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Transaction</td>
<td>Commit the results to the database</td>
</tr>
<tr>
<td>3</td>
<td>Form</td>
<td>Display Information</td>
</tr>
<tr>
<td></td>
<td>Menu</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Transaction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>User Message</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Bank Card Reader</td>
<td>Get a numeric value from the user</td>
</tr>
<tr>
<td></td>
<td>Transaction</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Deposit Transaction</td>
<td>Accept input from user</td>
</tr>
<tr>
<td></td>
<td>Form</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Menu</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Transaction</td>
<td></td>
</tr>
<tr>
<td></td>
<td>User Interaction</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Transaction</td>
<td>Get a user selection from a list of options</td>
</tr>
<tr>
<td></td>
<td>ATM</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Withdrawal</td>
<td>Output to the user</td>
</tr>
<tr>
<td></td>
<td>ATM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Transaction</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>ATM</td>
<td>Execute a financial transaction</td>
</tr>
<tr>
<td>9</td>
<td>Bank Card Reader</td>
<td>Display a message and wait for some event</td>
</tr>
<tr>
<td></td>
<td>ATM</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Transaction</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.2.2 Interacting Classes and their Functions**
However, there are following potential drawbacks when we model the system behavior with UML:

- **Lack of executable models** – the UML models is not executable to simulate the behavior of the system. The UML has no formal execution semantics. Colored Petri Nets (CPN) on the other hand has such formal execution in terms of enabling and occurrence rules. Therefore, CPN models allow us to verify the behavior of designed model.

- **Modeling of dependencies** – with UML it is difficult to describe state machines for various classes and also to show the communication behavior of various classes. If we try to show the state diagrams in UML which will show all the possible dependencies then there will be a large number places and model will grow very large. While on the other hand the functional dependencies can be properly modeled in CPN. Also, CPN models can be shown in a good hierarchical manner [JJ].

Next section will present how this is represented in Petri Nets model.

### 4.3 ATM Representation in Colored Petri Nets

For modeling of ATM in Colored Petri Nets (CPN), we are using DESIGN/CPN tool. We modeled the ATM as per the OOD mentioned above. The hierarchical model for ATM is represented in CPN as shown in Figure 4.3. We can see that ATM as AN interacts with Transaction object represented as AN through execute transaction. The inheritance hierarchy of Transaction is shown in the Figure 4.3.
FIGURE 4.3: HIERARCHIES IN ATM
Places and Transitions and their meanings in ATM model:

For all the places, there color sets are defined in the Figure 4.3.1 and the Figure 4.3.4.

1. ATM #1 (refer to Figure 4.3.1B) (Page represents ATM Class and it is interaction with the overall system). The color set for ATM is defined in Figure 4.3.1A.

1.1. Places and their meanings

1.1.1. ATM – (color set – atmstatus), the place ATM represents the state of an ATM machine, whose status can be idle or processing. ATM is in idle state when there is no card inserted in the system, so the only function it does is ‘display greeting’ message. When a valid card is inserted then ATM goes in processing state and performs various other functions from displaying main menu, initiating transactions and some other functions. The color set is atmstatus, to represent status of every ATM in the system. This place is initialized to ‘idle’ state for all ATMs in the system.

1.1.2. User Msg – (color set - atmmsg), the place User Msg represents an object for User Message. It will interact with display screen and it will request this display device to display message by sending a message through display information method call. It has a color set atmmsg, representing different ATMs will be performing different functions hence each ATM will display information as per the functions they are performing. This place appears on different other pages and since it has the same meaning and it has the same color set, it is represented as a fusion place. This will help us in building the net visibly clear and show the global picture of this place with the system.

1.1.3. Display Screen – (color set – atmstr), the place will represent Display Screen, subclass of Display Device in our Class diagram of Figure 4.2.1. The place will simply display on screen the message, which it accepts from User Message. Color set is atmstr, representing message is of type string and different ATMs can display
different messages at the same time. For every ATM in the system, this place is initialized to “Display greeting”. This place is also represented as a fusion place.

```plaintext
color str=string;
color atms=int with 1..10;
color integer=int;
color amount=int;
color status = with idle|processing|error;
color valid = with yes|no;
color atmmsg=product atms*valid*str;
color keystatus = with active|inactive;
color keystate = with pressed|unpressed;
color transaction = with bi|wt|dt|ft;
color atmtransactioncard = product atms*transaction*integer;
color cardpin = product integer*integer;
color account = with checking|savings;
color atmtransactioncardacc = product atms*transaction*integer*account;
color atmtransactioncardacctamt = product atms*transaction*integer*account*amount;
color cardacctamt = product integer*account*amount;
color atmststatus = product atms*status;
color atmvalid = product atms*valid;
color atmstr = product atms*str;

color atminteger = product atms*integer;
color atmkeystatus = product atms*keystatus;
color atmkeystate = product atms*keystate;
color atmcardtran =product atms*integer*transaction;
color atmaccount = product atms*account;
color atmamount = product atms*amount;
color atmtransaction = product atms*transaction;

var um,umi:str;
var i,carddb:integer;
var state:status;
var cl,ckey:valid;
var cancel:keystate;
var t:transaction;
var numkeys:keystatus;
var pindb,ibdb:integer;
var a,accdb,accto,acc2db:account;
var amt,amtdb,amt2db:amount;
var atmnum,atmnum1:atms;
```

**Figure 4.3.1A: Color Set for Automated Teller Machine in Design/CPN**
FIGURE 4.3.1B: AUTOMATED TELLER MACHINE (ATM) WITH HIERARCHICAL TRANSITIONS
1.1.4. **Card slot** – (color set - integer), the place where a user will insert the bankcard. This place is of color set integer. This place can be initialized to any valid integer number. Number of cards processed is dependent on number of ATMs available. This place is represented as a fusion place.

1.1.5. **Valid Card in** – (color set - atmvalid), when a card is inserted then the user will be asked for the PIN (Personal Identification Number), and if it is matches then card inserted is valid card else it is invalid card. The color set atmvalid will represent the card inserted at a particular ATM is valid or invalid. For each ATM, this place is initialized to ‘no’, to say no valid card is inserted.

1.1.6. **Card inserted** – (color set - atminteger), when a bank card is inserted and if it is valid then there will be a token at this place, meaning a card is inserted at this particular ATM.

1.1.7. **Cancel key** – (color set - atmvalid), for each ATM there is a Cancel key (as shown in Class diagram in Figure 4.2.1), which when pressed will abort the current processing and will eject card. The color set atmvalid, meaning for each ATM there is a Cancel key and user can give its input to this place. The input yes at this place for an ATM meaning current processing is to be cancelled else the input is no. For each ATM, this place is initialized to ‘no’, to say there is no cancel input.

1.1.8. **Cancel key status** – (color set - atmkeystatus), the cancel key status will represent if cancel key for every ATM is active or inactive. This place is initialized to ‘inactive’ cancel key for every ATM. When a valid card is inserted then the status of cancel key changes to ‘active’. User can hit the cancel key only if it is active.

1.1.9. **Cancel key state** – (color set - atmkeystate), when a user hits the cancel key then cancel key is ‘pressed’ else it is ‘unpressed’. It is initialized to cancel key ‘unpressed’ for every ATM.
1.1.10. **Transaction type** – (color set - atmtransactioncard), this place will have the information regarding for which ATM, what transaction is asked to do and for which card number.

1.1.11. **Finished transaction** – (color set – atmcardtran), it is a place, which will correspond to the requested transaction is finished. It has the color set atmcardtran, i.e. for ATM, card inserted is and the transaction it executed.

1.2. Transitions and their meanings:

1.2.1. **Display Message** – (corresponds to method Display_message() of class User Message in Class diagram in Figure 4.2.1) ATM requests the User Message to Display messages according to its state. If ATM is idle then it will request User Message to display Greeting else if valid card is inserted and the state is processing then it will request it to display Main Menu. This is a hierarchical substitution transition. It will be expanded to Figure 4.3.3.

1.2.2. **Display Information** – (corresponds to method Display_information() of class Display Device in Class diagram in Figure 4.2.1) when ATM/Bank Card Reader/Transaction sends a request to User Message to display messages then, User Message forwards this request to Display devices to Display appropriate information.

1.2.3. **Read Bank Cards** – (corresponds to method of class Bank Card Reader in Class diagram in Figure 4.2.1) when a card is inserted in the card slot then Bank card reader attempts to read the card and also checks if it is a valid card. This is again a hierarchical substitution transition representing the object Bank Card Reader. This transition will be expanded to Figure 4.3.4.

1.2.4. **Press Cancel key** – When a valid card is inserted then Cancel key becomes active and user can press the cancel key to cancel any ongoing operation.

1.2.5. **Get Transaction info** – (corresponds to method get_user_selection() of class Menu in Class diagram in Figure 4.2.1) When a main menu is displayed on the
display screen then user can enter the choice of transaction. In our model we have asked system to generate any random choice of operation. But, we can also give our input if required.

1.2.6.  *Execute Transaction* – (corresponds to method Execute_financial_trnsation() of class Transaction in Class diagram in Figure 4.2.1) this transition will execute the financial transaction asked by user. This is again a substitution transition representing the class Transaction and will be expanded as in Figure 4.3.8.

1.2.7.  *Cancel All* – when the cancel key is pressed the ATM has to cancel any ongoing operation. Cancel All will terminate all the operations and eject the bank card and print the receipts unless the transaction is modifying the database. If it is modifying the database then after committing the changes it will eject the card. This is a substitution transition as the system can be performing various operations from gathering information for transaction or displaying any messages. So cancel all will have to consider all these scenarios and hence it is represented as a substitution transition expanded to Figure 4.3.2.

2. Cancel all #2 (refer to Figure 4.3.2, After Cancel key is Pressed)

2.1. Places in Cancel all and their meanings:

2.1.1.  *Tranaccount* – (color set - atmtrancardace), the place will store the information for a transaction on an account for a bankcard at an ATM. This place is represented as a fusion place.

2.1.2.  *Card slot* – (color set - atminteger), if the transaction is cancelled then the bankcard will be ejected and a token will represent a bankcard at an ATM.

2.1.3.  *Print Transaction Aborted* – (color set - atmtransaction), if the user presses the cancel key then the withdrawal transaction will be aborted and it’ll be printed at that ATM and for that bankcard as withdrawal transfer transaction is aborted.
FIGURE 4.3.2: AFTER CANCEL KEY IS PRESSED
Transitions:

The following transitions will correspond to the actions which will happen when the user presses cancel key. The cancel key can be pressed at any time, when a valid card is inserted in an ATM. When a user presses cancel key then the only operation possible is cancel the current transaction. When any of the transition is fired to cancel the operation then a user message “Card Removed” will be displayed. Also, the card will be ejected and will be present in card slot. A message will be printed saying transaction is aborted.

2.1.4. *Abort Transaction and Exit Main Menu* – when the user presses the cancel key and an ATM is in a state where user has entered the transaction to execute then this transition gets enabled and it fires.

2.1.5. *Exit Main menu and Return Bank Card* – when an ATM is in a state where user inserts a valid card and presses the cancel key then this transition gets enabled.

2.1.6. *Transaction Aborted in info Gathering* – when an ATM is in state of gathering the information related to the transaction and the user presses the cancel key then this transition gets enabled and it fires.

3. Display messages #3 (refer to Figure 4.3.3)

3.1. Transition:

3.1.1. *Display Greeting or menu* – (corresponds to method Display_greeting_message() and Display_main_menu() of class ATM as in Class diagram in Figure 4.2.1) when an ATM is in idle state and when no valid card is inserted then ATM will request user message to display greeting message “DisplayGreeting”, which it will send it to output device to display it on the display screen. If the machine is in a state of processing and if the valid card is inserted then ATM will request user message to display “DisplayMainMenu” on the display screen.
4. ReadandValidateCards #4 (refer to Figure 4.3.4, It also shows the Local set of Color set)

4.1. Places

4.1.1. Card in – (color set - atmcardvalid), the token at this place will represent the card inserted at an ATM and it is not yet checked for its validity.

4.1.2. BankCard Reader – (color set - valid), represents the Class Bank Card Reader in Class diagram in Figure 4.2.1. Bankcard reader is set to ‘yes’ meaning cards are readable by bankcard reader and ‘no’ meaning cards cannot be read by bankcard reader.

4.1.3. Readable card – (color set - atmcardvalidreadable), A token at this place will represent the card inserted at an ATM, is not yet checked for its validity but it is readable. This place is represented as a fusion place.
FIGURE 4.3.4: READ AND VALIDATE PIN

```
color cardvalid = product integer*valid;
color cardvalidreadable = product integer*valid*valid;
color cardattemptes = product integer*integer;
color cardpinattemptes = product integer*integer*integer;
color atmcardvalid = product
atms*integer*valid;
color atmcardattemptes = product
atms*integer*valid;
color atmcardvalidreadable = product
atms*integer*valid*valid;
color atmcardpinattemptes = product
atms*integer*integer;
var cr, pinstatus:valid;
var inpin, attempt, j:integer;

<<No errors>>
```
4.2. Transitions:

4.2.1. *Insert Card* – bankcard reader will check if user inserts the card. So, if there is card in card slot and if an ATM is available then this transition will get enabled. When it fires, it’ll send a token to a place Card In, suggesting the card is inserted in ATM and is not yet checked for its validity.

4.2.2. *Read Card* – (corresponds to method Read_bank_cards() of class Bank Card Reader in Class diagram in Figure 4.2.1) when there is a card inserted for an ATM, then the bankcard reader checks if the card is readable or unreadable. If the card is readable then there will be a token at Readable card saying, bankcard reader could read the card. This is a substitution transition the functionality of which will be expanded as shown in Figure 4.3.5.

4.2.3. *Check validity of card* – if the card is readable then the user will be asked to input the PIN for the card and the card will be verified and validated for its authentication. The user has 3 attempts to provide the correct PIN for the bankcard inserted at an ATM. If the user provides correct PIN then the cancel key status will be set to active, meaning user can press the cancel key at any time. Also there will be a token at a place card inserted meaning the card is inserted at this ATM. This is a substitution transition and will be expanded to Figure 4.3.6.

5. Unreadable Cards #5 (refer to Figure 4.3.5)

5.1. Places:

5.1.1. *Key Pad* – (color set - atmkeystatus), represents Numeric_key_pad class in Class diagram in Figure 4.2.1. When a card is inserted and if it is readable then the numeric key pad is set to active state, so user can input the information requested with the help of keys as the key pad is active. This place is represented as a fusion place.
5.1.2. *Unreadable Cards* – (color set – atmcardvalidreadable), a token at this place will represent that the bankcard reader is unable to read the card information and hence it will be ejected.

5.1.3. *Card attempt* – (color set – atmcardattempts), will store the information for a bankcard inserted at an ATM, the number of attempts to check the PIN. This place will be set to ‘0’, when a readable card is inserted.

5.2. Transitions:
5.2.1. **Check if cards readable** – bankcard reader checks if the card is readable when the card is inserted at an ATM. If the card is readable then it will set the keypad to ‘active’ state so that user can press the numeric key input if requested. Also it will send a token to a place Readable card meaning the card is readable and it will initialize the PIN attempts for the card to ‘0’. If the card is unreadable then a token will be sent to Unreadable Cards saying the card inserted at ATM is unreadable.

5.2.2. **Eject Cards and Print Receipt and Display Messages** – (corresponds to method Eject_bank_cards() of Bank Card Reader Class, Print_receipt() of class Receipt Printer and Display_messages() of class User Message in Class diagram in Figure 4.2.1. All these three functionalities are combined here.) When the inserted card is unreadable then it will be ejected and the receipt will be printed saying the error in reading the card, also it will send a message “CardUnreadable” to user message to display on the screen.

6. **Validate PIN #6 (refer to Figure 4.3.6)**

6.1. **Places:**

6.1.1. **Card in** – (color set – atmcardattempts), the place will store the bankcard information inserted at an ATM showing number of attempts for the PIN.

6.1.2. **Cards at ATM** – (color set - integer), when the user is unsuccessful in providing valid PIN information for the card then the card is kept at an ATM.

6.1.3. **User Input** – (color set - atminteger), represents to get_numeric_value_from_user() of class Form in class diagram in Figure 4.2.1. When user inserts a card and if the card is readable then the user will be asked to input the PIN. The user response at an ATM is stored here.

6.1.4. **PIN DB** – (color set - cardpin), the place is a database storing the information for each user; its card information and the valid PIN.
6.1.5. *Knows keys pressed* – (color set - atmcardpinattempts), represents know_Key_pressed() of Keypad in class diagram in Figure 4.2.1. When the user responds to the request of input a PIN, then the information for an ATM, for the card where the keys are pressed as PIN and the number of attempts to check PIN is stored here.

6.2. Transitions:

6.2.1. *Prompt PIN* – (corresponds to method Prompt PIN() of class Bank Card Reader in Class diagram in Figure 4.2.1) if the card is readable and the number of attempts for the PIN are not more than 3, then the bankcard reader will prompt user for PIN. When this transition fires then “Input PIN” message will be sent to user message to display on screen. Also the keypad will be set to active state and the token will be sent to place card in.

6.2.2. *Display Information* – (corresponds to method Display_information() of class Display Device in Class diagram in Figure 4.2.1) user message requests output device to display information (“Input PIN”) on display screen.

6.2.3. *Accept User Input* – (corresponds to method Accept_user_input() of class Input Device in Class diagram in Figure 4.2.1) When a message “Input PIN” is displayed on the screen then the user can provide the input for PIN by using numeric keypad and when the user provides this user input then the Accept user input transition gets enabled. When this transition is fired it will send a message “checking PIN” to display on screen and it’ll store the keys pressed by user.

6.2.4. *Checking PIN* – the user input for the PIN for a card in an ATM, will be checked against the PIN DB, to verify if card is valid or not. This is also a substitution transition and will be expanded as shown in Figure 4.3.7.
7. Invalid PIN #7 (refer to Figure 4.3.7)

7.1. Places:

7.1.1. Valid PIN – (color set - atmcardvalid), the token at this place will represent whether the user has entered the valid PIN or not. The place will represent for an ATM, the card inserted is having ‘yes’ a valid PIN or ‘no’ not a valid PIN.

7.2. Transitions:

7.2.1. Check PIN – the user input for the PIN is checked against the PIN DB to check if correct PIN is entered. It will check for the same card number in an ATM and in database if the PIN is the same. If it matches then it will send a token to place called Valid PIN saying card inserted in an ATM is valid.

7.2.2. Invalid PIN – the transition will fire if the for the same card in an ATM and in database, their PINs do not match. When this transition fires then it will increment the number of attempts for entering the valid PIN by 1, and it will send a token to a place Valid PIN saying card inserted is not a valid card.

7.2.3. Validate PIN – when there is a token in valid PIN place and cancel key status then validate pin transition will fire. If the PIN entered by user is a correct PIN then the cancel key status will be set to active else inactive. Also it will send a token to a place Readable card, card inserted and Valid card in if the PIN is correct.
Figure 4.3.7: invalid pin
8. Execute Transaction #8 (refer to Figure 4.3.8) (represents Class Transaction in our Class Diagram shown in Figure 4.2.1)

8.1. Places:

8.1.1. *User acc choice* – (color set - atmaccount), represents get_user_selection() of class Menu in class diagram in Figure 4.2.1. When the user selects a transaction from the main menu displayed on the system, then the user will be asked the account choice for that transaction at an ATM. If the transaction is balance inquiry then the balance at this account will be shown, if the transaction is withdrawal transaction then the amount will be withdrawn from this account, if the transaction is funds transfer then this account will be treated as an account from which to transfer the money, if the transaction is deposit transaction then the money will be deposited at this account.

8.2. Transitions:

8.2.1. *Prompt account* – (corresponds to method Prompt_account() and gather_information() of class Transaction in Class diagram in Figure 4.2.1) when the user selects a transaction to execute from the main menu, then the user will be prompted to enter the account information. This account will have different meanings depending on the type of transaction. This transition is fired only if cancel key is in state unpressed. When this transition is fired then the token will be in a place tranaccount, having the user input for the account for the transaction for the card inserted at ATM.

8.2.2. When the transaction (any of the below), are executed successfully then a token will be sent to Valid card in and card inserted for further requests of any transactions. Also a token will go in a place finished transaction, which can be printed later on.
FIGURE 4.3.8: EXECUTE TRANSACTION
8.2.2.1. *Get info Balance Inq and Commit Changes* – (corresponds to class Balance Inquiry in Class diagram in Figure 4.2.1) the transition fires only if user selects the transaction Balance Inquiry and cancel key is not pressed. The functionality of this transition will be expanded as in Figure 4.3.10.

8.2.2.2. *Get info Funds transfer and commit changes* – (corresponds to class Funds Transfer in Class diagram in Figure 4.2.1) the transition fires only if user selects the transaction Funds transfer and cancel key is in state unpressed. The functionality of this transition will be expanded as in Figure 4.3.12.

8.2.2.3. *Get info Deposit tran and commit changes* – (corresponds to class Deposit Transaction in Class diagram in Figure 4.2.1) the transition fires only if user selects the transaction Deposit funds and cancel key is in state unpressed. The functionality of this transition will be expanded as in Figure 4.3.11.

8.2.2.4. *Get info for Withdrawal transaction and commit changes* – (corresponds to class Withdrawal Transaction in Class diagram in Figure 4.2.1) the transition fires only if user selects the transaction type withdrawal and cancel key is in state unpressed. The functionality of this transition will be expanded as in Figure 4.3.9

9. Withdrawal Transaction #9 (refer to Figure 4.3.9)

9.1. Places:

9.1.1. *Amount of tran* – (color set - atmamount), when a withdrawal transaction is selected then the user will be asked to the amount to withdrawn. This place will store the user input for the amount at an ATM.

9.1.2. *Alltraninfo* – (color set – atmtrancardaccamt), the place will store all the information required for the withdrawal transaction. The place will store for an
ATM, the transaction is withdrawal transaction on the card for the account and the amount to be withdrawn.

9.1.3. *Account database* – (color set – cardaccamt), corresponds to class Account in class diagram in Figure 4.2.1. This place knows the users of system. This place represents Class Account. The place will store the valid card numbers, there accounts with the bank and the amount present for that account. This is a fusion place.

9.1.4. *Cash withdrawn* – (color set - atmamount), this place will act as a cash dispenser, where the cash amount will be stored at an ATM where the withdrawal transaction is executed successfully.

9.1.5. *Print transaction aborted* – (color set - atmcardtran), if the user presses the cancel key then the withdrawal transaction will be aborted and it’ll be printed at that ATM and for that bankcard as withdrawal transfer transaction is aborted. This is a fusion place.

9.1.6. *Cardslot* – (color set - atminteger), if the transaction is cancelled then the bankcard will be ejected and a token will represent a bankcard at an ATM.

9.2. Transitions:

9.2.1. *Prompt Amount* – (corresponds to method prompt_for_amount() of class Withdrawal Transaction in class diagram in Figure 4.2.1.) When a token is in tranaccount place and call is for withdrawal transaction and the cancel key is in unpressed state then the user will be prompted for the amount to be withdrawn. When this transition fires then a token will be present in alltraninfo place representing all the information required for executing withdrawal transaction is present here.

9.2.2. *Withdraw funds* – (corresponds to method withdraw_funds() of class Withdrawal Transaction and commit_changes_to_database() in class Account in class diagram in Figure 4.2.1.) when a token is present in a place alltraninfo and the cancel key is
unpressed then the withdraw funds will check if the account exists for the card and the amount present at that account is more than amount requested to withdraw if this case is true then it’ll execute the transaction by making proper changes in the account database and will also dispense the money in cash withdrawn. When this transition fires that is when this transaction is executed then a token goes in a place finished transaction that can be printed later. Also token goes to place valid card in and card inserted so that card will be available for any further transactions.

9.2.3.  *Ask to redo tran* – if the amount to withdraw from an account is more than the actual amount present for that account and the cancel key is unpressed then the user will be asked to redo the transaction. When this transition fires it’ll display a message “Not Enough Amt in Acc”, on display screen.

9.2.4.  *Cancel withdraw transaction* – if the user presses cancel key and if withdraw transaction is not committing changes to database then the message transaction aborted is printed and the user message “Card Removed” will be displayed and the card will be ejected and will be put in card slot.
FIGURE 4.3.9: WITHDRAWAL TRANSACTION
10. Access Balance Inquiry #10 (refer to Figure 4.3.10)

10.1. Places:

10.1.1. Balance – (color set – amount), when the user selects the balance inquiry transaction and if the transaction is executed successfully then this place will show the current balance at an account selected by the user.

10.2. Transitions:

10.2.1. Access Balance – (corresponds to method access_balance() of class Balance Inquiry in class diagram in Figure 4.2.1.) When the transaction selected in balance inquiry and the cancel key is not pressed then the account balance for the account provided by the user will be printed or displayed on screen. It is checked to see if account exists for that card.

![Figure 4.3.10: Access Balance Inquiry](image-url)
11. Deposit Transfer #11 (refer to Figure 4.3.11)

11.1. Places:

11.1.1. *Deposit Amt* – (color set - atminteger), represents Deposit slot in class diagram in Figure 4.2.1. When the deposit transaction is selected by the user then the user will be asked for an amount to deposit. Since this input is for that ATM, where the user selects the deposit transfer account, the color set is atminteger, representing for an ATM, the amount to be deposited is this number. The place will represent Deposit Slot.

11.2. Transitions:

11.2.1. *Accept deposit amount* – (corresponds to method deposit_funds() of class Deposit Transaction and commit_changes_to_database() in class Account in class diagram in Figure 4.2.1.) If the user choice for the transaction is deposit transfer and the cancel key is not pressed then the user will be asked to enter the amount greater than ‘0’ to be deposited that can be put in a deposit slot. When the user put the money in deposit slot then it will be added to the account provided by the user.
FIGURE 4.3.11: DEPOSIT TRANSFER TRANSACTION
12. Funds Transfer #12 (refer to Figure 4.3.12)

12.1. Places:

12.1.1. *Transfer to acc* – (color set - atmaccount), when the funds transfer transaction is selected, then the user will be asked the account to which to transfer the money. This place has will store the user input of the account type for the ATM, where that user inserts the bankcard.

12.1.2. *Acc from* – (color set - atmtrancardacc), this place will store the information associated with for the ATM, the transaction type is funds transfer, for the card is inserted, and from which account the money is to be transferred.

12.1.3. *Acc to* – (color set - atmtrancardacc), this place will store the information associated with for the ATM, the transaction type is funds transfer on the card and to which account money will be transferred to.

12.1.4. *Valid Acc from* – (color set - atmtrancardacc), this place will check if user has the valid account from which user has to transfer money.

12.1.5. *Valid Acc to* – (color set - atmtrancardacc), this place will check if user has the valid account to which user has to send money.

12.1.6. *Amt to transfer* – (color set - atminteger), this place will store the input for the amount to transfer during funds transfer transaction at an ATM.

12.1.7. *Print aborted transaction* – (color set - atmcardtran), if the user presses the cancel key then funds transfer transaction will be aborted and then it will be printed at that ATM and for that bankcard as funds transfer transaction is aborted.

12.1.8. *Card slot* – (color set - atminteger), if the transaction is cancelled then the bankcard will be ejected and a token will represent a bankcard at an ATM.
FIGURE 4.3.12: FUNDS TRANSFER TRANSACTION
12.2. Transitions:

12.2.1. *Input transfer to account* – when the choice for the transaction is funds transfer then the user will be asked to enter an account to which to transfer the money. If the cancel key is not pressed and if there is information available for account from and account to then this transition fires. When this transition fires there is a token at a place account to and account from to transfer money.

12.2.2. *Check Acc if exists* – when there is a token at account from and account to which to transfer the money and cancel key is not pressed, then this account information will be checked against the database to see if these are the valid accounts for the card inserted.

12.2.3. *Ask amount to transfer and commit changes* – (corresponds to the Transfer_funds() of class Funds Transfer and commit_changes_to_database() in class Account in the class diagram in Figure 4.2.1) if the user has entered the valid account information to transfer the money then the user will be prompted to input the amount to transfer if the cancel key is ‘unpressed’. If the amount to transfer is less than amount present in the account from which to transfer the money then the transaction will be executed successfully by decrementing the amount in the account from which user wanted to transfer money by amount entered and by incrementing the amount at account to by that same amount.

12.2.4. *Abort funds transfer* – if the cancel key is pressed before committing the changes to the database then the transaction funds transfer is aborted. A message will be displayed saying transaction is aborted and the card will be ejected, also the receipt of this aborted transaction will be printed.

12.2.5. *Cancel funds transfer* – if the cancel key is pressed and funds transfer transaction is still in the phase of gathering account information then the card will be ejected and the receipt of this cancelled transaction will be printed and a user message saying transaction is aborted is displayed.
Conclusions

Object Oriented Methodologies are useful because it provide inheritance, polymorphism and encapsulation. Petri Nets is a formal mathematical modeling technique. Petri Nets model are executable. Petri Nets provide graphical presentation, and concurrency can be modeled easily in Petri Nets. Hence, incorporating Object Oriented Design with Petri Nets has advantages of both. Based on the ATM example we can say that it is possible to do effectively integration of both these methodologies.

One of the advantages of using Abstract Node is that it is constructed using Colored Petri Nets, hence the level of abstractions can be varied simply by changing the input and output arcs to Abstract Node and also it can be done by changing the color of Abstract Node Place. Objects and Object interactions can be easily modeled with Abstract Node approach. Inheritance is shown as the hierarchical structure within the AN model.

We can also state that it is possible to incrementally update the system with AN approach and so Inheritance Anomaly can be solved. AN and hence Petri Nets models can be incrementally updated by addition of places and transitions in the system and then correctly assigning the interaction of this newly added place or transition with the entire model.

Also, modeling can be done in top-down fashion to represent higher level abstractions or it can be done in the ‘bottom-up’ way to show lower level interactions among objects. Model checking is possible in both ways.

Verification and Validation of designed systems by Abstract Node approach gives a formal characterization for the objects that enhances reuse. Models can be made more efficient by identifying concurrency which is not obvious during OOD.

Object Oriented tools like UML allows automatic code generation, that helps in software development process. To handle highly distributed and parallel systems and to ensure their correctness, correct behavior and dependencies, CPN is an excellent supplement to UML based software development. We can use the existing tools like DESIGN/CPN.
References


