Analysis Software with an Object-Oriented Petri Net Model

[1]H. Motameni, [2]A. Movaghar, [3]B. Shirazi, [3]M. Aminzadeh and [4]H. Samadi

[1] Department of Computer Engineering, Islamic Azad University, Sari Branch, Iran
[2]Department of Computer Engineering, Sharif University of Technology, Tehran, Iran
[3]Department of Computer Engineering, University of Science and Technology of Mazandaran, Babol, Iran
[4]Research office, Islamic Azad University, Sari Branch, Iran

Abstract: Petri net is used widely to analyze and model various systems formally. Recently, Many Petri nets mania devote their efforts to enhancing and extending the expressive Power of Petri nets. One such effort is to extend Petri nets with object-oriented concepts. An object-oriented paradigm provides excellent concepts to model real-world problems. Object-oriented concepts allow us to build software systems easily, intuitively and Naturally. Although several high-level Petri nets with the concept of objects are suggested, These nets do not fully support the object-oriented concepts. Object Oriented Methodology lacks the rigor to verify formal validate the designed system. Petri Nets provide formal graphical representation, incorporate concurrency and parallelism. In Colored Petri Nets (CPNs), Objects and object attributes can be modeled with data structures. The hierarchical structure of (CPNs) is useful in representing Class Inheritance and to describe dynamics of objects. To check the correctness of the designed system, there is a need to integrate Object Oriented techniques at design level and use of (CPNs) at the Verification and Validation level in software system development. This paper presents a technique to transform an Object Oriented Design (OOD) into Hierarchical (CPNs) model with Object Oriented Petri Nets Model (OOPNM) approach.

Key word: OOPNM . class . object . inheritance . encapsulation

INTRODUCTION

The development of a software system begins with two main activities: requirements analysis and system modeling. Requirements analysis serves two purposes; to throughly understand the problem and to reduce potential errors caused by ambiguous requirements. The purpose of system modeling is to depict the overall structure of the system by decomposing it into its logical components. Many researchers have focused their interests on the development of techniques to support these two activities [1]. There are two ways to achieve the goals of those two activities. One is to formally specify and analyze systems and the other is to naturally describe and model systems. When specifying, modeling and analyzing the behavior of a critical and complex system, we choose a language which can formally depict the properties of that system. Formal languages support us in describing system properties clearly, exactly and in detail. In particular, Petri net is the most widely used in various application domains because of its simplicity and exibility in depicting dynamic system behaviors and its strong expressive and analytic power of system behaviors. Despite the powerful capability of formal methods,

designers are still likely to recognize the shortage of formal methods when analyzing and modeling systems of various domains. Thus, many Petri net researchers have devoted their efforts to enhancing and extending the theory and techniques of Petri nets [2-4] including high-level Petri nets such as (CPNs) [5,6].

The concepts of object-oriented paradigm such as encapsulation, inheritance, etc., have been widely used in the system modeling because they allow us to describe systems easily, intuitively and naturally [5,6]. Designers who are familiar with formal methods have come to understand the usefulness of the object-oriented concepts. Along with this trend, object-oriented formal methods have also become of particular interest to researchers in recent years and many experts have suggested object-oriented formal methods such as object Petri nets (OPN), VDM++, Object-Z [6], etc. Among these studies, research on OPN formalism has been actively studied to extend Petri nets formalism to OPN such as OBJSA [7], COOPN/2 [8] and LOOPN++ [9].

Although the results of such studies have shown promise, these nets do not support fully all the major concepts of object-oriented paradigm. Object Oriented Methodology is an established techniques for structure

Corresponding Author: Dr. H. Motameni, Department of Computer engineering, Islamic Azad University, Sari Branch, Iran

software design.Object Orientedness supports Inheritance, Polymorphism. Object Oriented methodology is useful in order to design software comprehensible, maintainable and flexible. However, Objct Orientedness lacks analysis and verification methods of designed systems. Petri Nets (PN) [9] are useful in describing and information systems that are characterized as being concurrent, asynchronous, distributed [8,9], parallel, nondeterministic and/or stochastic. The graphical representation, simplicity and executable nasture of Petri Nets model make Petri Nets suitable for simulation, rapid prototyping and verification of systems. There is a need to combine the Object Orientation with Petri Nets to get the advantages of both approaches. In this paper are presenting an approach of (OOPNM) that takes advantage of both Object Orientedness (OO) and Petri Nets methodologies by using Object Orientedness at the design stage and Petri Nets at Verification and Validation stage. (OOPNM) is based on the combination of (OOD) [10] and Petri Nets (PN). (OOPNM) uses the existing tools that support both methodologies. We can design a system in Object Oriented fashion using Unified Modeling Language (UML) [9] and then we can transform this design in into (CPNs) model using CPN tools [10].

OBJECT ORIENTED PETRI NET MODEL (OOPNM)

The presented model is a combination of object oriented and CPN. In this model we proposed a CPN that support all specifications of an object oriented model such as:

Class, Objcet, encapsulation, inheritance. OOD defines the hierarchy of classes and object co-operations and then Petri Nets are used to represent those classes and object interaction. This integration will result in a Petri Nets inside Object construct[18]. The use of Petri Nets model will allow us to verify the system before its implementation.In these models new notations are used rather than existed notations in CPN model. The notations illustrate in Table 1:

More details about each of those notations are explained in part 3.

OOPNM structure: Regarding to the former notations and some notations like place, transition, arc the following structure could be presented for the proposed model:

The OOPNM structure is a 7-tuple $C = (P, T, OB, E, IN, I, O)$:

P: Finite set of places (P = {p1, p2, …, pn}),
T: Finite set of transitions (T = {t1, t2, …, tn}),
      P and T: Disjoint (P n T = Ø)
OB: Finite set of object (OB = {ob1, ob2, …, obn}),
E   Finite set of encapsulation (E = {e1, e2, …, en}),
IN: Finite set of inheritance (IN = {in1, in2, …, inn}),
I: Input function:
O: Output function:

The input and output functions in general are $T \rightarrow P$n by as soon as creation of notations as OB, E, IN it may change. Those changes are in the following 4 forms. We discuss these 4 forms for class notation, for other three notations it is the same [10-12].

Input/Output are places for class: The input to class is input place. so there is an input transition within class, which will transform the data and will put the data in an output place as in Fig. 1.

I(t1) = {p1} O(t1) = {p2}
I(t2) = {p1} O(t2) = {p3}
➔➔➔ *I(ob1) = {p1} O(ob1) = {p4}*
I(t3) = {p2} O(t3) = {p4}
I(t4) = {p3} O(t4) = {p4}

```
public class class1{
  public void t1(){
    p2="1";
    p1="0";
 }
Public void t2(){
  P3= "1";
  P1 = "0";
 }
public void t3(){
  p4 = "1";
  p2 ="0";
 }
```
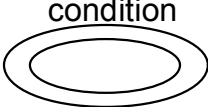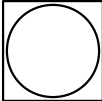
Table 1: Notations illustrate by their properties

condition			
To show the encapsulation specification	To show the class specification	To show the object specification	To show the inheritance specification

Fig. 1: A class and its object when I/O are places
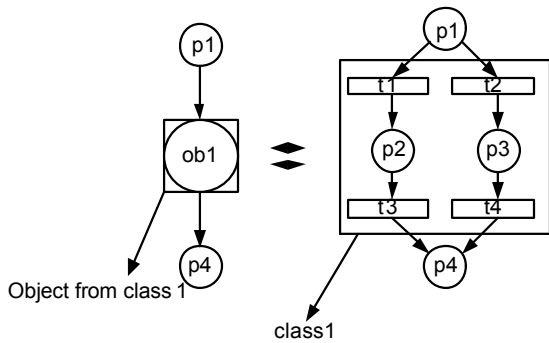

Fig. 2: A class and its object when Input from a place and output to a transition

```
Public void t4(){
   P4 = "1";
   P3 = "0";
   }
}
```

```
class1 ob1 = new class1();
 if (p1="1") {
   ob1.t1();
   ob1.t2();
}
 if (p2 = "1") {
   ob1.t3();
}
 if (p3 = "1"){
   ob1.t4();
 }
```

The input to class is an input place. So, class will be explored to transition or transitions depending on the functionality of class. These transitions are input transitions within class. These transitions are then explored to model the functionality performed by class as represented in Fig. 1 and the output will be returned to the output place that is connected to class by output arc. The expansion of transitions inside class will depend on the type of functions it performs.

In Fig. 1, the arrows to transform class to the corresponding Petri Net model are shown in both directions to represent that we can refine class to get the Petri Nets models or we can abstract the Petri Nets model to have class constructs.

Input from a place and output to a transition for class: Here, there is a transition or transitions within class that will process on this input, as shown in Fig. 2. This processed data will be stored in a place or places inside class, which is connected to the transition outside the class. Petri Net model shown within the dotted line represents the various methods accepted by class.

Again, the arrows to transform class to Petri Nets model are shown in both directions, meaning expansion of class and also abstraction of Petri Nets model to class is possible [12,13].

$I(t1) = \{p1\}\ o(t1) = \{p2\}$
$I(t2) = \{p1\}\ o(t2) = \{p3\}$
➔ ➔ ➔ $I(ob2) = \{p1\}\ o(ob2) = \{t5\}$
$I(t3) = \{p2\}\ o(t3) = \{p4\}$
$I(t4) = \{p3\}\ o(t4) = \{p5\}$
$I(t5) = \{p4,p5\}$

```
public class class2 {
   public void t1(){
      p2="1"
      p1="0";
   }
   public void t2(){
      P3="1";
      P1="0";
   }
   public void t3(){
      p4="1";
      p2="0";
   }
   public void t4(){
      P5="1";
      P3="0";
   }
   public void t5(){
      send out p4 and p5;
   }
}
```

```
class2 ob2= new class2();
if(p1="1"){
   ob2.t1();
   ob2.t2();
}
```
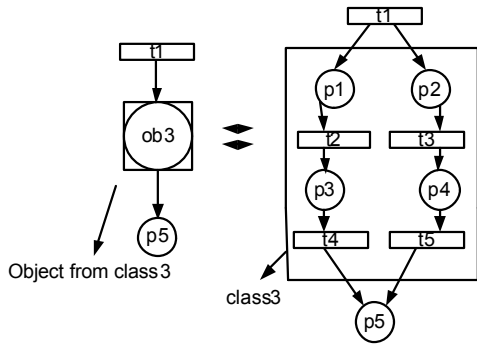
567

Fig. 3: A class and its object input from a transition and output to a place

```
if(p2="1"){
   ob2.t3();
}
if(p3="1"){
   ob2.t4();
}
if(p1="1"&& p5="1"){
   ob2.t5();
}
```

Input from a transition and output to a place for class: Here, input to class is from a transition. So, there is a place or places inside class to store the data. Later, this data will be processed inside the class, depending on the type of data that can be mo0deled as shown in Fig. 3. The output from class goes to place. Therefore, there is a transition inside class, which is connected to the output place. Again, the transformation from class to Petri Nets model is shown in both directions, as it is possible in both ways.

$I(t2) = \{p1\}\ o(t1) = \{p1,p2\}$
$I(t3) = \{p2\}\ o(t2) = \{p3\}$
➔➔➔ $I(ob3) = \{t1\}\ o(ob3) = \{p5\}$
$I(t4) = \{p3\}\ o(t3) = \{p4\}$
$I(t5) = \{p4\}\ o(t4) = \{p5\}$
$o(t5) = \{p6\}$

```
public class class3{
  public void t1(){
     p1="1";
     p2 ="1";
  }
  public void t2(){
     p3="1";
     p1 ="0";
  }
  public void t3(){
     p4="1";
```

```
     p2 ="0";
  }
  public void t4(){
     p5="1";
     p3 ="0";
  }
  public void t5(){
    p5="1";
    p4 ="0";
  }
}
```

```
class3 ob3 = new class3();
ob3.t1();
   if(p1="1"){
   ob3.t2();
}
 if(p2="1"){
   ob3.t3();
}
 if(p3 = "1"){
   ob3.t4();
}
if(p4 = "1"){
   ob3.t5();
}
```

Input/Output are transition for class: This case is represented in Fig. 4. In this case, Input and Output from class is from/to transition. The class expansion is as shown in Fig. 4, where there are places in class that will store the input data and after performing functions on this data, it will be sent to other transitions for further processing. Here also transformation from class to Petri Nets model is possible in both ways, that is abstraction and refinement of class is possible.

$I(t2) = \{p1\}\ o(t1) = \{p1,p2\}$
$I(t3) = \{p2\}\ o(t2) = \{p2\}$
➔➔➔ $I(ob4) = \{t1\}\ o(ob4) = \{t4\}$
$I(t4) = \{p3,p4\}\ o(t3) = \{p4\}$

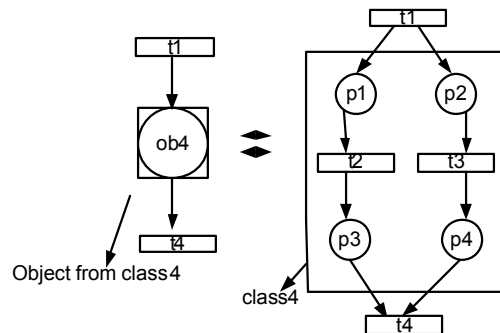

Fig. 4: A class and its object input/output are transition

```
public class class4{
  public void t1(){
    p1="1";
    p2 ="1";
  }
    public void t2(){
    p3="1";
    p1 ="0";
  }
  public void t3(){
    p4="1";
    p2 ="0";
  }
  public void t4(){
    send out p3 and p4;
  }
}

class4 ob4 = new class4();
ob4.t1();
 if(p1="1"){
   ob4.t2();
 }
 if(p2="1"){
   ob4.t3();
 }
 if(p3="1" && p4="1"){
   ob4.t4();
 }
```

NOTATIONS

The significant point in applying these notations is the Petri model simplification. To understand better, initially some explanations are given to identify the concepts of those notations in an object oriented model and then the applications [13, 14] of them are described in an example model.we may have been wondering what the big deal is with objects and object-oriented technology. Is it something we should be concerned with and if so, why? If we sift through the hype surrounding the whole object-oriented issue, we'll find a very powerful technology that provides a lot of benefits to software design. The problem is that object-oriented concepts can be difficult to grasp. And we can't embrace the benefits of object-oriented design if we don't completely understand what they are. Because of this, a complete understanding of the theory behind object-oriented programming is usually developed over time through practice. A lot of the confusion among developers in regard to object-oriented technology has led to confusion among computer users in general. How many products have we seen that claim they are object-oriented? Considering the fact that object orientation is

a software design is sue, what can this statement possibly mean to a software consumer? In many ways, "object-oriented" has become to the software industry what "new and improved" is to the household cleanser industry. The truth is that the real world is already object oriented, which is no surprise to anyone. The significance of object-oriented technology is that it enables programmers to design software in much the same way that they perceive the real world [14].

Encapsulation: Encapsulation is the process of packaging an object's data together with its methods. A powerful benefit of encapsulation is the hiding of implementation details from other objects. This means that the internal portion of an object has more limited visibility than the external portion. This arrangement results in the safeguarding of the internal portion against unwanted external access. The external portion of an object is often referred to as the object's interface because it acts as the object's interface to the rest of the program. Because other objects must communicate with the object only through its interface, the internal portion of the object is protected from outside tampering. And because an outside program has no access to the internal implementation of an object, the internal implementation can change at any time without affecting other parts of the program. Encapsulation provides two primary benefits to programmers:

Implementation hiding: This refers to the protection of the internal implementation of an object. An object is composed of a public interface and a private section that can be a combination of internal data and methods. The internal data and methods are the sections of the object hidden. The primary benefit is that these sections can change without affecting other parts of the program [15,16].

Modularity: This means that an object can be maintained independently of other objects. Because the source code for the internal sections of an object is maintained separately from the interface, we are free to make modifications with confidence that our object won't cause problems to other areas. This makes it easier to distribute objects throughout a system [16].

Example 1: As we know time is presented as " hour : minute : second" and it is clear that, 0 < hour <=12, 0 <= minute <=60 , 0<= second <=60, assume that we receive 3 numbers and we want to show the time. The input numbers can be positive or negative.

Now the same Petri model is presented by the concept of encapsulation [17].
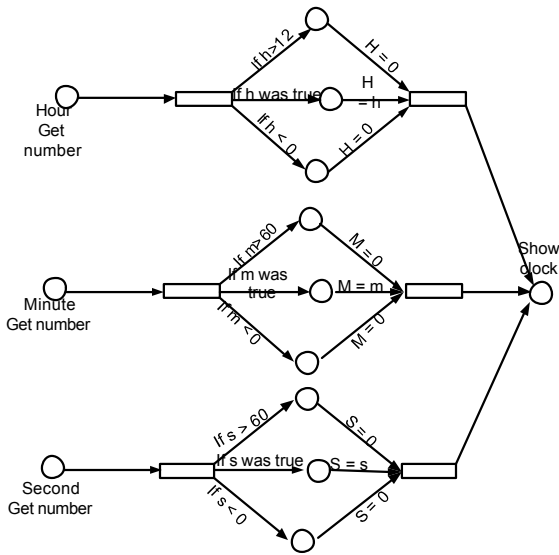
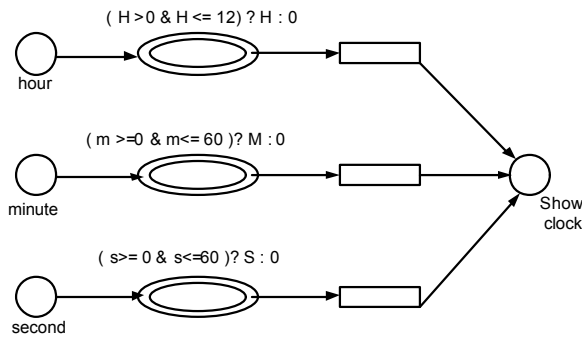Fig. 5: PN before use encapsulation specification

Fig. 6: PN after use encapsulation specification

As it is considered by comparing the Fig 5 and 6, the below one is more simple and clear than the top figure.

Class and object: Objects are software bundles of data and the procedures that act on that data. The procedures are also known as methods. The merger of data and methods provides a means of more accurately representing real-world objects in software. Without objects, modeling a real-world problem in software requires a significant logical leap. Objects, on the other hand, enable programmers to solve real-world problems in the software domain much easier and more logically. As evident by its name, objects are at the heart of object-oriented technology [17]. To understand how software objects are beneficial, think about the common characteristics of all real-world objects. Lions, cars and calculators all share two common characteristics: state and behavior. For example, the state of a lion includes color, weight and whether the lion is tired or hungry.

Lions also have certain behaviors, such as roaring, sleeping and hunting. The state of a car includes the current speed, the type of transmission, whether it is two-wheel or four-wheel drive, whether the lights are on and the current gear, among other things. The behaviors for a car include turning, braking and accelerating. As with real-world objects, software objects also have these two common characteristics (state and behavior). To relate this back to programming terms, the state of an object is determined by its data; the behavior of an object is defined by its methods. By making this connection between real-world objects and software objects, we begin to see how objects help bridge the gap between the real world and the world of software inside our computer [18].

We've dealt only with the concept of an object that already exists in a system. We may be wondering how objects get into a system in the first place. This question brings we to the most fundamental structure in object-oriented technology: the class. A class is a template or prototype that defines a type of object. A class is to an object what a blueprint is to a house. Many houses can be built from a single blueprint; the blueprint outlines the makeup of the houses. Classes work exactly the same way, except that they outline the makeup of objects. In the real world, there are often many objects of the same kind. Using the house analogy, there are many different houses around the world, but all houses share common characteristics. In object-oriented terms, we would say that our house is a specific instance of the class of objects known as houses. All houses have states and behaviors in common that define them as houses. When builders start building a new neighborhood of houses, they typically build them all from a set of blueprints. It wouldn't be as efficient to create a new blueprint for every single house, especially when there are so many similarities shared between each one. The same thing is true in object-oriented software development; why rewrite tons of code when we can reuse code that solves similar problems? In object-oriented technology, as in construction, it's also common to have many objects of the same kind that share similar characteristics [19]. And like the blueprints for similar houses, we can create blueprints for objects that share certain characteristics. What it boils down to is that classes are software blueprints for objects. As an example, the car class discussed earlier would contain several variables representing the state of the car, along with implementations for the methods that enable the driver to control the car. The state variables of the car remain hidden underneath the interface. Each instance, or instantiated object, of the car class gets a fresh set of state variables. This brings we to another important point: When an instance of
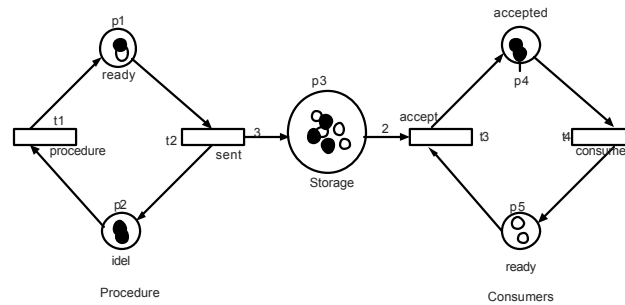
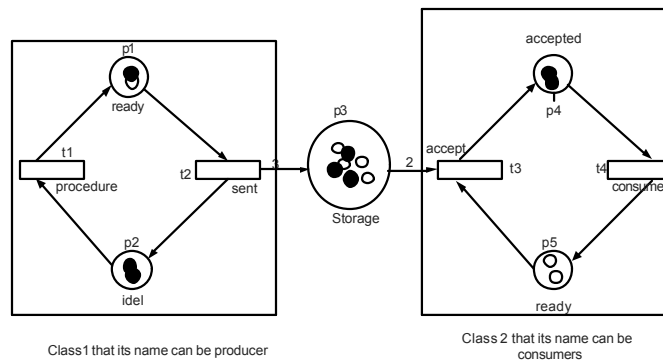Fig. 7: PN before use class and object specification
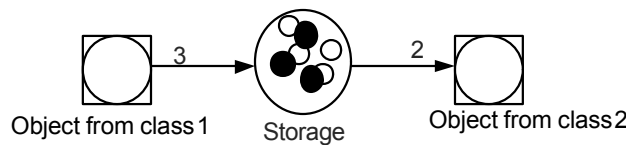

Fig. 8: PN after use class specification

Fig. 9: PN after use object specification

an object is created from a class, the variables declared by that class are allocated in memory. The variables are then modified through the object's methods. Instances of the same class share method implementations but have their own object data. Where objects provide the benefits of modularity and information hiding, classes provide the benefit of reusability. Just as the builder reuses the blueprint for a house, the software developer reuses the class for an object. Software programmers can use a class over and over again to create many objects. Each of these objects gets its own data but shares a single method implementation. Using a class notation doesn't make any change in the structure of a Petri model, but when instead of a whole class an object is used the simplification is realized. Partitioning a Petri model to some classes depends on the designer; it is possible for a designer to divide a Petri model to one or two classes and consequently one or two objects and the same model may be divided into 3 or more classes in another designer's view point [20-22].

Example 2: Producer-Consumer System

A producer-consumer system, consist of one producer, two consumers and one storage buffer with the following conditions:

The storage buffer may contain at most 5 items;
- The producer sends 3 items in each production;
- At most one consumer is able to access the storage buffer at one time;
- Each consumer removes two items when accessing the storage buffer.

A producer-consumer system
- In this Petri net, every place has a *capacity* and every arc has a *weight*.
- This allows multiple tokens to reside in a place to model more complex behaviour.

In OOPNM one can divide PN model to one or two classes and another may divide it into several classes. It
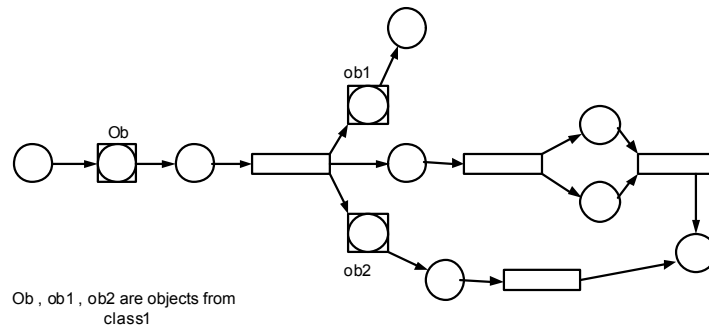
Ob , ob1 , ob2 are objects from
class1

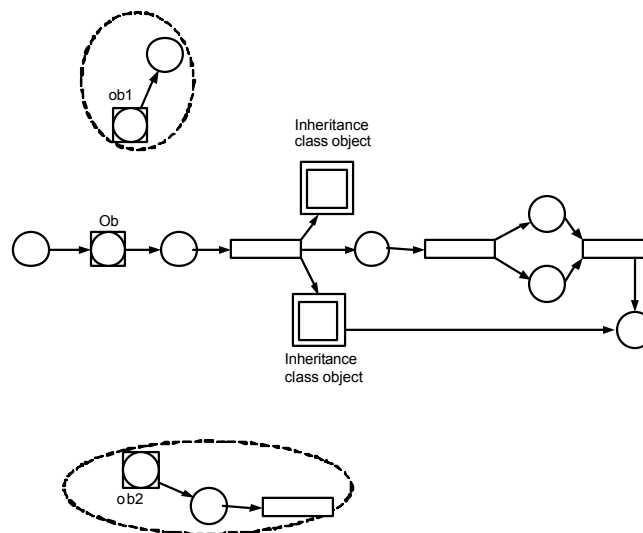Fig. 10: PN before use inheritance specification

Fig. 11: PN after use inheritance specification

has considered that when the number of classes and object creation are increased, difficult and busy Petri models become simpler [22].

Inheritance: What happens if we want an object that is very similar to one we already have, but that has a few extra characteristics? We just inherit a new class based on the class of the similar object. Inheritance is the process of creating a new class with the characteristics of an existing class, along with additional characteristics unique to the new class. Inheritance provides a powerful and natural mechanism for organizing and structuring programs. So far, the discussion of classes has been limited to the data and methods that make up a class. Based on this understanding, all classes are built from scratch by defining all the data and all the associated methods. Inheritance provides a means to create classes based on other classes. When a class is based on another class, it inherits all the properties of that class, including the data and methods for the class. The class doing the

inheriting is referred to as the subclass (or the child class) and the class providing the information to inherit is referred to as the superclass (or the parent class). Using the car example, child classes could be inherited from the car class for gas-powered cars and cars powered by electricity. Both new car classes share common "car" characteristics, but they also add a few characteristics of their own. The gas car would add, among other things, a fuel tank and a gas cap; the electric car would add a battery and a plug for recharging. Each subclass inherits state information (in the form of variable declarations) from the superclass. Inheriting the state and behaviors of a superclass alone wouldn't do all that much for a subclass. The real power of inheritance is the ability to inherit properties and methods and add new ones; subclasses can add variables and methods to the ones they inherited from the superclass. Remember that the electric car added a battery and a recharging plug. Additionally, subclasses have the ability to override inherited methods and provide different implementations for them. For
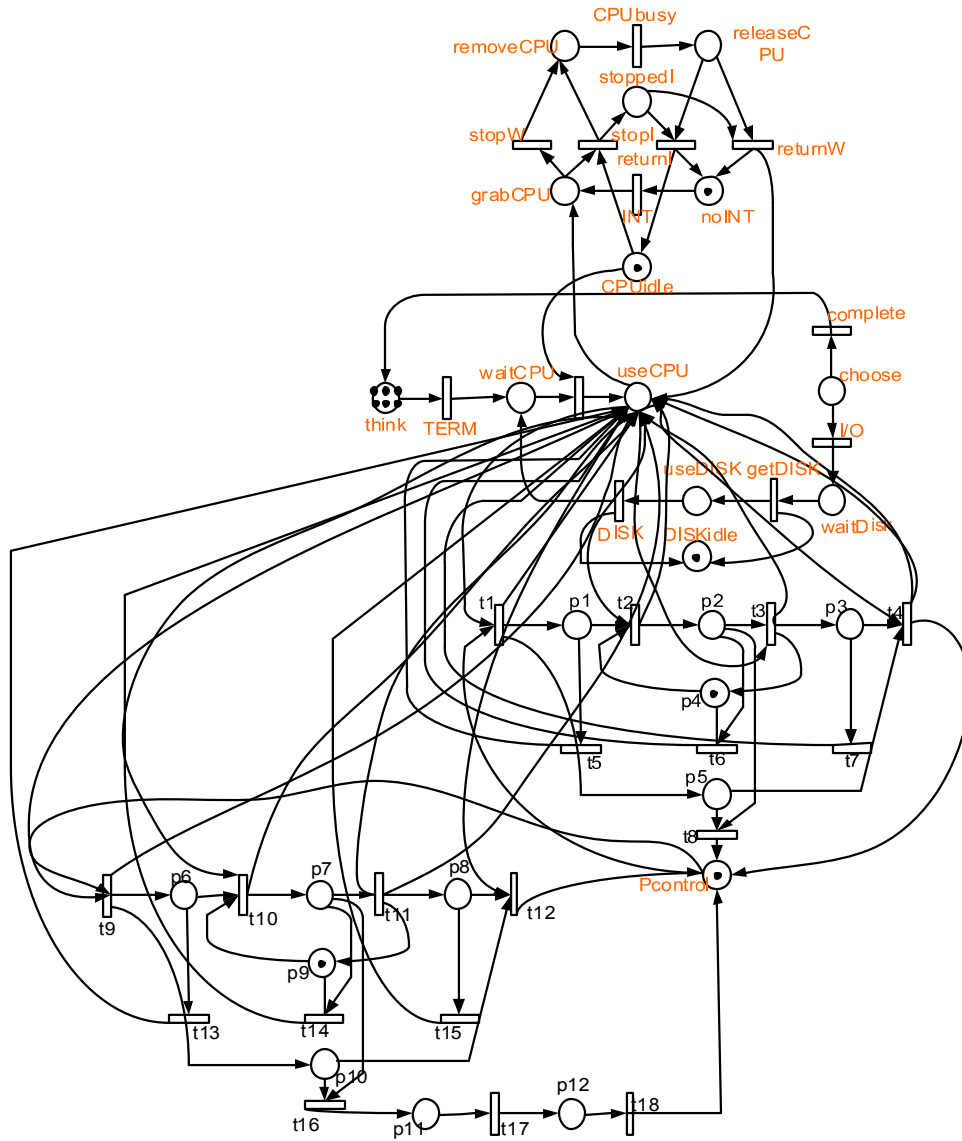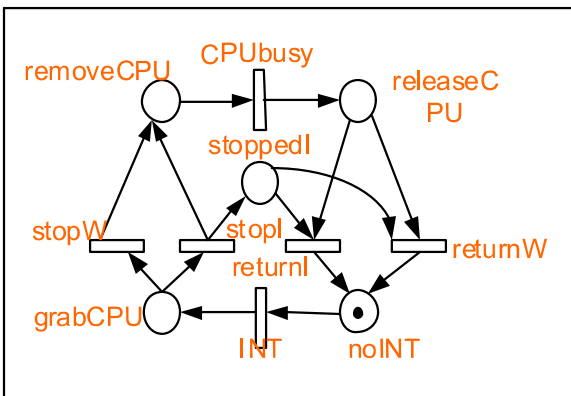
Fig. 12: PN before use notations

Fig. 13: CPU_interruption_ cycle class

example, the gas car would probably be able to go much faster than the electric car. The accelerate method for the gas car could reflect this difference.Class inheritance is designed to allow as much flexibility as possible. A group of interrelated classes is called an inheritance tree, or class hierarchy. An inheritance tree looks much like a family tree: it shows the relationships between classes. Unlike a family tree, the classes in an inheritance tree get more specific as we move down the tree [23]. We can create inheritance trees as deep as necessary to carry out our design, although it is important to not go so deep that it becomes cumbersome to see the relationship between classes. By understanding the concept of inheritance, we understand how subclasses can allow specialized data
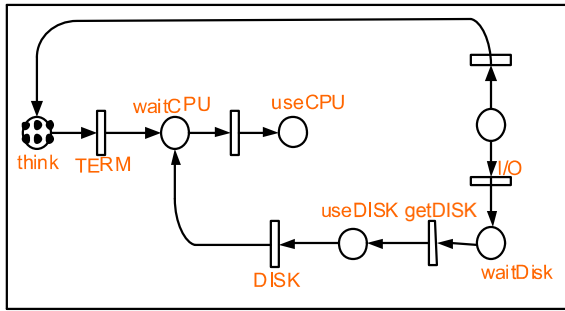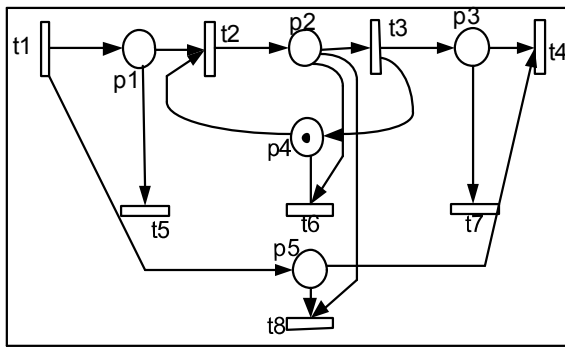
Fig. 14: Central_Server_Model class

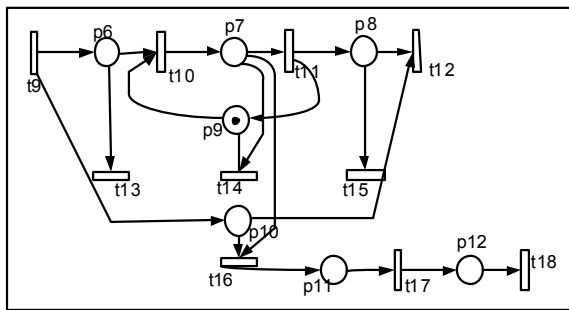Fig. 15: Enabling_memory1_conflict class

Fig. 16: Enabling_memory2_conflict class extends Enabling_memory1_conflict class

and methods in addition to the common ones provided by the superclass. This enables programmers to reuse the code in the superclass many times, thus saving extra coding effort and eliminating potential bugs. One final point to make in regard to inheritance: It is possible and sometimes useful to create superclasses that act purely as templates for more usable subclasses [24]. In this situation, the superclass serves as nothing more than an abstraction for the common class functionality shared by the subclasses. For this reason, these types of superclasses are referred to as abstract classes. An abstract class cannot be instantiated, meaning that no objects can be created from an abstract class. The

reason an abstract class can't be instantiated is that parts of it have been specifically left unimplemented. More specifically, these parts are made up of methods that have yet to be implemented--abstract methods. Using the car example once more, the accelerate method really can't be defined until the car's acceleration capabilities are known. Of course, how a car accelerates is determined by the type of engine it has. Because the engine type is unknown in the car superclass, the accelerate method could be defined but left unimplemented, which would make both the accelerate method and the car superclass abstract. Then the gas and electric car child classes would implement the accelerate method to reflect the acceleration capabilities of their respective engines or motors.

In Petri models the point of using a same structure Petri model with some variations in many places is observed. To show the concept of inheritance pay attention to the Fig. 10 and 11:

Assume that ob1, ob2, ob3 are 3 objects of a class called class1. If we want to use the concept of inheritance, the above model could be summarized as follows:

The object-oriented Petri net model OOPNM is useful because

- OOd is transformed to Petri Nets model and hence Verification and Validation is possible with object-oriented Petri net model approach.
- Object-oriented Petri net model supports Inheritance therefore incremental modification is possible. With incremental modification, enhancement[25] of the system is also possible.
- Functionalities of objects can be explored in detail, so that allows system designer to see various states of object during its execution. This helps in maintaining the software systems[24,25].
- System debugging is easier as we can verify and validate each individual object-oriented Petri net model And later we can connect them together and verify and validate the interactions among them.This also helps in adding new object-oriented Petri net model into the net. We can verify the new object-oriented Petri net model before their integration with the system and then we can correctly specify their interactions with other object-oriented Petri net model in the system and then these interactions can be verified and validated.
- OOPNM can transform difficult and busy PN model to simpler.
- Use of existing tools for both OOD and Petri Nets modeling. These existing tools can be enhanced
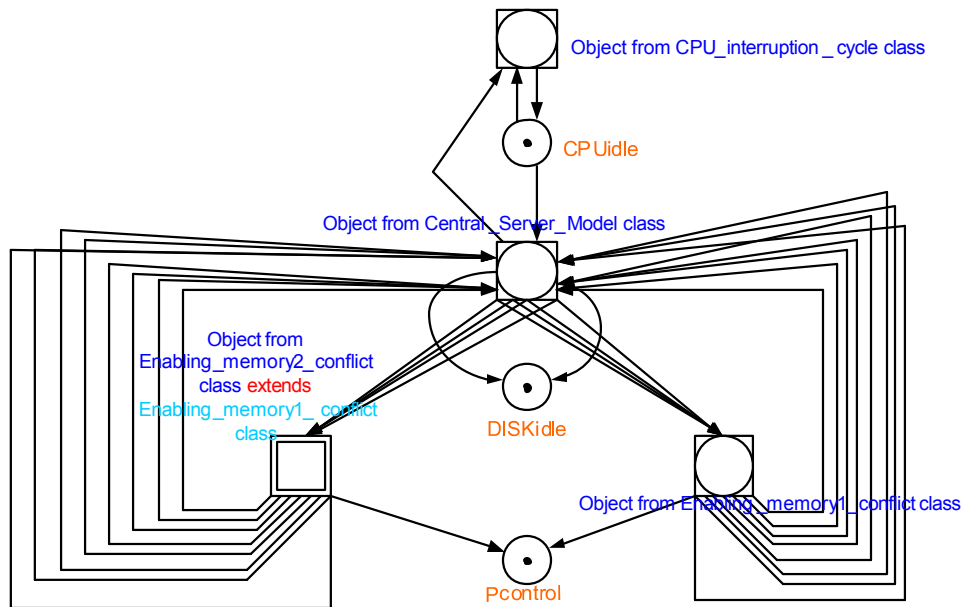
Fig. 17: PN when use notations

further to integrate this approach and hence to support direct mapping from OOD to Petri Nets models with object-oriented Petri net model.

THE CENTRAL SERVER MODEL EXAMPLE

The example chosen to illustrate this possibility is a simple central server system in which the CPU can be interrupted by the arrival of higher priority tasks as well as by failures [25].

CONCLUSIONS

The maturity and popularity of object-oriented paradigms have steadily increased. One of the main requirements in modeling and analysis[22-27] for complex and large software systems is that the design models should be unambiguous, precise and variable. To full these requirements, experts have suggested several methods which combine object-oriented method with formal methods. Although a number of high-level Petri nets[23] with the concepts of objects were suggested with a clear idea in specific concerns, they did not fully support suficient features that are needed in modeling of systems with object-oriented concepts. To solve this problem, we suggest an object-oriented Petri net model OOPNM, which supports most features of object-oriented concepts with clear semantics. Further, we describe the modeling and analysis methods for system models and making it possible to develop a complex system incrementally and iteratively. This has

been achieved from such bases as encapsulated and modularized objects, abstract information modeling, decomposition and refinement approach and incremental reachability analysis.

The object-oriented Petri net model can be transformed into CPN models that can be verified and validated by simulation or analytical methods such as place and atransition invariants. Hence, OO system can be checked for various exception cases or various behavioral scenarios. We can use existing tools that support OOD to design system and there are numerous tools that support CPN modeling.Hence, we need some in-built features in these tools that will automatically, or with human guidance, transform OOD diagrams to CPN models [25,29].

REFERENCES

1. Thayer, R.H. and M. Dortman, 1997. Software Requirement Engineering, IEEE Computer Society Press, Silver Spring.
2. Brauer, W., R. Gold and W. Vogler, 1990. A survey of behaviour and equivalence preserving refinement of Petri nets. In: APN'90, Lecture Noters in Computer Science, 483: 1-46.
3. Cardoso, J., R. Valette and D. Dubios, 1991. Petri nets with uncertain markings. In: APN'90, Lecture Notes in Computer Science, 483: 64-78.
4. Fehling, R., 1993. A concept of hierarchical Petri nets with building blocks. In: APN'93, Lecture Notes in Computer Science, 674: 148-168.

5. Huber, P., K. Jensen and R.M. Shapiro, 1991. Hierarchies in coloured Petri nets. In: APN'90, Lecture Notes in Computer Science, 483: 313-341.

6. Jensen, K., 1992. Coloured Petri Nets: Basic concepts, Analysis methods and Practical use, Springer, Berlin, Vol: 1.2.

7. Eliens, A., 1995. Principles of Object-Oriented Software Development, Addison-Wesley, Wokhingham, UK.

8. Jacobson *et al*. I., 1992. Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, Wokhingham, UK.

9. Rumbaughetal, J., 1991. Object-oriented Modeling and Design, Prentice-Hall, Englewood Cliffs, NJ.

10. Bastide, R., 1995. Approaches in unifying Petri nets and the object-oriented approach. In: Proceeding of the International Workshop on Object-oriented Programming and Models of Concurrency, Turin, Italy, June, 1995, http://wrcm.dsi.unimi.it/PetriLab/ws95/home.html.

11. Harel, D. and E. Gery, 1996. Executable object modeling with statechart. In: Proceedings of the 18th International Conference on Software Engineering, Germany, March 1996, pp: 246-257.

12. Schuman, S.A., 1997. Formal Object-oriented Development, Springer, Berlin.

13. Bsattiston, E., F.D. Cindio and G. Mauri, 1988. OBJSA Nets: A class of high-level nets having objects as domains. In: APN'88, Lecture Notes in Computer Science, 340: 20-43.

14. Biberstein, O. and D. Buchs, 1994. An object-oriented specification language based on hierarchical algebraic Petri nets. In: Proceedings of the IS-CORE Workshop Amsterdam, September 1994 (and TR: EPFL-DI 94-76).

15. Lakos, C. and C. Keen, 1994. LOOPN++: A new language for object-oriented Perti nets, Technical Report R94-4, Networking Research Group, Univesity of Tasmania, Australia.

16. Jensen, K., 1997. Coloured Petri Nets. Basic Concepts, Analysis methods and Practical Use. Vol.Vol. 1, Basic Concepts,Springer-Verlag.

17. Rumbaugh, J., I. Jacobson and G. Booch, 2005. The Unified Modeling Language reference Manual, 2nd Edn. Boston, Mass.: Addison Wesley.

18. Lakos, C., 1995. The object orientation of object Petri nets. In: Proceeding of the International Workshop on Object-oriented and Models of Concurrency Turin, Italy, (within the 16th International conference on ATPN 95).

19. Lakos, C., 1997. On the abstraction of coloured Petri nets. In: Proceedings of Petri Net Conference 97, Touloure, France.

20. Lee, Y.K. and S.J. Park, 1993. OPNets: An object-oriented high-level Petri net model for real-time system modeling. J. Syst. Software, 20: 69-86.

21. Murata, T., 1989. Petri nets: Properties, analysis and applications, Proc. IEEE, 77 (4): 541-580.

22. Perkusich, A. and J.C.A. Figueiredo, 1997. G-Nets: A Petri net based approach for logical and timing analysis of complex software systems. J. Syst. Software, 39: 39-59.

23. Some, S. and R. Dssouli, 1996. An enhancement of timed automata generation from timed scenarios using grouped states, Technical Report 1029, University of Montreal, Canada.

24. Ullman, J.D., 1998. Elements of ML Programming, ML97 Edition, Prentice-Hall, Englewood Cliffs, NJ, 164 J.-E. Hong, D.-H. Bae/Information Sciences 130 (2000), pp: 133-164.

25. Mikolajczak, B. and D. Mukhin, A Method of Concurrent Object-oriented Design Using High-Level Petri Nets. Proceedings of the IEEE International Conference on Systems, Man, Computers, SMC'98, San Diago, USA, pp: 295-300.

26. Lakos, Ch., 2002. The Challenge of Object Orientation for the Analysis of Concurrent Systems. Proceeding of the Conference on Applications and Theory of Petri Nets, Springer Verlag, LNCS-2360, pp: 59-67.

27. Bauskar, B.E. and B. Mikolajczak, 2006. Abstract Node Method For Integration of Object Oriented design with Colored Petri Nets. Proceedings of the Third International Conference on Information Technology.

28. Nihal, Y.Ö., 2007. On the Numbers of the Form $n = x2 + Ny2$, World Applied Sciences Journal, 2(1): 45-48.

29. Erçetin, S.S., Çetin, B. and N. Potas, 2007. Multi-Dimensional Organizational Intelligence Scale (Muldimorins), World Applied Sciences Journal, 2(3): 151-157.