IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Vol.-7, Issue-5, May 2019 E-ISSN: 2347-2693

Fault Modelling of an Object-Oriented System using CPN

Shubham Kaushik[1], Ratneshwer[2*]

[1]University Institute of Engineering & Technology, M.D. University, Rohtak, India
[2]SC&SS, Jawaharlal Nehru University, New Delhi, India

Corresponding Author: ratnesh@main.jnu.ac.in

DOI: https://doi.org/10.26438/ijcse/v7i5.18281845 | Available online at: www.ijcseonline.org

Accepted: 26/May/2019, Published: 31/May/2019

***Abstract*—**Object-oriented development is a mechanism in which objects provide services to other objects by various means like inheritance, polymorphism, etc. Faults, in object-oriented software, may occur at two levels i.e. object level and interaction level (when one object provides/receives some services from others). A formal representation, of an object-oriented system, may be helpful to understand the behavior of software faults. Faults identification, at earlier stages, may help during the development and testing stages. In this paper, an attempt has been made to model several faults, in an object-oriented system, with the help of Colored Petri Nets. First, a formal representation of object-oriented properties is depicted by Colored Petri Nets. Secondly, various possible faults are modeled using different programming scenarios. The main emphasis was on faults that may arise due to objects and their interactions i.e. inheritance and polymorphism state. Such information may be useful during the testing and maintenance phases of software development.

***Keywords*—**Object, Faults, Colored Petri Nets, Inheritance, Polymorphism

I. INTRODUCTION

Object-oriented software development is a popular approach by having the property of developing software with the help of objects. This modular way of development enhances the quality of the system and promotes the re-usability of objects across the systems. Objects, in an Object-Oriented System (OOS), may provide/receive services by several means like inheritance, polymorphism, etc. The quality of an OOS depends on the quality of an individual object and how effectively one object interacts with other objects. The behavioral analysis of objects and their interactions may be helpful in fault identification. A fault is an abnormal activity that if not handled properly may lead to failure of the system. Faults may occur due to design flaw or programmer's mistake. If a failure occurs then the system may not be able to achieve its intended functionality. If faults may be diagnosed at an early stage and proper repair mechanism have been taken, then the quality of the system will enhance. The main intention of testing is to spot faults and failures that occurred in development and to guarantee that software is bug-free **[1]**. Automated software testing's the finest way to increase the effectiveness, efficiency and coverage of software testing **[2]**. There are several approaches available for modeling faults in an OOS. Many approaches are related to the prediction of faulty objects using object-oriented metrics **([3], [4], [5]).** Several approaches are based on coupling and cohesion studies of OOS **([3], [6], [7]).** There are some studies using Petri Nets **([8], [9]**) but only a few studies are available related to fault modeling of OOS using Petri nets. Most of the researchers have emphasized on the modeling of dependency analysis (using cohesion/coupling) and fault prediction using OOS metrics rather than how faults can be diagnosed in the OOS. Our approach contributes to the modeling of fault analysis of OOS in case of a faulty situation. We have modeled different scenarios of fault diagnosis approaches.

Petri Net (PN) is a basic modeling tool for parallel and distributed systems. It was originated from Carl Adam Petri's dissertation in 1962 for the purpose of describing chemical processes **[10].** Nowadays, Petri Net is grasping the popularity in modeling the concurrent, parallel and dynamic systems. A normal Petri Net would not be able to model in different cases like while a node only holds the same kind of items or tokens. The time-related dependency of the system also cannot be shown by simple Petri Net. Thus, there are many extensions of Petri Nets are created such as Colored Petri Nets [11]**,** Timed Petri Nets **[12]**, Stochastic Petri Nets **[13]**, Labeled Petri Nets [10], state-charts **[14]**, hierarchical state machines **[15],** etc.

In this paper, we adopt Colored Petri-Nets (CPN) to model the faults in OOS. Inheritance and polymorphism are two important property of an OOS. Objects that are involved in inheritance and polymorphism are more error-prone. Mainly inheritance and polymorphism related faults, along with their sub-types, are covered under discussion. Different programming scenarios have been taken to demonstrate various fault in OOS. Different OOS properties are used for fault modeling. This approach enhances the understanding of

© 2019, IJCSE All Rights Reserved

faults in the aspect of OOS and will be helpful in the proposition of newer fault recovery mechanisms.

The rest of paper is organized as follows: Section 2 briefly point out the work related to fault modeling of OOS. Section 3 introduces the *CPN* representations of *OOP's* concepts. Some faults are described in Section 4 which may occur commonly during the use of *OOP's* concepts. Section 5 contains the fault modeling of OOS using CPN. Section 6 Conclude the work.

II. RELATED WORK

In this section, some related work in the field of fault handling of OOS is discussed. The importance of Colored Petri Nets (CPN) as modeling tool has been recognized by many researchers. Extensions on Petri Nets to fault modeling of OOS are observed to be an important approach. Some properties of Petri nets like its graphical ability and formal representation of the system make it applicable to the researchers and professionals.

Some relevant contributions are as follows Marcus, Poshyvanyk and Forence **[3]** proposed a new measure for the cohesion of classes in OO software systems based on the analysis of the unstructured information embedded in the source code, such as comments and identifiers. A large case study on three open source software systems is presented which compares the new measure with an extensive set of existing metrics and uses them to construct models that predict software faults. Rajkumar, Viji and Duraisamy **[16]** performed a survey of various fault prediction techniques and measuring quality parameters object-oriented systems. Their survey includes traditional techniques like Fault tree analysis, Information theoretic approach, coupling & cohesion measurement, and conceptual cohesion and coupling. Offutt **[17]** presented a model for the appearance and realization of object-oriented faults and discussed specific categories of inheritance and polymorphic faults. As per the authors, the model and categories can be used to support empirical investigations of object-oriented testing techniques, to inspire further research into object-oriented testing and analysis, and to help improve design and development of object-oriented software. Buzato, Rubira and Lisboa **[18]** proposed a new object-oriented reflective software architecture for developing fault-tolerant applications using meta-level programming. Their reflective architecture is composed of three levels and is based on the abstraction of object groups. Aggarwal et al **[19]** have empirically explored the relationship between object-oriented design metrics and fault-proneness of object-oriented system classes. Their study used data collected from Java applications is containing 136 classes. They have used a set of twenty-six design metrics in their work. Result of their study shows that many metrics are based on comparable ideas and provide redundant information. Our approach presents the OOS faults and their modeling using CPN. Researchers and practitioners can get the conceptual view of inheritance and polymorphism related faults. Some features are also extracted to demonstrate the faulty situation in OOS. We extended the above contributions further by presenting a CPN based fault diagnosis method for OOS.

III. CPN REPRESENTATION OF OOP'S CONCEPT

Formal representation provides the behavioral aspects of modules. In this section, the representation of *OOP's* concepts is shown with Colored Petri Nets. The *CPN* tools are used to represent the *OOP's* concept. In CPN representation, *places*, *transitions* and *arcs* components like *set-subpage*, *input-port*, *output-port*, *succession-constraint*, *fusion-sets* are used to represent the *OOP's* concept.
In the following subsections, *CPN* representation of several of OOP's concepts is shown below.

5.1 CLASS VARIABLES AND METHODS

A *class* is a user-defined blueprint or prototype from which *objects* are created. It represents the set of *properties* or *methods* that are common to all *objects* of one type **[20]**. In *CPN tool* the *'places', 'transitions'* with *'arcs'* between them are used to represent a full *class* (which have *variables* and *methods*). *The object* is a basic unit of *Object Oriented Programming* and represents the real-life entities **[20]**. The *Object* is represented by the upper *inscription* of a *place* as shown below. A simple *class* can be represented by a single *place*. Figure 3-1 depicts the general representation of simple *class* representation.

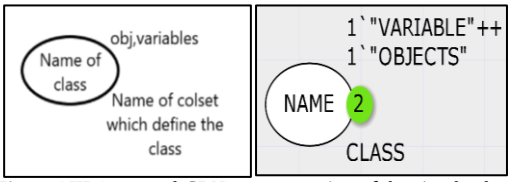


Figure 3III-1 general CPN representation of the simple class

The upper *inscription* of *place* depends on the lower *inscription* of it. The lower *inscription* is the name of the *color set* which can be defined using '*colset' keyword* of *CPN*. *Color set* is the full form of '*colset' keyword*. *Color set* defines the type of *tokens* a *place* can hold. To hold multiple types of *tokens* '*product' keyword* is used in '*colset'* declaration as shown below.

```
colset STRING = string;  //simple color set declaration
colset NAME_OF_COLSET = product datatype 1 * datatype 2 *……*datatype n;  //product colset
```

The *CPN* representation may vary from program to program. The *class* of *java* program which does not have any *method* in it can be represented by only one *place* and its *inscription* but the *class* of *java* program which have *methods* into it can be

© 2019, IJCSE All Rights Reserved

represented with multiple *places* and *transitions*. This may be observed in another concept of *OOP* like *inheritance* and *Polymorphism*.

5.2 INHERITANCE

Inheritance is an im*portant* part of *OOP's* in which the *object* of *child class* can obtain all the properties and behavior of *parent class*. In *Java*, the *''extends'' keyword* is used to inherit the properties and behavior of *parent class* into *child class*. In *CPN* '*input/output ports'* and the '*set-subpage'* tool, are used to represent the *Inheritance* property. In Java *inheritance* is of three types. The *CPN* representations of different types of *inheritance* are given below.

3.2.1 SINGLE INHERITANCE

It is a type of *inheritance* in which a single *'class A'* inherits another single *'class B'*, it means a *class* can inherit the properties of another *class* which must be the *superclass*. In *CPN* a couple of *'pages'* are used to represent the *single inheritance*. On every page, a single *class* can be represented by its *variables* and *methods*. As all *Java* Applications begins execution by *CALLING 'main ()' function* **[21]**. The general representation of *Single inheritance* is shown in Figure 3-2 where two *classes class A* and *class B* are shown and *class B 'extends' class A* using *'set-subpage'* tool of *Hierarchy Palette* of *CPN*. On another page *class A* is represented with *input* and *output ports* which means the *input* on this *class* came from *input port place* and the *output* will go through the *output port place*.

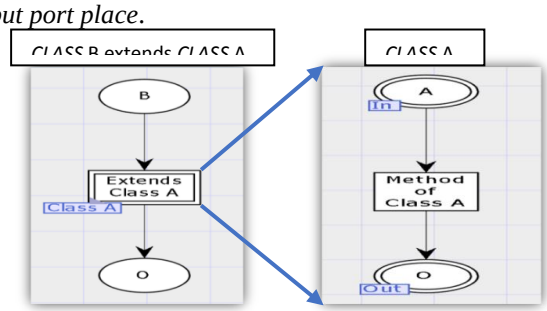


Figure 3-2 general CPN representation of Single inheritance

This representation is used in every type of *inheritance* representation. The concept behind setting the *subpage* for the inherited *class* is that when the *main class* creates an *object* for *child class* then internally the *parent class* is called first by the *child class* and also the *default constructor* of *parent class* executes first before the *child class constructor* execution. If the programmer does not create a *constructor* in *parent class* the *JVM (java virtual machine)* itself creates an empty default *constructor* and executes it.

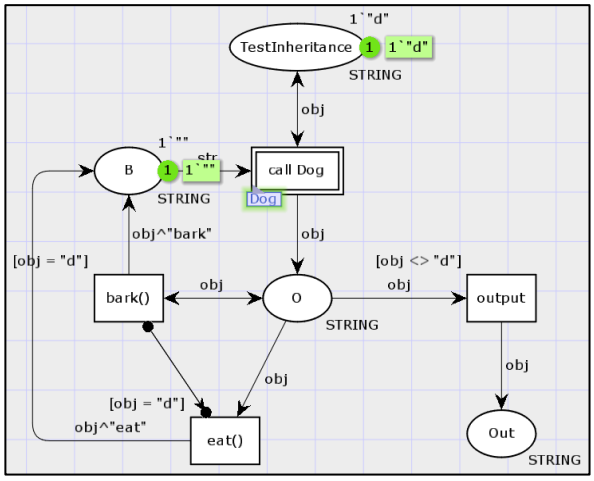
An example of a java program for *single inheritance* is taken below.

Example: Java Program of Single Inheritance **[22]**

```
Output:barking... Eating...
class Animal{
void eat(){      System.out.println("eating...");   } }
class Dog extends Animal{
void bark(){      System.out.println("barking...");    } }
class TestInheritance{
 public static void main(String args[]){
 Dog d=new Dog();    d.bark();
 d.eat(); } }
```

The *CPN* representation, for the above program of *single inheritance*, consists of three pages with one *class* on every page. The *set subpage* tool is used to connect these pages. The *place* which represents the *main class* contains the tokens of *main class* instant *variables* and *objects*.

The *CPN* representation of *main class 'TestInheritance'* has two *String* type tokens one is *"d"* and another is an empty token. The token *"d"* is the *object* of *Dog class* which is created in the *main class 'TestInheritance'*. It also represents two *methods* which are represented with the help of *transitions* as shown. The *methods* which are called from the *main class* are represented by *transitions '(bark() and eat())'* and the *arcs* are connected to *'call Dog' transition* which transfers the tokens to another *class*. The *classes* on another page are set with *input and output ports*, where the *input port* is used to *input* the tokens from another page and *output port* is used to transfer the tokens from the current page to another page. The *colset* declaration for a *place* of *output port* must be same as the *input port* otherwise *CPN* will throw an error for type mismatch. Below is the *CPN* representation of *main class TestIheritance*.

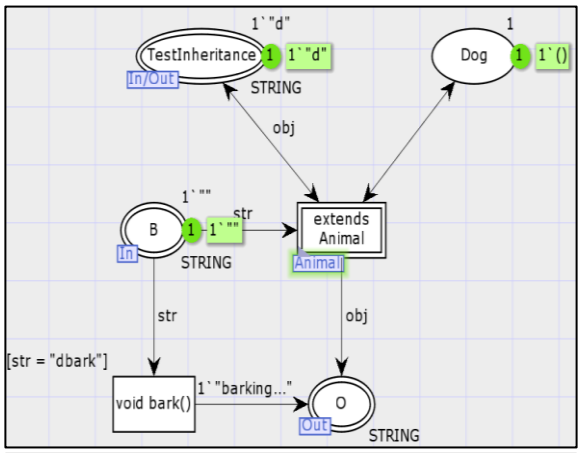



```
class TestInheritance{
        public static void main(String args[]){
        Dog d = new Dog();
        d.bark();
        d.eat();
        }}
```

Figure 3-3 class TestInheritance CPN representation of example java program for single inheritance

© 2019, IJCSE All Rights Reserved

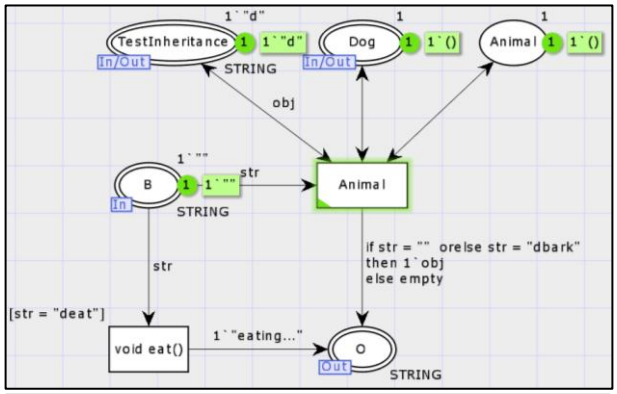
Now, other *classes* are represented on other pages. The *Dog class* is the *child class* which is represented by a *unit colset* as there are no instant data variables in the *Dog class*. The *bark() method* of *Dog class* is also defined on this page with the help of extra *place* and *transitions*. It also *''extends''* *Animal class* which is designed on another page and represented by the *inscribed transition*. The *CPN* representation of the *Dog class* is shown in Figure 3-4.




```
class Dog extends Animal{
        void bark(){
        System.out.println("barking...");
        }
```

Figure 3-4 CPN representation of class Dog of example for Single inheritance

When the *object* of *Dog class* is created inside the *main class TestInheritance*, the *Dog class* is loaded by the *JVM* so in *CPN* representation the *object* token *''d''* is fired on the *main class* first, after this using that *object* token *main class* calls the method of *Dog class* and its *parent class*. The *CPN* representation of *parent class Animal* is shown in below Figure.




```
class Animal{
        void eat(){
        System.out.println("eating...");
        }
```

Figure 3-5 CPN representation of Animal class which is the superclass of Single inheritance example

When *Dog class object* has created this *class* also loads automatically because of the *inheritance*. The *eat() method* is also represented in *Animal class* using *place* and *transition*. There is no *inscribed transition* in the *Animal class* which means it does not inherit any *class*. When the *bark() method* executes it statements the *transition* of *eating ()* method becomes active. The *output*s are fired on another *place* named as out through *output transition*.

3.2.2 MULTILEVEL INHERITANCE

When a *class ''extends''* a *class*, which *''extends''* another *class* then this is called *multilevel inheritance* **[23]**. So one *class* inherit a *class* which already *''extends''* some another *class*. In Java one *class* never inherit multiple *classes* as multiple *inheritances* are not allowed. A level by level *inheritance* of multiple *classes* is known as *multilevel inheritance*. In *CPN* the *multilevel inheritance* is represented by using multiple subpages as used in *single inheritance* and connect them with the help of hierarchy palette tools '*set subpage*' and *input* and *output ports*. It is just extending the *single inheritance* representation by using multiple subpages with the same concept of *single-level inheritance*. The general *CPN* representation of *Multilevel Inheritance* is shown in Figure 3-6.

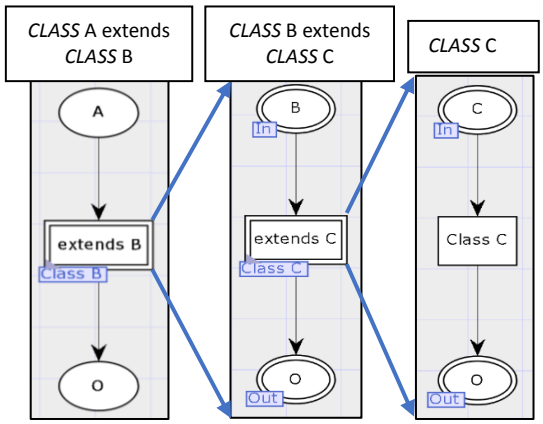


Figure 3-6 general CPN representation of multilevel inheritance

Here we have three *classes* in which *class* C is the *superclass* and also the *parent class* of *class* B. The next level contains *class* B which is the *superclass* for *class* A and also the *parent class* of *class* A. An example of *Multilevel Inheritance* is taken in which three *classes class Car*, *class Maruti*, and *class Maruti800* are defined. The *class Car* is the *superclass* and its *child class* is *class Maruti* which is the *parent class* of *class Maruti800*.

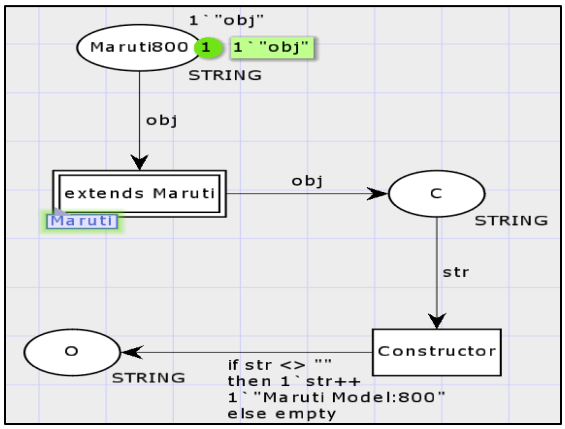
© 2019, IJCSE All Rights Reserved

Example: Program of *Multilevel Inheritance* [23]

Output: *Class* Car *Class* Maruti Maruti Model : 800

```
class Car{
  public Car(){
System.out.println("Class Car");
  }
}
class Maruti extends Car{
  public Maruti(){
System.out.println("Class Maruti");
  }
}
public class Maruti800 extends Maruti{
  public Maruti800(){
System.out.println("Maruti Model: 800");
  }
  public static void main(String args[]){
 Maruti800 obj=new Maruti800();
  }
}
```

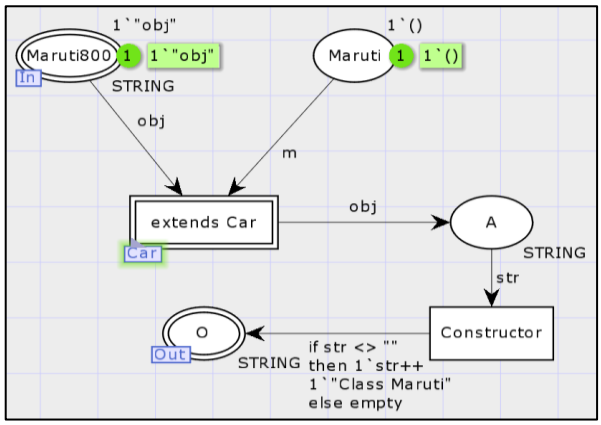
The *CPN* representation of this program consists of three pages as we have three *classes*. The *CPN* representation of *class Maruti800* is shown below.




```
public class Maruti800 extends Maruti{
        public Maruti800(){
        System.out.println("Maruti Model: 800"); }
        public static void main(String args[]){
        Maruti800 obj=new Maruti800();
}}
```

Figure 3-7 CPN representation of class Maruti800 of Multilevel Inheritance

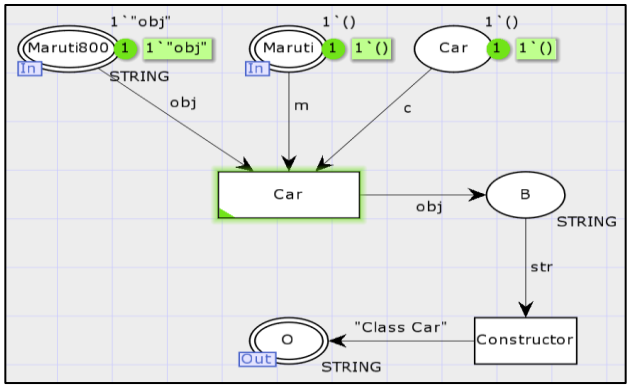
The *Maruti800 class* has one token for *object 'obj'* at *place Maruti800*. The *constructor* for this *class* is also represented with the extra *place* and *transition*. As the *constructor* of *child class* always executes after the execution of *parent class constructor* so the token initially fired to another page of *class Maruti* which is the *parent class* of *class Maruti800*. The *CPN* representation of *Maruti class* is shown in Figure 3-8 *Maruti class* representation.




```
class Maruti extends Car{
        public Maruti(){
         System.out.println("Class Maruti");   }}
```

Figure 3-8 CPN representation of Maruti class of Multilevel Inheritance

Maruti class has no variable and *object* declaration in so we the unit (empty) token at *Maruti place*. The *constructor* for this *class* is also represented by extra *place* and *transition*. Now *Maruti class 'extends' class Car* so *Maruti class* make an internal call to its *parent class* and *class Car* default *constructor* will execute first, so the representation of *class Car* is shown in Figure 3-9 (*CPN* representation of *Car class*).




```
class Car{public Car(){
        System.out.println("Class Car");
        }}
```

Figure 3-9 CPN representation of class Car of Multilevel Inheritance

The *Car class* is also represented with the unit token as there is no variables or *objects* declared in this *class* and the *constructor* is represented by *place* and *transition*. Now when *Maruti800 class* creates an *object 'obj'* as a token it will automatically make a call to its *parent class*. When *class Maruti* is called it makes a call to its *parent class* i.e. *class Car*. Also, there is no *inscribed transition* on the page of *class Car* its means there is no more *inheritance* from *class Car*, so now the default *constructor* starts invoking one by one from *parent class* to *child class*. When the tokens are

© 2019, IJCSE All Rights Reserved

received by *class Maruti800* the default *constructor* of this *class* will execute its print statement and the *output* will be "*Class Car*", "*Class Maruti*" & "*Maruti* Model: 800".

3.2.3 HIERARCHICAL INHERITANCE

When more than one *classes* inherit the same *class* then this is called *hierarchical inheritance* **[23]**. This is a type of *inheritance* in which same *class* is inherited by more than one *class* and forms a hierarchy. In *CPN* the same concept of *single inheritance* is used to model the *hierarchical inheritance* i.e. using '*set subpage*' tool and *input, output ports* of the *hierarchical* palette. In *CPN* one page is connected to the multiple pages for *hierarchical inheritance* as shown in Figure 3-10 general representation of *hierarchical inheritance*.

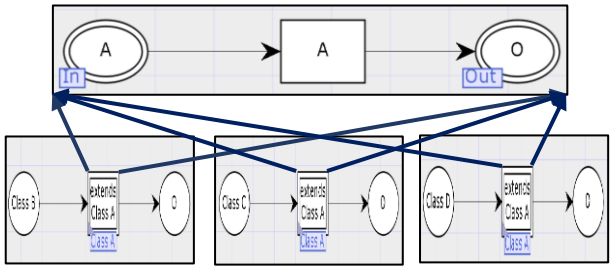


Figure 3-10 general representation of hierarchical inheritance through CPN

Here we have four *classes* in which one is the *superclass* of another three *classes*. The three *classes* '*class* B', '*class* C' and '*class* D' *'extends'* the '*class* A'. So *class* A is the *superclass* and *class* B, *class* C, and *class* D is the *child classes* of *class* A. Every *child class* has *inscribed transition* into it which shows *inheritance* with *class* A. The *superclass* is inherited to multiple pages using '*set subpage*' tool and *CPN* make multiple copies of *superclass* to connect them with every *child class* of it. An example of *Hierarchical Inheritance* is shown below.

Example: Program of *Hierarchical Inheritance* **[23]**

Output: disp() method of *Class*Adisp() method of *Class*B disp() method of *Class*A disp() method of *Class*C disp() method of *Class*A disp() method of *Class*D

```
class ClassA {
public ClassA(){
 System.out.println("disp() method of ClassA");}}
class ClassB extends ClassA {
public ClassB(){
 System.out.println("disp() method of ClassB");}}
class ClassC extends ClassA{
public ClassC(){
 System.out.println("disp() method of ClassC");}}
class ClassD extends ClassA{
public ClassD(){
 System.out.println("disp() method of ClassD");}}
public class HierarchicalInheritanceTest {
public static void main(String args[]){
 ClassB objB = new ClassB();
 ClassC objC = new ClassC();
```

The above program consists of five *classes* in which the main *class* is named as *class 'HierarchicalInheritanceTest'*. The *superclass* is *class* A which is inherited by the other four *classes*, so the *child classes* are *class* B, *class* C and *class* D. Every *class* contain its default *constructor* in its *CPN* representation. The *objects* for every *child class* are created in main *class 'HierarchicalInheritanceTest'* as are shown in Figure 3-11. The *class 'HierarchicalInheritanceTest'* is represented by a *place* which *Carries* three tokens *"objB"*, *"objC"* and *"objD"*. These are the *objects* of *child classes class* B, *class* C, and *class* D respectively. In *CPN* representation the *inscribed transitions* are used to call or load *classes* when the *object* of that particular *class* is made.

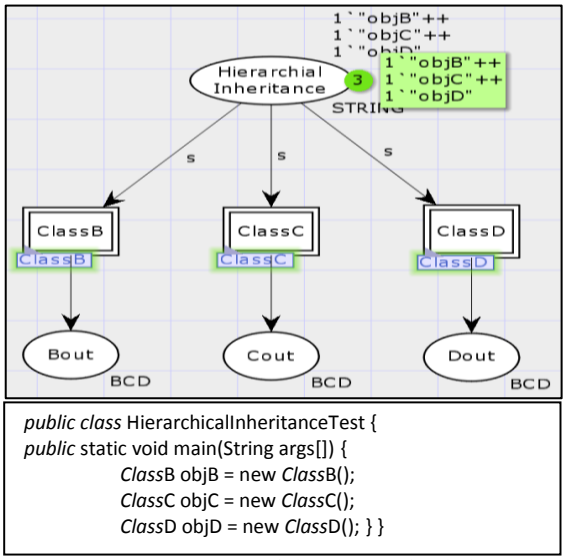
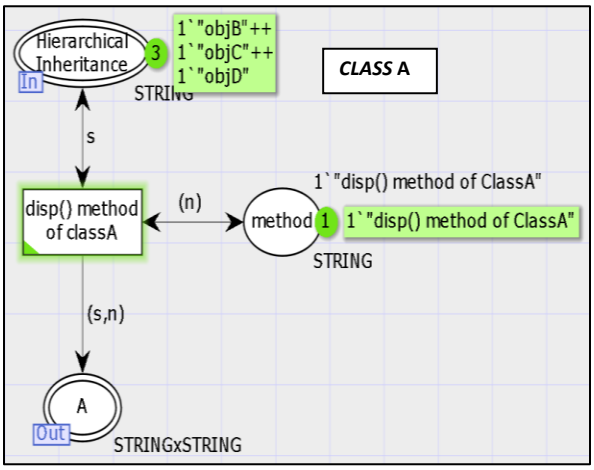


Figure 3-11 CPN representation of class 'HierarchicalInheritanceTest'

When the tokens are fired from the main *class 'HierarchicalInheritanceTest'* and received by the *child classes*, they start executing their statements. The *child classes* also extend *superclass* A so the tokens are transferred to the *superclass* first. The *superclass* contains one default *constructor* which prints *"disp() method of class A"* after which the *child class* executes its *constructor* which prints *"disp() method of* (*child class* name which calls the *class* A)". The *superclass* will run every time the new *object* is created by the main *class*. The *CPN* representation of a *superclass* is shown below in Figure 3-12 in which the *place Hierarchical Inheritance* is the *input port* which fire tokens. When the connection between two pages is formed through *set-subpage* tool the extra *places* are to be made on the subpages to abate this we can use fusions in *CPN* tool.

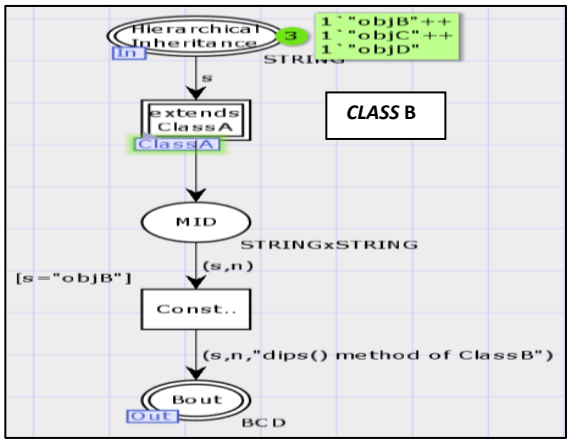
© 2019, IJCSE All Rights Reserved




```
class ClassA {
public ClassA(){
        System.out.println("disp()   method of ClassA");
}}
```

Figure 3-12 CPN representation of class A of Hierarchical Inheritance program

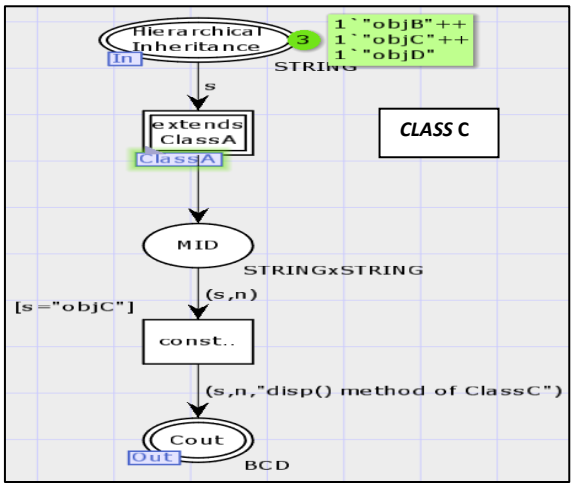
When the tokens are fired from the *superclass* they are received by the *child classes* subpages. The *CPN* representation of *class* B which also *'extends' class* A which is represented by the *inscribed transition* is shown in Figure 3-13 as shown below.




```
class ClassB extends ClassA {
public ClassB() {
        System.out.println("disp() method   of ClassB");
}}
```

Figure 3-13 CPN representation of class B of Hierarchical Inheritance program

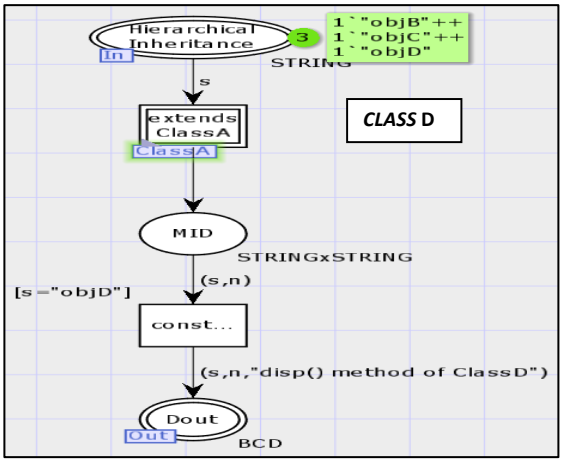
The *constructor* for every *class* is represented by extra *place* and *transition* which will only execute after the execution of *parent class* A *constructor*. Similarly, the representation for *class* C is shown in Figure 3-14.




```
class ClassC extends ClassA{
public ClassC() {
        System.out.println("disp() method   of ClassC");
}}
```

Figure 3-14 CPN representation of class C of Hierarchical Inheritance program

The *CPN* representation of *class* D is shown in Figure 3-15.




```
class ClassD extends ClassA{public ClassD()  {
        System.out.println("disp() method    of ClassD");
        }}
```

Figure 3-15 CPN representation of class D of Hierarchical Inheritance program

Initially, the *object* of *class* B is created so the token of *class* B *"objB"* is fired on the subpage of *class* B, after which the internal call for *parent class* is made through *class* B and *class* A is called. When *class* A is called the default *constructor* of *class* A runs automatically which fire token *"disp() method of class A"* after this the default *constructor* of *class* B executes and print *"disp() method of class* B". After the *object* of *class* B, the *object* of *class* C is created so same process of execution of *constructor*s and *classes* is

© 2019, IJCSE All Rights Reserved

followed by *class* C and *class* D and the *output* tokens will be fired on the main *class 'Hierarchical*InheritanC*Test*'.

5.3 POLYMORPHISM

Polymorphism in Java is a concept by which we can perform a single action in different ways. *Polymorphism* is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So *Polymorphism* means many forms **[24]**. The *CPN Polymorphism* can be represented by using *Inheritance* concept and defining the same method on multiple pages which are connected with the help of *set subpage* tool. The method *overriding* is performed by using let expressions which change the existed value of any variable or method. Here the fusion concept of *CPN* tool is also used for sharing tokens from one page to another.

The syntax of let expression is as follows:

```
let
   val //value of variable
in
   //expression
end
```

The *CPN* representation of *Polymorphism* property is given in Figure 3-16.

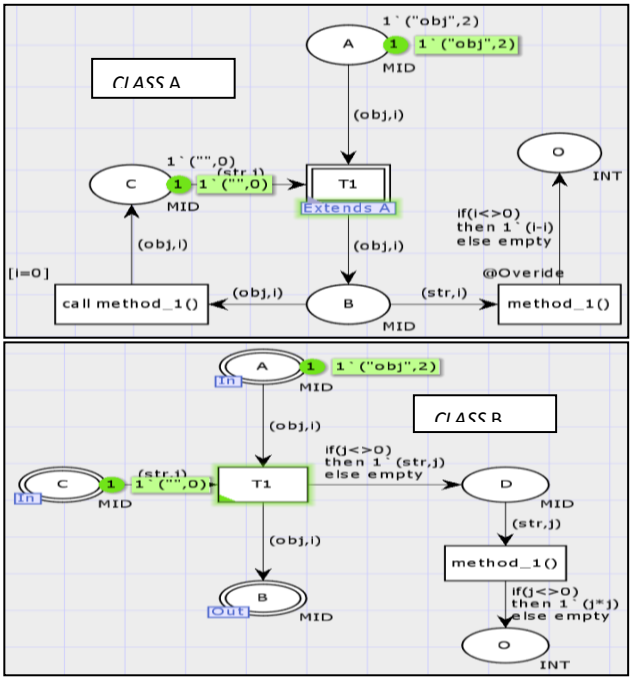


Figure 3-16 showing method overriding of 'method_1()' when one class inherit another

In *Polymorphism* representation of *CPN*, there is one assumption that the values used by the overridden method should not initialize with zero. The zero value is used to run the *parent class* method and if the value is not zero the overridden method will execute. The above *CPN* is the design of two *classes class* A and *class* B where the *class* B is the *parent class* of *class* A. The method *'method_1()'* is defined in *parent class* B which is further overridden by redefining in *child class* A. In *parent class 'method_1()'* the Square of integer value is performed and in the overridden *'method_1()'* the subtraction is performed.

The call *'method_1()' transition* remain inactive as the program runs the overridden method instead of *parent class* method *'method_1()'*.The integer value provided to the program is '2' and the *output* value of the program is '0'.

IV. FAULTS IN OBJECT-ORIENTED PROGRAMMING

In this section, some common faults in OOS and their possible effects are discussed. In programming, a fault is an incorrect statement or a group of incorrect statements in the program which terminates the program abnormally or it may also cause a failure of the program. In computing and operating systems, a trap, also known as an exception or a fault, is typically a type of synchronous interrupt typically caused by an exceptional condition (e.g., breakpoint, division by zero, invalid memory access)**[25]**. A fault or failure may be observed in multiple steps, the initial condition is when the defective statement is reachable by the program then only it can be observed, second if after the execution of that defective statements the program enters into abnormal behavior or stops working then it can be observed. If after the execution of defective statements program gives correct *output* then the fault cannot be observed, so it must provide wrong *output* after the execution of defective statements. A defective program may run and execute successfully without entering into abnormal behavior but it may also produce data anomaly which is also a type of fault. In case of a data anomaly system will not be affected many times but in case of the real-time operating system, a data anomaly may also create a problem. Software faults are most often caused by design faults. Design faults occur when a designer, (in this case a programmer,) either misunderstands a specification or simply makes a mistake **[26]**. If some data anomaly or fault occur then the programmer needs to backtrack/review the full program for which much time is required.

Faults in OOP are of several kinds, many of them are dependent upon OOP's concepts like *inheritance* related faults (e.g. *Improper declaration of parameters, Access violation*, etc.) or *Polymorphism* related faults. Most of the faults are data related faults in which the variables, methods, and *constructor* are not well defined or initialized by the programmer. Some faults are standard faults which can be examined by the compiler at the compilation time like *Variable Access Violation* related faults, uninitialized *class*/package or methods *CALLING*, trying to achieve circularity *inheritance* or *CALLING* the *child class* method in the *parent class*. Many of these faults are discussed in section 5.

© 2019, IJCSE All Rights Reserved

Most of these faults are producing data anomaly because of the improper declaration of method or *constructor* by the programmer. Also, there are some faults which occur due to *Variable Access Violation*.

A summarization of the OOP faults is shown below in this chart.

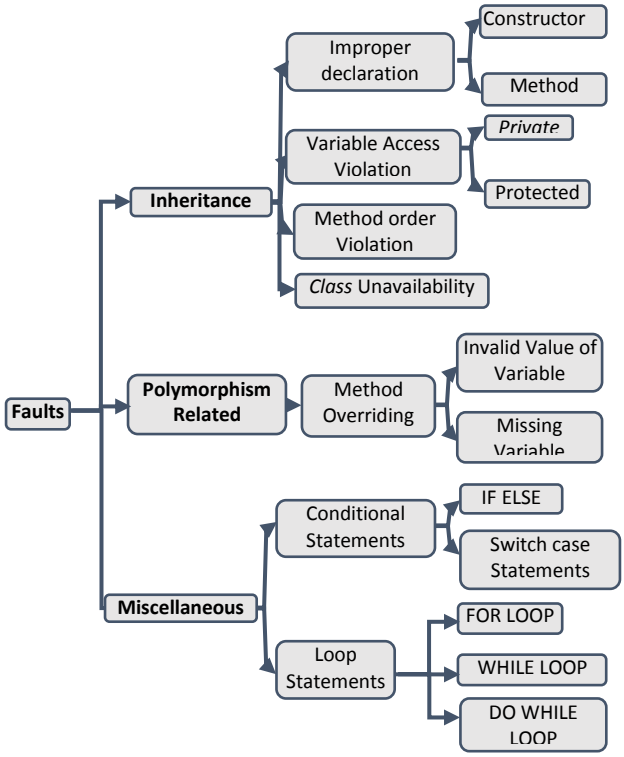


Figure 4.1: Summarization of OOS faults

The order of method *CALLING* also matters in many cases for example if someone is trying to call a method of *child class* inside the *parent class* of that *child class* then this will throw an error which is also not possible in *CPN*. In the case of *Polymorphism*, there are some faults related to method overriding which are also discussed below with an example. The fault in case of *Polymorphism* can occur when the overridden method is not defined properly or maybe some value is missing and declared with another variable.

V. DEMONSTRATION OF OOP'S FAULTS THROUGH *CPN*

In this section, the faults in OOP are demonstrated through *CPN*. The *CPN* representations are based on section 3 (representations of OOP concepts). Most of the faults may be observed during the development of the *CPN* model.

Demonstration of several OOP faults is given as follows.

5.4 INHERITANCE FAULTS

The *inheritance* is the most im*port*ant concept of OOP which helps for code re-usability. Unfortunately, it also allows some data anomalies and faults which either produce wrong *output* or forcefully to push the system into abnormal behavior. There are several faults discussed related to *inheritance* below.

5.5 IMPROPER DECLARATION OF PARAMETERS

The *improper declaration of parameters (IMDP)* will produce a data anomaly without any abnormal termination or failure of the system software program. The improper declaration of variables or parameter can be done either in the *constructor* of *class* or by the method of the *class*. The improper declaration in the *constructor* will affect the program when that *constructor* is used to set the values and those values are further used by another *constructor* or method in the program. As if the *constructor* defined is the default *constructor* then it will run on every *object* creation and affect the program. The improper declaration in the method can also affect the program in the same way as the *constructor* is affecting. This is rarely possible in the single *class* declaration so this is shown by using the *inheritance* concept.

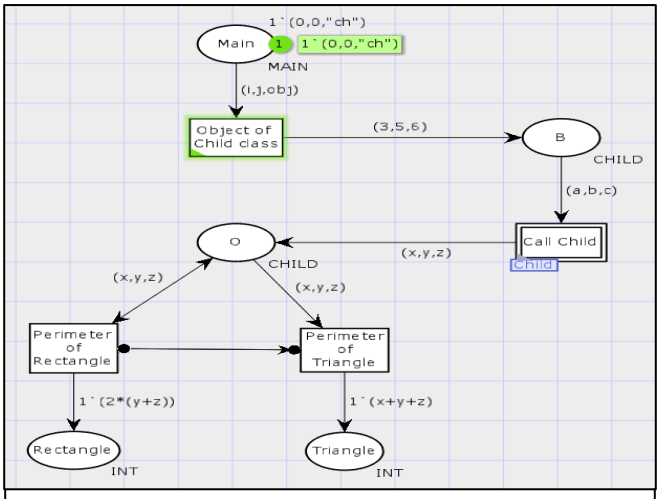
The example of java program for an improper declaration of the *constructor* is given below in which main *class* is defined as *IMDP* (*improper declaration of parameters*) and other two *classes child* and *parent class* are defined with the name '*Child*' and '*Parent*' respectively. The main *class* consists of two integer variables *'periofsq (perimeter of the square)'* & *'perioftri (perimeter of the triangle)'* and one *object "ch"* of the *child class* is created. The '*Parent*' and '*Child*' contain parameterized *constructor*s which are used to set the values of variables.

Example: Java program with *improper declaration of parameters*

```
Output: Perimeter of Rectangle = 18 // (incorrect)
            the correct is 22   Perimeter of triangle = 14
class Parent {
int x,y;
Parent(int l, int b){
x = b;
y = l;}}
class Child extends Parent {
int z;
Child(int l, int b, int h){
super(l,b);
z = h;}
public int perimeterOfTri(){
return (x+y+z);}
public int perimeterOfSq(){
return 2(y+z);}}
class IMDP{
public static void main(String args[]){
  int periofsq, perioftri;
  Child ch = new Child(3,5,6);
  periofsq = ch.perimeterOfSq();
  perioftri = ch.perimeterOfTri();
  System.out.println("Perimeter of Rectangle = "+periofsq);
  System.out.println("Perimeter of triangle = "+perioftri);  }}
```

© 2019, IJCSE All Rights Reserved

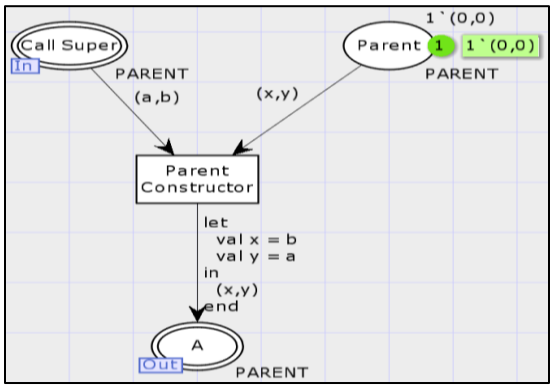
The main *class IMDP* declare two variables '*periofsq*' and '*perioftri*'. It also creates a *Child class object 'ch'* and makes a call for the parameterized *constructor* of that *class* by p*assing* three values 3, 5 and 6. The *CPN* representation of the above program is shown below




```
class IMDP{
public static void main(String args[]){
                int periofsq, perioftri;
                Child ch = new Child(3,5,6);
                 periofsq = ch.perimeterOfSq();
                perioftri = ch.perimeterOfTri();
                 System.out.println("Perimeter of
                Rectangle="+periofsq+"sqr meter");
                System.out.println("Perimeter of  triangle="+perioftri+"sqr
                meter");   }}
```

Figure 5-1 CPN representation Main class of Java program with (IMDP) using inheritance

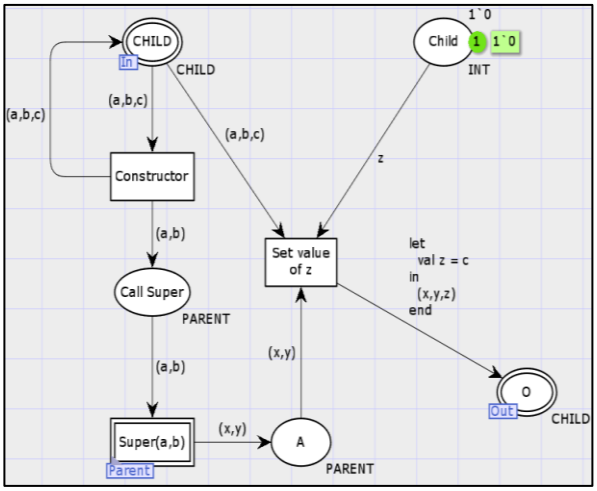
The *CPN* representation of '*Parent*' *class* is shown below which contain two integer variables x and y which are initialized with zero value and also contain parameterized *constructor* which set the values of variable x and y.




```
class Parent {
int x,y;  Parent(int l, int b){
x = b;  y = l;   }}
```

Figure 5-2 CPN representation of 'Parent' class of Java program IMDP

The *CPN* representation of '*Child*' *class* is shown below which consist of one integer variable z (Z is also initialized to zero) and a parameterized *constructor* which explicitly call the parameterized *constructor* of *Parent class* using '*super*()' keyword. The call of *super*() must be in the first line of java program so in *CPN* also the *place* which represents the *superclass* is next to the *input port*. Only after the execution of *super*(), the next *transition* will be active on the page of the '*Child*' *class*. The '*Child*' *class* is represented as shown in Figure 5-3.




```
class Child extends Parent {
int z;
Child(int l, int b, int h)  {super(l,b);   z = h;}
public int perimtrOfTri()   { return (x+y+z);}
public int perimtrOfSq()    {return (y+z);  }}
```

Figure 5-3 CPN representation of 'Child' class of Java program IMDP

When the *object* token is fired with parameters from the main *class IMDP* it will be received by the '*Child*' *class* and the parameterized *constructor* of '*Child*' *class* starts execution. The call of *super*() is made first which fire tokens to the parameterized *constructor* of '*Parent*' *class* as shown in Figure 5-4.

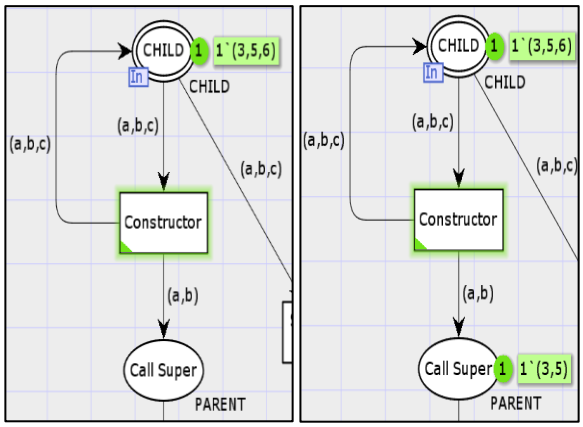


Figure 5-4 firing of tokens from the main *class* to *Child* in the *IMDP* program

© 2019, IJCSE All Rights Reserved

Fault in the *IMDP* program is also observed during the development of the *CPN* model or after running the *CPN* model. The *input* given to the IDMP program is the three sides in which two of them are the sides of the rectangle which are also equal to the two sides of the triangle.

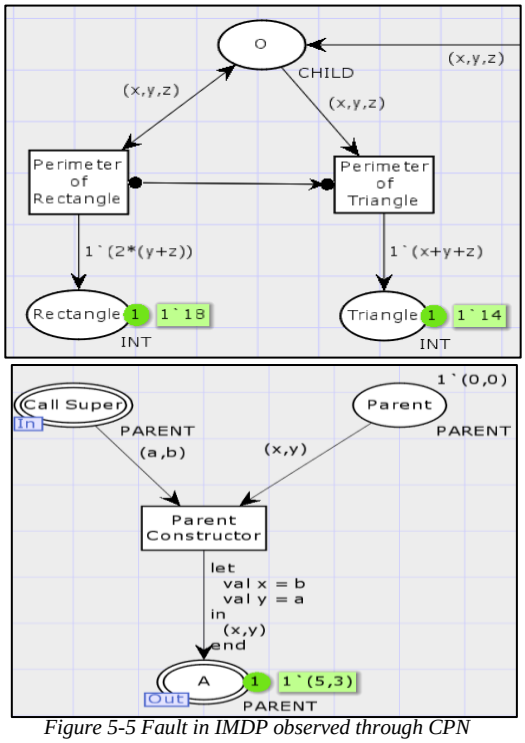


Figure 5-5 Fault in IMDP observed through CPN

The *output* of the program is the perimeter of the Rectangle and perimeter of Triangle which are 18 and 14 respectively. In which the perimeter of the Rectangle is wrong because of values swapping in '*Parent*' *class constructor*. The token fired from the '*Child*' *class* is 1` (3, 5) when the *super* is called but after setting the values of x and y, the token fired have values 1` (5, 3) which can be easily observed that there is some fault in the '*Parent*' *class*. Here the values are swapped in '*Parent*' *class constructor*.

The right answer for the perimeter of a Rectangle is 22 which can be calculated as 2(6+5) but the program is calculating 2(3+6) which is equal to 14. So, this is the *improper declaration of parameters* (*IMDP*) in the '*Parent*' *class constructor* which can be observed through *CPN* and also figured out easily in the program.

5.6 VARIABLE ACCESS VIOLATION

Variable Access Violation (VAV) is the standard fault which can also catch by the compiler. In Java, Access specifiers are used to specifying the accessibility (scope) of data members, methods, *constructor* or *class*. The access specifiers help in controlling unwanted or unnecessary access. There are four types of Java access specifiers. The most restricted access specifier is *Private* by which the *private* members, methods or *constructor*s are not accessible outside the block in which they are defined. The *Private class* is not accessible outside the package. The most non-restricted access specifier is *Public* in which *Public* members, methods, *constructor*s or *classes* are accessible anywhere in the project. When the *CPN* model of a program is made there will be an error for *Private* member or methods.

A general *CPN* example for *Variable Access Violation* is taken for showing an error.

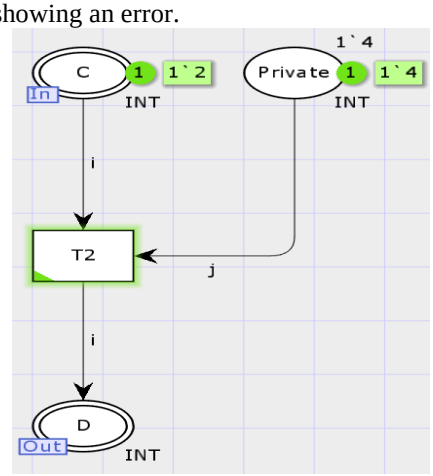


Figure 5-6 CPN representation of Parent class B Of java program for VAV

The *class* B consist of two integer variables *'i' & 'j'* from which *'i'* is *public* data member and *'j'* is *private* data member. Only *public* tokens are fired through *transition* T2 so that the *private* data member *'j'* is not available outside the *class* B. Now, when a *private* data member is called outside *class* B *CPN* throw an error which says that only one integer value is available and the call is made for two integer values. The *child class* is shown below in Figure 5-7. *CPN* representation of *child class* B of *parent class* A with an error popup.

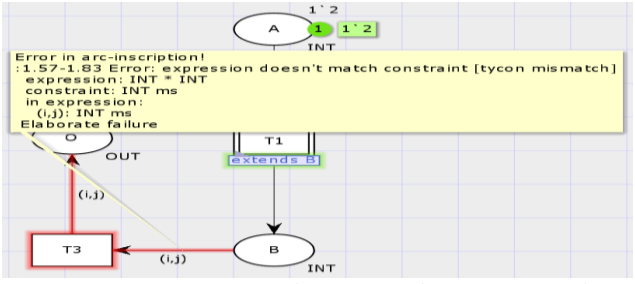


Figure 5-7 CPN representation of child class A for java program of VAV

The *output port* of *class* B is sending tokens to *place* B so value available at *place* B is of only *'i'* variable so the popup is shown for the extra value called by the *arc* connecting through *place* B to *transition* T3.

5.7 METHOD ORDER VIOLATION

Method Order Violation (MOV) is the fault occur when the *child class* method is called from its *parent class*. This can also catch by the compiler at compile time and in *CPN* if this kind of method *call* is performed then *CPN* will raise an error

© 2019, IJCSE All Rights Reserved

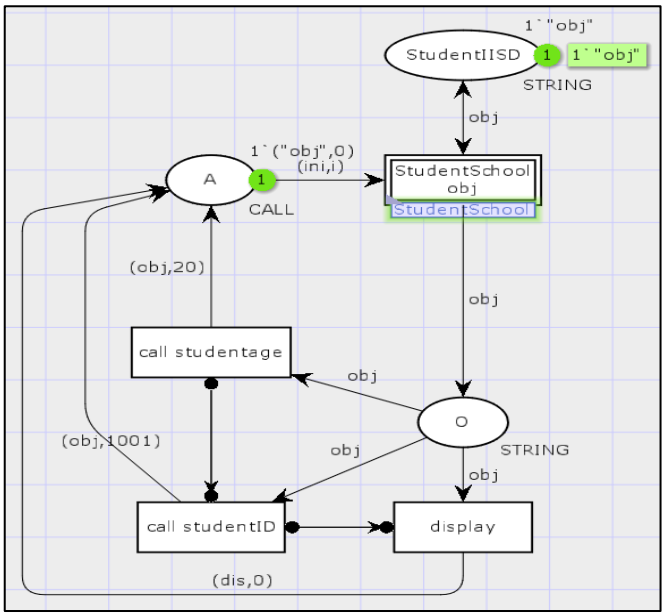
of circularity. As the *set-subpage* tool is used for *inheritance* so to make a call for a method defined on another page the connection between those pages is required which makes a circularity, for example, let us assume the method defined in the *child class* is *Child:: M()*. In *CPN* the *child class* is connected to the *parent class* with the help of *inscribed transition* on the *child class* page and there is no *inscribed transition* in the *parent class* which shows the *parent class* is the *super*() *class* of the program. For *calling* the method of the *child class*, *parent class* needs to connect that *transition* with the *child class* page which is not possible and throws a circularity error. This can also be observed with an example of a java program which performs *method order violation* as given below.

Example: Java program with *method order violation using*

```
Output: error: cannot find symbol
school = StudentSchool();
symbol:   method StudentSchool()
location: class StudentDetails
class StudentDetails {
int age,ID;String school = "RVS";
public int studentage(int a){
age = a;return age;}
public int studentID(int b){
ID = b;return ID;}
public void display(){
school = StudentSchool();
System.out.println(age+" "+ID+" "+school);  }}
class StudentSchool extends StudentDetails{
public void StudentSchool(){
school = "Ridhikul Vidyapeeth Sonipat";  }}
class StudentIISD {
public static void main(String args[]){
StudentSchool obj = new StudentSchool();
obj.studentage(20);
 obj.studentID(1001);obj.display();
 }}
```

The program consists of one main *class* named as *StudentIISD* and two *classes* from which one is *parent class* named as *StudentDetails* and another is *child class* named as *StudentSchool*. So the main *class StudentIISD* creates an *object "obj"* of *child class StudentSchool* and used for method *calling* for *child class* and *parent class*. The *parent class* consists of two integer variables *'age'* & *'ID'* and one string variable *'school'* which is set with the string value *"RVS"*. Some setter methods and one display method are also defined. The *child class StudentSchool* consist of one *StudentSchool*() method to reset the value of '*school*' before displaying with the display method.

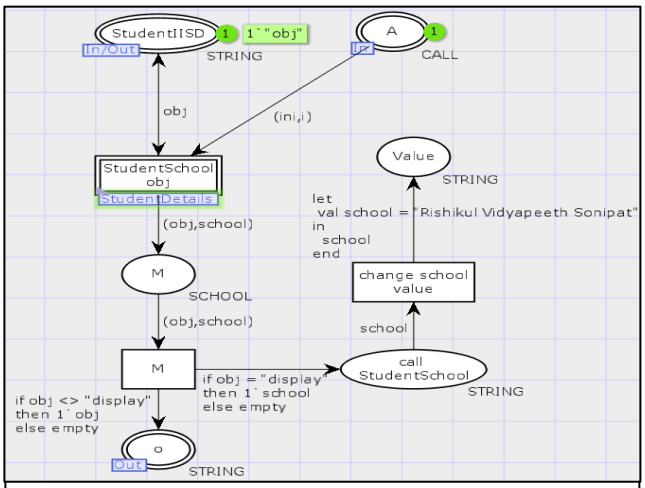
The *CPN* representation of the above program is given below.




```
class StudentIISD {
public static void main(String  args[]){
StudentSchool obj = new  StudentSchool();  obj.studentage(20);
obj.studentID(1001);
obj.display(); }}
```

Figure 5-8 gives the CPN representation of main class StudentIISD

The *place* A in Figure 5-8 helps in *calling* different methods of *child* and *parent class* and transfer the parameterized call with the help of tokens, so initially the *place* A is initialized with the null values to activate the *transition*s. The *CPN* representation of *child class StudentSchool* is shown below.

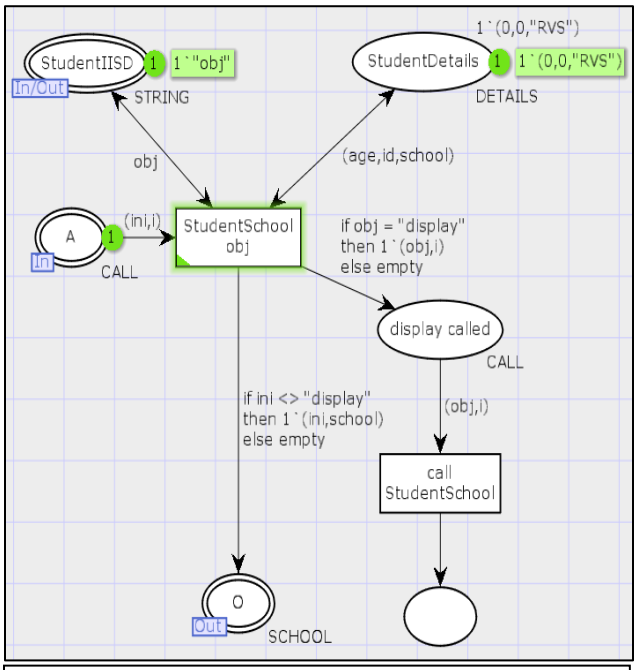



```
class StudentSchool extends StudentDetails {
public void StudentSchool()  {
school = "Ridhikul Vidyapeeth  Sonipat";
 }
}
```

Figure 5-9 Gives the CPN representation of child class StudentSchool

© 2019, IJCSE All Rights Reserved

The *child class* contains only one method *StudentSchool*() which set the value of variable *school* (of string type). The *parent class* consists of two integer variables *ID & age* and one string variable *school*. The variable *school* is initialized with *"RVS"*. Some setter methods are also defined to set the values of *ID & age*. The *CPN* representation for *parent class* is given in Figure 5-10.




```
class StudentDetails {
int age,ID;
String school = "RVS";
public int studentage(int a){
age = a;
return age; }
public int studentID(int b){
ID = b;
return ID;
}
public void display(){
school = StudentSchool();
System.out.println(age+"  "+ID+" "+school);
  }
}
```

Figure 5-10 Given CPN representation of parent class StudentDetails

In this program, the programmer wants to change the value of the *school* variable and to change the value of *school* variable the *StudentSchool*() method must be called which is defined in the *child class*. The *calling* of *child class* method from *parent class* is not possible in Java and also not possible in *CPN* so *CPN* throws a circularity error as shown in Figure 5-11.

Circularity error is thrown while setting subpage form *transition* 'call *StudentSchool*()' of the *parent class* to *child class* page.

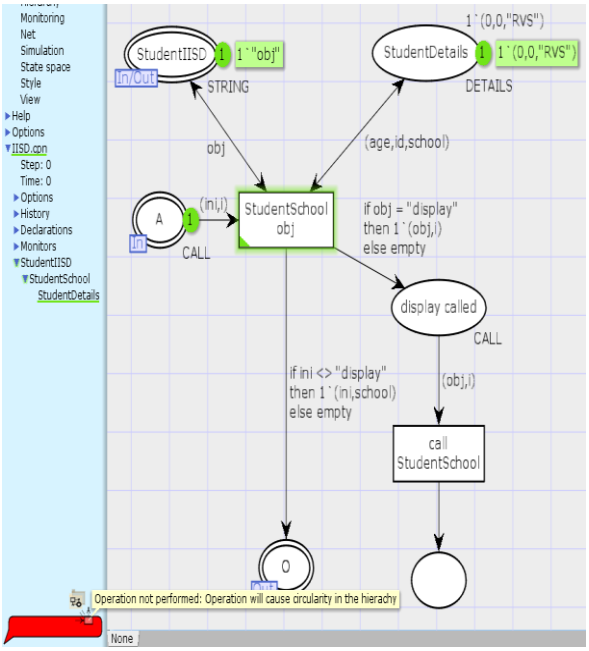


Figure 5-11 Gives the error representation when the set-subpage tool is applied on transition call StudentSchool()

So *Method Order Violation* can occur when someone calls the method defined in *child class* through *parent class* and for this kind of error *CPN* throws a circularity error.

5.8 CLASS UNAVAILABILITY ERROR

This kind of Error will occur when the call for the *unavailable class* is made. This is also caught by the compiler when compiling that program. In a java program, the programmer will *'extends'* a *class* that is not available yet, so the compiler will throw an error at compile time checking. The example of this kind of fault is shown below.

Example: Java program with *Class Unavailability Error using*

Output: error: cannot find symbol
symbol: *class Universe* 1 error

```
class Earth extends Universe {
public static void main (String args[]){
Earth e = new Earth();
System.out.println("Yes!!! I kno everything");
}
}
```

In the above example, the *class Earth 'extends' class Universe* and the *Universe class* is uninitialized so *Universe class* is not found is the error thrown by the compiler. In *CPN* the pages are used to define a *class* so here a page for *Universe class* is defined but the defined page is empty which can be thought as the *class* is not defined so the error for *class* unavailability will be thrown as shown below.

© 2019, IJCSE All Rights Reserved

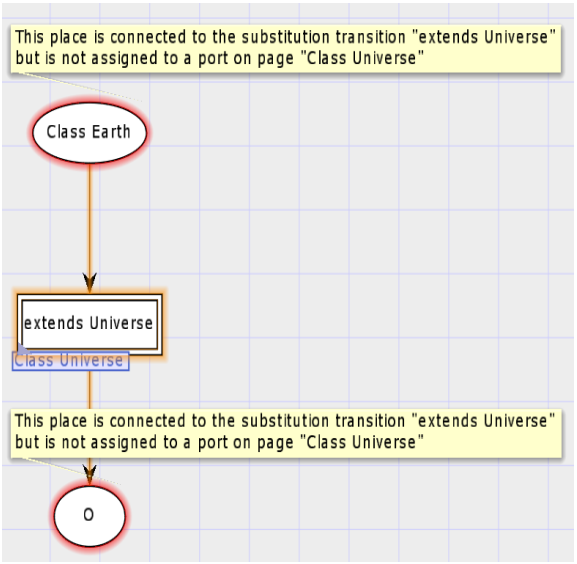


Figure 5-12 Representation of class Unavailability error raised in CPN

The above *CPN* representation is of *class Earth* which is extending the *class Universe*. The page which is assigned for the *class Universe* is empty. This means the *Universe class* is not defined hence the error occurred. The *places* which are connected to the *inscribed transition* are not able to throw and receive the tokens from the extended *class*. The error is the '*place* connected to the substitution *transition* is not assigned to the *port* on the *class Universe*' which means the *input* and *output ports* are not assigned on the *class Universe* page.

The *CPN* representation of *Universe class* does not contain anything.

5.9 POLYMORPHISM FAULTS

Polymorphism is one of the OOPs features that allow us to perform a single action in different ways **[27]**. Unfortunately, it also allows some data anomalies and faults which either produce wrong output or forcefully to push the system into abnormal behavior. There are several faults related to *Polymorphism*, one of the fault is discussed below.

5.10METHOD OVERRIDING FAULT (INVALID VARIABLE OR VALUE)

Method *overriding* is a run-time *Polymorphism*. It is also known as dynamic *Polymorphism*. There are some data anomalies which occur due to variable missing or an invalid value is given in the *overridden* method. In this kind of data anomaly, the *output* of the program will be wrong. The scenario here is supposed there are two *classes* in which one is *parent class* and another is *child class*. The data anomaly will occur here when the *child class* redefines a method of the *parent class* in an inconsistent way. The inconsistent declaration could be the variables defined by the *parent class* method are not redefined by the *child class* method. The *child class* method defines its local variables. Also, there are some methods which were using the values defined by the *parent class* method and now they use the values defined by the overridden method of *child class* which will produce data anomaly. There is one more scenario in which the program will throw a data anomaly. In this, the overridden method is redefining the variables in an inconsistent way which means the *child class* method is redefining the variables defined by the *parent class* method but the values provided to them are not correct. The second scenario rarely occurs in the program. The example of a java program for the first scenario is given below which is producing data anomaly. In the java program *multilevel inheritance* is also used in which three *classes* are defined, the main *class* named as *'Extra'* which *'extends'* the *class 'Assing'*. The *class 'assing' 'extends'* the *class Marks*, so *Marks* is the *superclass*.

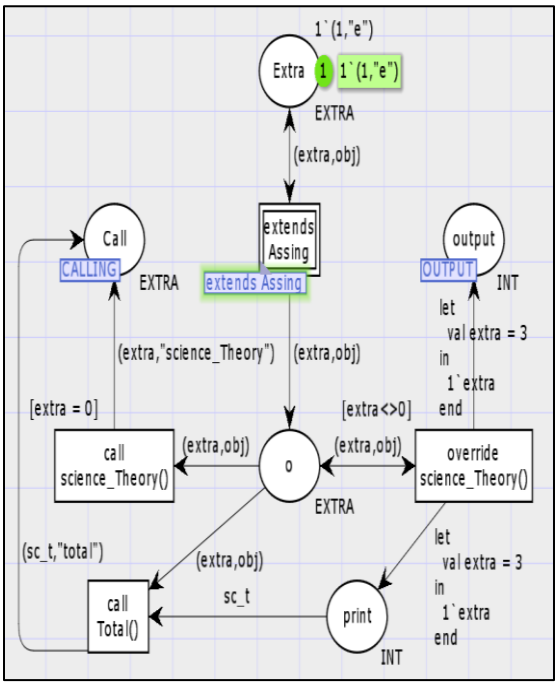
Example: Java program with *Method Overriding Faults using*

```
Output:  Science Theory Marks = 3  // incorrect value  Science
assignment Marks = 5  //incorrect

class Marks{
public int science_T,science_P,total;
public int science_Theory(){
science_T = 52;
return science_T;}
public int practical_Marks(){
science_P = 30;
return science_P;}
public int Total(){
total = science_P + science_T;
return total;   }}
class Assing extends Marks{
int assin;
public int Total(){
if(science_T>50){assin = 10;}
else if(science_T<50 && science_T>45){assin = 7;}
else{assin = 5;}
return assin;    }}
class Extra extends Assing{
int extra_M;
public int science_Theory(){
extra_M = 3;return extra_M;}
public static void main(String[] args) {
Extra e = new Extra();
int science_T = e.science_Theory();
int assin = e.Total();
System.out.println("Science Theory Marks = "+science_T);
System.out.println("Science assignment Marks = "+assin);
   }}
```

Here the program is designed for student marks details for a science subject. The *superclass 'Marks'* defines the theory marks and practical marks where the assignment marks are assigned in the *Assing class* and some extra marks are added into the main *class* which overrides the theory marks method of the *superclass* and produce a data anomaly. The *CPN* representation of main *class* Extra is given below.

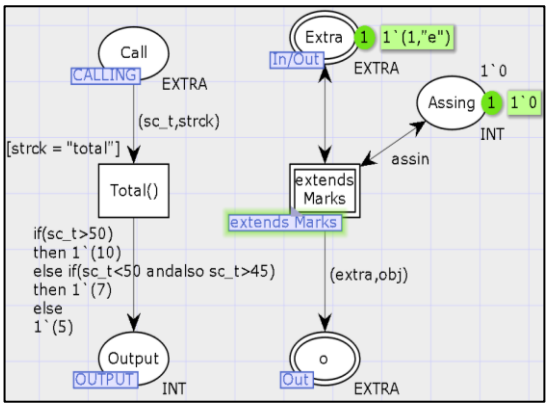
© 2019, IJCSE All Rights Reserved




```
class Extra extends Assing{
int extra_M;
public int science_Theory(){
extra_M = 3;
return extra_M;  }
public static void main(String[]args){
Extra e = new Extra();
int science_T=e.science_Theory();
int assin = e.Total();   System.out.println("Science Theory
Marks = "+science_T);   System.out.println("Science
assignment Marks = "+assin);   }}
```

Figure 5 13 CPN representation of Java program for main class Extra

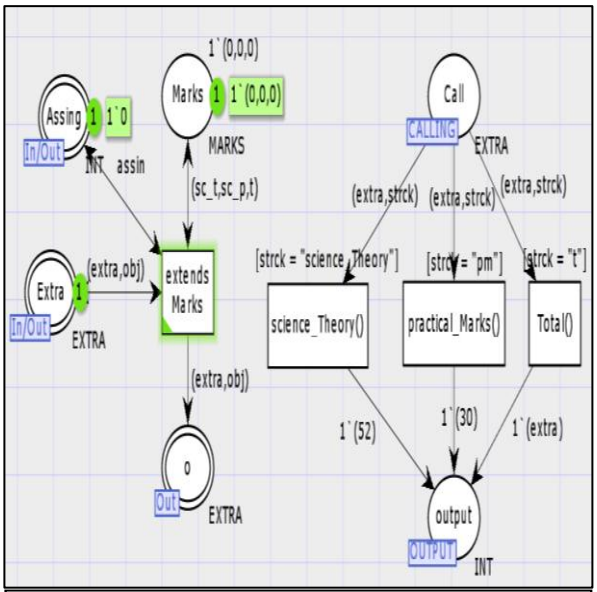
The above representation of the java program contains one integer variable '*extra'* and one *object* '*e*' of *class Extra*. *Class Extra* also *'extends' class Assing* which is represented by the *inscribed transition*. The *class* Extra also contains one overriding method *science_Theory()*. The *place Call* is assigned with a *fusion* name as *CALLING*. The token fired on this *place* will show on every page which comes under *CALLING* fusion set. The *place Output* is also assigned with a fusion name as *OUTPUT* which contains only *output* tokens with it.




```
class Assing extends Marks{
int assin;  public int Total(){
if(science_T>50){
assin = 10;  }
else if(science_T<50  &&  science_T>45)  {
assin = 7;}
else{
assin = 5;}
```

Figure 5-14 CPN representation of java program for class Assing

The *parent class* of main *class* Extra is *class Assing*. The *CPN* representation of *class Assing* which consist of one integer variable '*assin*' and one method *total()*. The method *total()* is connected with the *CALLING* fusion which fires the called tokens to the *total()* method *transition* and *output* are connected with *OUTPUT* fusion. Now, the *parent class* of *Assing* is *class* Marks which is shown below.




```
class Marks{
public int science_T,  science_P,total;
public int science_Theory(){
science_T = 52;   return science_T;}
public int practical_Marks(){
science_P = 30;   return science_P;}
public int Total(){
total = science_P + science_T;  return total;  }}
```

© 2019, IJCSE All Rights Reserved

Figure 5-15 CPN representation of java program for superclass Marks

The *superclass* Marks consist of three integer variables (*science_T, science_P, total*) and three methods *science_Theory(), practical_Marks(), Total()*.
Now, when the token of the *object* has fired the *transition* which overrides the method *science_Theory()* become active and the call to *science_Theory()* remain inactive which is shown in below Figure 5-16.

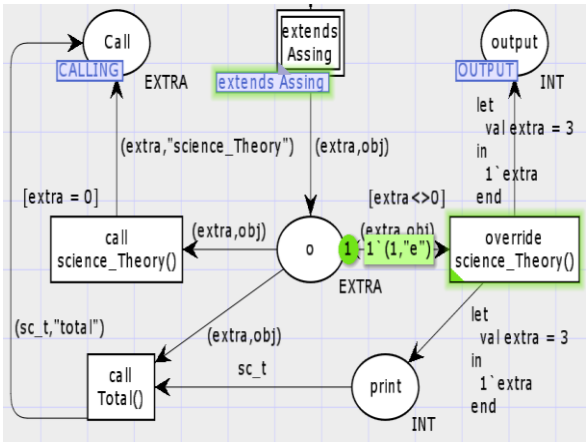


Figure 5-16 the overriding method science_Theory() become active and the call to science_Theory() remain inactive.

When *transition science_Theory()* fires the token then the next *transition* for *Total()* method become active and the *output place* gets '3' as *output*. *Total()* method is not overridden by the main *class* so the call for *Total()* method is completed by firing the token to another *class*. The firing of the token from one page to another is shown below.

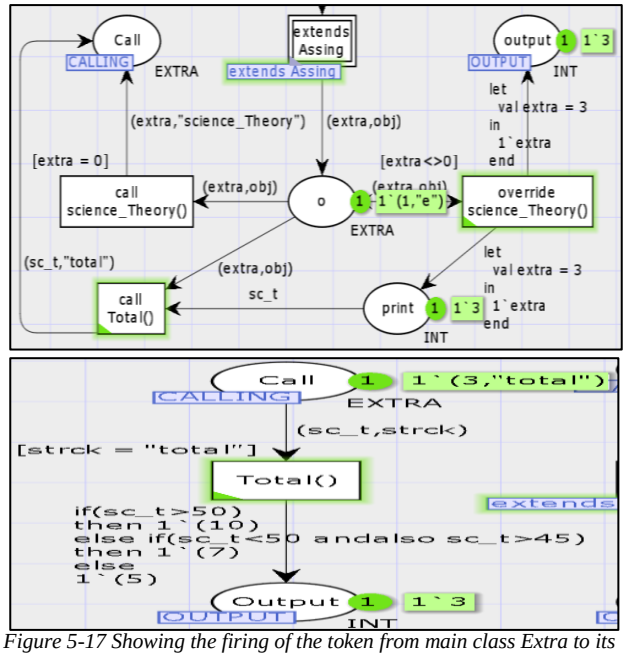


Figure 5-17 Showing the firing of the token from main class Extra to its parent class Assing

When the token is fired from *transition* call *Total()*, the *transition Total()* of *class Assing* become active. After firing of tokens from *transition Total()* the *output* will be 2 tokens as shown below.

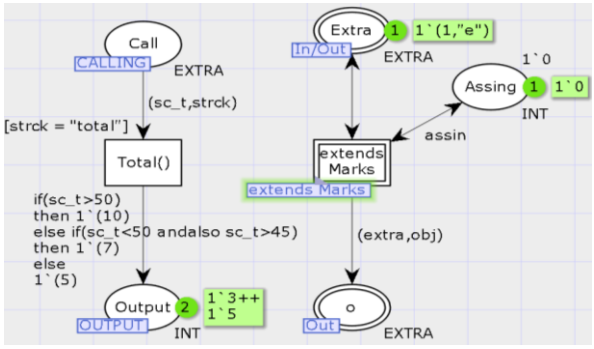


Figure 5-18 output of the example java program

Here the flaw/fault is that the overriding method is defined inconsistently because of which the correct value of science theory marks is overridden by the Extra marks. The science marks are also used to calculate the assignment marks which again produce data anomaly due to the overridden value of science theory marks.

5.11MISCELLANEOUS FAULTS

There are some more common faults other than OOP's faults which are discussed below.

5.12CONDITIONAL STATEMENTS

Statements that allow us to check a condition and execute certain parts of code depending on whether the condition is true or false. Such statements are called conditional statements. This section describes the *CPN* representation of conditional statements. Also, give a brief representation of errors when the syntax or condition is wrong.

5.13IF / ELSE STATEMENTS

The *if-else* statement allows us to select between two alternatives. The condition in an *if-else* statement can be an arbitrary expression of type Boolean. The else part of an *if-else* statement is optional **[28]**. The *CPN* representation for if else is shown below.

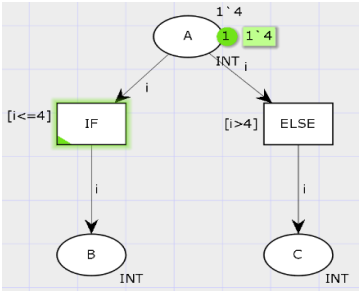


Figure 5-19 CPN representation of If-else conditional statements

© 2019, IJCSE All Rights Reserved

In the above Figure, the value of the token is 4 and the *IF* condition is satisfied so *IF transition* becomes active. The *ELSE transition* remains inactive. The inscription to the IF *transition* checks the condition which helps in activating the *transition*. The *transition* basically has three inscriptions with it. On hitting 'tab key' one time the control goes to the conditional inscription. The syntax for conditional inscription is shown in the figure. It should be in square brackets. Also, it must have some relation to the token fired from the *input place* of that *transition*.

5.14SWITCH CASE STATEMENTS

When there is a multiple-choice selection the several nested *if-else* statements are used **[28]**. The switch statement in java does the same. As *SWITCH CASE* is the extension of *IF ELSE*, so the *IF ELSE* representation of *CPN* is used in the *SWITCH CASE* representation. The *CPN* representation of *SWITCH CASE* is shown below.

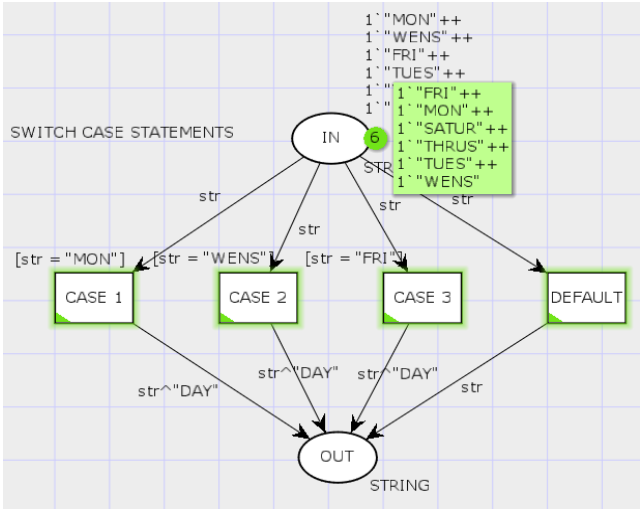


Figure 5-20 CPN representation of SWITCH CASE conditional statements

5.3.4 LOOPS REPRESENTATION IN *CPN*

In this section, the *CPN* representation of loops like *for loop, while loop and do while loop* is discussed, also loops which formed when the same method is called again and again (recursion). The representation of all loops is almost the same.

A *while* loop is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement **[29]**.

Do While loop is similar to while loop with the only difference that it checks for the condition after executing the statements, and therefore it is an example of Exit Control Loop **[29]**.

For loop provides a concise way of writing the loop structure. Unlike a while loop, a for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping **[29]**.

The *CPN* representation for loops is shown below.

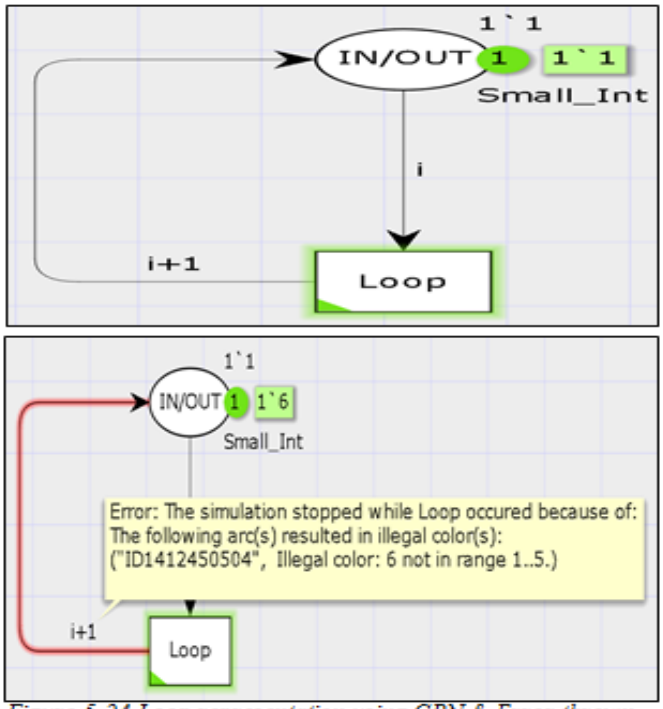


Figure 5-24 Loop representation using CPN & Error thrown by the CPN when the value exceeds the Final value set

In this, the *Small_Int* is the color set used to set the initial and final value for the loop. The "i+1" inscription on the arc creating Loop from the *transition* to *IN/OUT place* is for the increment of value. The final value is set to 5. If the value is greater than the final value of Loop CPN model will throw an error.

VI. CONCLUSION

This work presents the CPN modeling of possible faults that may occur in an OOS. Fault information at an early stage may help in testing and maintenance phase of software development. Mainly inheritance and polymorphism related faults are covered. modeling or done with a small Java program example. This can be extended to create a neural nets so that those nets can learn the OOS concepts and start detecting faults. In future work, more faults may be covered also try to model a real object-oriented software like Management System.

REFERENCES

[1] S. Malve, P. Sharma *Investigation of Manual and Automation Testing using Assorted Approaches,* International Journal of Scientific Research in Computer Science and Engineering,Vol. 5, Issue. 2, pp.81-87, 2017.

© 2019, IJCSE All Rights Reserved

[2] S. Jat, P. Sharma *Analysis of Different Software Testing Techniques*, International Journal of Scientific Research in Computer Science and Engineering,Vol. 5, Issue. 2, pp.77-80, 2017.
[3] A. Marcus, D. Poshyvanyk, R. Ferenc. Using the Conceptual Cohesion of Classes for Fault Prediction in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, Vol: 34 , Issue. 2, pp. 287-300, 2008.
[4] G. Antoniol, R. Fiutem, & L. Cristoforetti, Using Metrics to Identify Design Patterns in Object-Oriented Software. *IEEE METRICS*, 1998
[5] V.R. Basili, L.C. Briand, & W.L. Melo, A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Trans. Software Eng., Vol. 22*, Issue. 10, pp. 751-761, 1996.
[6] E. Arisholm, L.C. Briand, & A.Føyen, Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering, Vol. 30*, Issue. 8, pp. 491-506, 2004.
[7] L.C. Briand, J. Wüst, & H. Lounis. Using Coupling Measurement for Impact Analysis in Object-Oriented Systems. *ICSM*, 1999.
[8] C. Lakos,. Object Oriented modelling with Object Petri Nets. *Concurrent Object-Oriented Programming and Petri Nets*, Springer,Vol. 2001, pp. 1-37, 2001
[9] L.C. Wang, Object-oriented Petri nets for modelling and analysis of automated manufacturing systems, ISSN. Vol. 9, Issue. 2, pp. 111-125, 1996
[10]T. Murata,. Petri Nets : Properties , Analysis and Appl ka t ions., IEEE, vol. 77, Issue. 4pp. 541-580 2004
[11]K. Jensen, & L.M. Kristensen,. Colored Petri nets: a graphical language for formal modelling and validation of concurrent systems. *Commun. ACM, Vol. 58*, Issue. 6, pp. 61-70, 2015.
[12]J.R. Silva, & P.M. Foyo, Timed Petri Nets. ABCM Vol.3, pp. 471-478, 2012
[13]F. Bause, & P.S. Kritzinger,. Stochastic Petri nets - an introduction to the theory (2. ed.). *Advanced studies of computer science*. 1996
[14]D. Wodtke, & G. Weikum,. A Formal Foundation for Distributed Workflow Execution Based on State Charts. *ICDT*, Vol. 1186, pp.230-246, 1996
[15]R. Alur, & M. Yannakakis,. Model Checking of Hierarchical State Machines. *ACM Trans. Program. Lang. Syst., Vol. 23*, Issue. 3, pp. 273-303, 2001
[16]A.S. Dange, A. Marcus, & D. Poshyvanykquot,. Survey of Fault Prediction Methods in Object Oriented Systems, IEEE, Vol. 38, Issue. 99, pp. 1-1, 2011
[17]A.J. Offutt, R.T. Alexander, Y. Wu, Q. Xiao, & C. Hutchinson,. A Fault Model for Subtype Inheritance and Polymorphism. *ISSRE*, pp. 27-30, 2001
[18]L.E. Buzato, C.M. Rubira, & M.L. Lisbôa,. A Reflective Object-Oriented Architecture for Developing Fault-Tolerant Software. *J. Braz. Comp. Soc., Vol. 4*, 1997
[19]K.K. Aggarwal, Y. Singh, A. Kaur, & R Malhotra,.. Investigating effect of Design Metrics on Fault Proneness in Object-Oriented Systems. *Journal of Object Technology, Vol. 6*, pp. 127-141. 2007
[20]S. Jain, H. Gulati, V. Bajpai, S. Goel, D. Singh, S. Baranwal, "*Classes* and *Object*s in Java," A Computer Science *port*al for geeks.. [webpage]. https://www.geeksforgeeks.org/*classes-object*s-java/. [April 20, 2018].
[21]M. Sheridan, "Java Fundamentals," Java - Oracle, Mike Sheridan at Sun Microsystems in 1991. [pdf]. http://www.oracle.com/events/global/en/java-outreach/resources/java-a-beginners-guide-1720064.pdf. [April 15, 2018].
[22]S. Jaiswal, "*Inheritance* in Java," Free Online Tutorials, Javatpoint provides tutorials and interview questions of all technology. [webpage]. https://www.javatpoint.com/*inheritance*-in-java. [April 16, 2018].
[23]BeginnersBook.com, "*Multilevel inheritance* in java with an example," BeginnersBook is a tutorials site for beginners that covers topics like Java, Collections, AWT, JSP, Servlet, JSTL, C, C++, DBMS, Perl, WordPress, SEO.. [webpage]. https://beginnersbook.com/2013/12/*multilevel-inheritance*-in-java-with-example/. [April 16, 2018].
[24]S. Jaiswal, "*Polymorphism* in Java," Free Online Tutorials, Javatpoint provides tutorials and interview questions of all technology. [webpage]. https://www.javatpoint.com/runtime-*Polymorphism*-in-java. [April 24, 2018].
[25]Wikipedia, "Trap (computing)" para. 1, 15, March 2018 [Online]. Available: https://en.wikipedia.org/wiki/Trap_(computing). [Accessed 4, April 2018].
[26]C. Inacio, "Software Fault Tolerance," Engineering Research Accelerator, *Car*negie Mellon University Dependable Embedded Systems, Spring 1998. [webpage]. Available: https://users.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance/. [04,April 2018].
[27]BeginnersBook.com, "*Polymorphism* in java with an example," BeginnersBook is a tutorials site for beginners that covers topics like Java, Collections, AWT, JSP, Servlet, JSTL, C, C++, DBMS, Perl, WordPress, SEO.. [webpage]. https://beginnersbook.com/2013/03/*Polymorphism*-in-java/.[April 27, 2018].
[28]D. Calvanese, "Conditional Statements," Diego Calvanese.Unibz online study lectures [pdf]. https://www.inf.unibz.it/~calvanese/teaching/04-05-ip/lecture-notes/uni05.pdf. [May 8, 2018].
[29]S. Jain, H. Gulati, V. Bajpai, S. Goel, D. Singh, S. Baranwal, "Loops in Java," A Computer Science *port*al for geeks.. [webpage]. https://www.geeksforgeeks.org/loops-in-java/. [April 20, 2018].

Authors Profile

Mr. S Kaushik completed Bachelor of Computer Science and Engineering from University Institute of Engineering & Technology, MDU Rohtak in 2018 and currently working as Software Developer in Wipro since 2018. His main research work focuses on Software Engineering. He has 10 months of job experience and 6 months of Research Experience.

Dr. Ratneshwer is working as an Assistant Professor at School of Computer and Systems Sciences, Jawaharlal Nehru University New Delhi. He has 12 years of teaching and research experience. His area of working is computer networks and software engineering. He has 28 research papers in various international journals.

© 2019, IJCSE All Rights Reserved