

# Concurrent Object-Oriented Programming and Anomalies

Susmit Bagchi

Department of Computer and Information Science  
Norwegian University of Science and Technology (NTNU)  
Trondheim, Norway.

*(To fulfill the course requirement; Course DIF8901, IDI, NTNU, 2003)*

**Abstract:** The concepts of object-oriented programming and concurrency are well established. However, the formal definitions about the both are weak. The framework becomes more prone to errors when the behaviour of objects is made concurrent. In this thesis, an effort is made to establish the formal framework modeling the concurrent object-oriented programming paradigm. In addition, it is shown that in such a system the mechanism of inheritance and object synchronization may produce anomalous results, if not carefully modeled.

## 1. Introduction

The concept of object-oriented programming was conceptualized in 1960s [5]. However, the object-oriented programming research direction had become prominent in the 1980s. The first object-oriented programming language is Simula [7]. The two notable publications in this area are Byte in August 1981 that described the programming language SmallTalk and the first international conference on object-oriented programming languages and applications, held in Portland in 1986 [5].

### 1.1 The OOP

From the user point of view, object-oriented programming (OOP) is a technique for the improvement of productivity, quality [4]. The OOP paradigm brings the innovation in software development. OOP is frequently referred to as the new programming dimension [6]. The traditional procedure based software system views the procedures as the sequences of statements that specify a transformation from inputs to outputs. In 1930s and 1940s, a couple of mathematical computation models were developed such as, Church's model in 1936. Other well-known theories are Markov model in 1951 and Turing model in 1936. However, the Church's conjecture is worth to mention, which says [6]:

*“Any computation for which there is an effective procedure can be realized by a Turing machine.”*

Acceptance of Church's conjecture has a profound impact on the studies of programming languages. Actually, any language that supports simulation of Turing machine is powerful to realize any algorithm that is realizable.

The main difference between a procedure oriented approach and an OOP is that procedures act as Turing machine where as an object is a collection of such computable procedures along with a state. In addition, the objects provide persistent services over time [4]. An object - oriented system consists of a collection of interacting objects. The two main beauties of object-oriented system development are that objects are encapsulated and interact with message passing and mutate its states.

## 1.2 The concurrency

The understanding of concurrent programming is greatly influenced by the concept of sequential programming. The sequential programming assumes a set of preconditions [1] such as,

- A single CPU connected to RAM and I/O devices via a bus.
- Instructions and data are shared in RAM.
- Processor activity consists of fetch-execute cycles.
- One instruction per cycle (ignoring the pipeline architecture).

These preconditions indicate a uniprocessor machine with a Von Neumann architecture [2]. Languages such as C, C++ and Pascal are all influenced by the underlying system architecture, which is typically a above mentioned system model. This indicates that the code is, essentially, written in the order we expect the machine to execute it. This is a total ordering on a set of instructions.

**Def. Total order [8]:** *If a set  $A$  and a relation  $\mathcal{R}$  defined over  $A$  creates  $(A, \mathcal{R})$ , which is a poset, then  $A$  is totally ordered if  $\forall x, y \in A$ , either  $x\mathcal{R}y$  or  $y\mathcal{R}x$ . In this case  $\mathcal{R}$  is also a total order.*

Consider the program fragment P, Q and R where, P, Q, R are statements in a high-level language (i.e. C, C++ and Java etc.). Hence, in a sequential programming system the total ordering is presented by  $\langle P, Q, R \rangle$ .

Now, let  $P = \{p_i : 1 \leq i \leq L, L \in \mathbb{N}^+\}$ ,  $Q = \{q_j : 1 \leq j \leq L, L \in \mathbb{N}^+\}$  and  $R = \{r_k : 1 \leq k \leq L, L \in \mathbb{N}^+\}$ . Then,  $\forall e \in P, Q$  and  $R$ ,  $P \rightarrow Q \rightarrow R = \{ \{p_i\} \rightarrow \{q_j\} \rightarrow \{r_k\} \}$ .

Hence, if we choose any two instructions from the set defined as  $P \cup Q \cup R$ , then we can always rank them according to their execution order [3]. Hence, we can demonstrate that a sequential programming system two characteristics exist and they are,

- The textual order of statements specified their order of execution.
- Successive statements must be executed without any temporal overlap.

However, neither of these applied to the concurrent programming.

**Def. Concurrent Programming [3][5]:** *Operations in the source text are concurrent if they could be, but need not be, executed in parallel. Operations that occur one after the other, ordered in time, are said to be contra concurrent.*

The fundamental characteristics of the concurrent programming are,

- The notion of a process, which corresponds to a sequential computation, with its own thread of control.
- The thread of a sequential computation is the sequence of program points that are reached as control flows through the source text of the program.

The other factor that results in increased concurrency is the cost-effectiveness of using several microprocessors to achieve equivalent or better performance as a single high-end

microprocessor. This is SMP (symmetric multiprocessor) machine [1]. The examples of concurrent programming languages are,

- ACTORS (Hewitt, Guha)
- ABCL (Tokoro, Yonezawa)
- POOL (America et al.)
- Obliq (Cardelli)

In addition, there are some semantic frameworks for concurrent programming such as actor semantics. However, the extension of C++, that is COOL, is also a promising object oriented concurrent language.

In order to model concurrency, we allow the composition of process. Let, P and Q are two processes then, if P is concurrent to Q then,

$$P||Q \Leftrightarrow \exists e : \neg (P \rightarrow Q) \text{ and } \neg (Q \rightarrow P)$$

Intuitively, this means that processes P and Q execute concurrently through synchronous steps. If one process wants to perform an input action, the other process will perform a matching output action. Thus we have a partial ordering.

**Def. Partial Order [8]:** A relation  $\mathcal{R}$  on a set A is a partial order if,

- $\forall x \in A, \langle x, x \rangle \in \mathcal{R}$
- $\forall x, y \in A, \text{ if } \langle x, y \rangle \in \mathcal{R} \text{ and } \langle y, x \rangle \in \mathcal{R} \text{ then, } x=y$
- $\forall x, y, z \in A, \text{ if } \langle x, y \rangle \in \mathcal{R} \text{ and } \langle y, z \rangle \in \mathcal{R} \text{ then, } \langle x, z \rangle \in \mathcal{R}$

Hence, given two instructions we may or may not be able to rank them according to their execution order. So, we may characterize a concurrent system as [1],

- The textual order of statements does not specify their order of execution.
- The operations are permitted (but not obliged to) overlap in time.

Again, consider there are three threads {p, q, r} executing concurrently. So, we say  $p||q||r$ . The concurrency  $p||q||r$  has properties such as [1],

- $p||q = q||p$
- $(p||q)||r = p||(q||r)$
- $(p||q) \wedge (q||r) \Rightarrow (p||r)$

Hence, if  $P_1, P_2, \dots, P_n$  are such a set of concurrent processes then the total combinatorial execution order is given by  $(|\cup_{j=1, n} P_j|)!$ .

Now, a process P may terminate successfully or may fail. In addition, the process P will have multiple intermediate states of transition  $p', p'', \dots, p^n$ . Let,  $t: P \rightarrow T$  be a termination detection function where,  $T = \{s, f\}$ . Hence, a process P can be expressed as,

$$P: [ p' \rightarrow p'' \rightarrow \dots \dots \dots ( p^n \text{ OR } t(P) ) ] \text{ where, } p^n \text{ may be expressed as } p^n \rightarrow p^x, 1 \leq x \leq n.$$

The set of actions a process can perform is called as alphabet of a process,  $\alpha P = \{a, b, \dots\}$ .

## 2. The object calculus and concurrent process expression

The concurrent object-oriented programming language development has suffered from the lack of any widely accepted formal foundations for defining the semantics [2]. In addition, the relationship between object-oriented features supporting the reuse and other features related to state changing and object interaction is poorly understood under the concept of concurrency [2]. The object-oriented language is an effective approach to achieve reusable software components. However, object-oriented model should be capable of handling concurrency and distribution. It is shown in [9] that,

- Most object-oriented languages lack a well-defined semantic foundation.
- Clean integration of concurrency reusable objects and encapsulation concepts are difficult to achieve.
- Composition of concurrent objects is poorly understood.

Until recently, the research in concurrency model establishment and the research in OOP were independent of each other. The earliest approach to resolve this separation was ACTOR model [7]. The other promising direction is to use process calculi as a semantic foundation for concurrent object-oriented languages [9]. The ACTOR model of computation bears comparison to the agent based models. However, it is based on asynchronous communication.

### 2.1 Requirement for an object calculus

There are three fundamental aspects of concurrent object-oriented languages that we would like to capture.

- Encapsulation.
- Active objects.
- Composition.

Encapsulation: Objects are processes that encapsulate services. The communication with an object is message based request-reply communication. Objects also have internal states and these states may or may not change. Object states are indirectly accessible through services provided by an object. All the objects will have unique names to identify them.

Active object: Objects are entity and autonomous. In addition, objects may be concurrent internally.

Object composition: Objects may be composed of a set of primitive objects. Objects may be specified as the functional composition of higher-order abstractions over objects and services. While composing objects overriding of inherited services is possible.

## 3. Concurrent Object calculus

We have said before that objects interact through request-reply message communication and an object is a collection of encapsulated states and methods. The  $\pi$ -calculus is a mathematical model [10] of processes whose interconnection changes as they interact. We begin with a

sequence of definitions and conventions. It is assumed that a potentially infinite set of names  $N$ , ranged over by  $a, b, \dots, z$ , which will function as all of communication ports, variables and data values, and a set of identifiers ranged over by  $A$ , each with a fixed non-negative arity. The objects, ranged over by  $P, Q$  are defined below [3][10].

### 3.1 Prefixes

$P, Q, R ::= 0$	(inert process)
$a(x).P$	(input prefix)
$a \hat{x}.P$	(output prefix)
$P Q$	(parallel composition)
$(vx)P$	(restriction)
$!P$	(replication)
$\tau.P$	

### 3.2 The (Abelian) monoid properties

- 1)  $P|Q \equiv Q|P$  (Commutativity of parallel composition)
- 2)  $(P|Q)|R \equiv P|(Q|R)$  (Associativity of parallel composition)
- 3)  $P|0 \equiv P$

### 3.3 The scope extension property

$((vx)P)|Q \equiv (vx)(P|Q)$  if  $x \notin FV(P)$  ( $(vx)$  is non-distributive;  $FV$  are free-variables)

### 3.4 Replication/ Creation

$$!P \equiv (P|_n)!P$$

From the properties defined above, we see that the objects can be of the following forms.

1. The empty object is represented by '0' and it cannot perform any actions.
2. An output method of an object is defined as  $a \hat{x}.P$ . This means that the data  $x$  is sent along the name 'a' and thereafter the object continues as  $P$ . So,  $a \hat{x}$  can be thought of as an output port and  $x$  as a datum sent out through that port.
3. An input method of an object is defined as  $a(x).P$ , meaning that a data is received along port  $a$  and  $x$  is a placeholder for the received data. So,  $a$  can be thought of as an input port and  $x$  as a variable which will get its value from the input along  $a$ .
4. A silent prefix is defined as  $\tau.P$ , which represents an object that can evolve to  $P$  without interaction with the environment.
5. A sum of two objects is defined as  $P+Q$  representing an object that can enact either as  $P$  or as  $Q$ .
6. A parallel composition is given by  $P|Q$ , which represents the combined behaviour of  $P$  and  $Q$  executing in parallel. The objects  $P$  and  $Q$  may also communicate if one performs an output and the other an input along the same port.
7. A match is defined in phrase "if  $x=y$  then  $P$ ". Hence, an object will behave as  $P$  if  $x$  and  $y$  are the same name, otherwise the object will perform nothing.

8. A mismatch is phrased as “ $x \neq y$  then P”. The object will behave as P if x and y are not the same name, otherwise the object performs nothing.
9. A restriction is given as  $(\forall x)P$ . An object will behave as P where the name x is local, meaning it cannot immediately be used as a port of communication between P and its environment. However, it can be used for communication between components within the object P.
10. An identifier is represented by  $A(y_1, y_2, \dots, y_n)$  where n is the arity of A. Every identifier has a definition  $A(x_1, x_2, \dots, x_n) = P$  where the  $x_i$  must be pair wise distinct, and  $A(y_1, y_2, \dots, y_n)$  behaves as object P with  $y_i$  replacing  $x_i$  for each i. So, a definition can be thought of as a process declaration with  $x_1, \dots, x_n$  as formal parameters, and identifier  $A(y_1, y_2, \dots, y_n)$  as an invocation with actual parameters  $y_1, \dots, y_n$ . As an example, of the test operation on an object we consider that an object P receives a name and continues as Q if the name is y and as R if the name is z. So, formally we write,  $P = a(x).(x \hat{=} (y.Q + z.R))$ .
11. The abelian monoid properties says that for some objects P and Q, P and Q holds abelian and monoid properties if the commutativity and associativity properties are defined over P and Q. In addition, the composition with null object should also be defined.

#### 4. Object-oriented concurrency

In computer science, the concept of data abstraction is the basic necessity. The introduction of abstract data type (ADT) enables a software designer to design his own data type according to his requirement [9]. Object-oriented programming adds to abstract data types the important ability to incrementally extend an existing type. A large number of experimental concurrent object-oriented languages are developed. The technical factors motivating concurrent object-oriented languages include at least the three factors. These are described below.

1. The application designer feels free from the burden of management of application control flow and the organization of objects. Hence, the object-oriented model has unified the processor and the memory model. This is an advantage to application developer because it avoids presenting the two semantic models – one model for control and a different model for data and functions.
2. The modeling fidelity is enhanced in applications where autonomous and real world entity are pervasive.
3. A concurrent object-oriented language surpasses substantial detail about the control flow management, specially the invocation and scheduling of object execution. By contrast, sequential languages force the developer to explicitly construct a invocation and scheduling sequence.

##### 4.1 Object-oriented concurrency issues

There are mainly four issues related to object-oriented concurrency. We are describing these four distinct features in short.

###### 4.1.1 Object interaction primitive

This issue deals with the object interaction and synchronization primitives. There are two basic models such as synchronous and asynchronous communication models. In the synchronous object interaction primitive, the client object gets blocked (waits) until the server object outputs the response message to client object. However, in the asynchronous interaction model, the client object does not wait for any message from server.

#### 4.1.2 Object granularity

This parameter deals with the size of the objects. It is the bias that a language offers toward the size of an object. For an example, ACTOR offers fine-grain objects but SmallTalk offers a coarse grain object primitive.

#### 4.1.3 Encapsulation

The concept of encapsulation is a very common concept in an object-oriented programming language construct. The SmallTalk language offers the multiple instances of a class variable, which can be updated concurrently. This flexibility also invites the problem of “interference” in concurrent system among the same kind of objects existing simultaneously.

#### 4.1.4 Class consistency

A few concurrent object-oriented languages offer the possibility of object migration and independent class definition modification. This invites the problem of class consistency in a object-oriented concurrent system.

### 5. A few concurrent object-oriented languages

#### 5.1 The ACTOR model

Actors originated through Carl Hewitt work on the artificial intelligence system *Planner* in the early 1970's. Actor approach was formulated around three main design objectives [7][11]:

- **Shared, mutable data.** The actor model is designed to deal with shared resources that may change state. An example is a bank account whose balance may change.
- **Reconfigurability.** New objects may be created and it is possible to communicate with new objects after they are created.
- **Inherent concurrency.** It should be possible to understand the "inherent concurrency" of a program by examining it.

The phrase "inherent concurrency" refers to the number of activities that could be carried out in parallel if there were an unbounded number of processes available.

**Def.** *An actor is an object that carries out its actions in response to communications it receives. Actors support large-scale concurrent symbolic computation. There are three basic actions that an actor may perform [11].*

- Send communication to itself or other actors
- Create new actors
- Specify a *replacement behavior*, which is essentially another actor that takes the place of the actor that, creates it, for the purpose of responding to certain communications.

A few important points to remember about ACTOR model are mentioned below.

- Each actor receives a linearly ordered sequence of communications.
- There is no assignment to local variables in the basic actor model.
- One execution of an actor script is an atomic computation.
- An important part of the model is the *mail system*, which routes and buffers communication between actors. Every communication must be sent to a *mail address* and an actor may communicate with any actor if it knows its mail address.

One important character of the Actor model is that this model is *asynchronous*, with no *global clock*. Hence, the communication primitives are explicit, through mail addresses, *without shared variables*. In addition, all communication in ACTOR model are buffered and asynchronous. A weak fairness is assumed, in the following forms:

- Every message that is sent is eventually delivered. (However the notion of "deliver" is subtle, because an actor may or may not be prepared to process a message at a given time. More discussion below.)
- Every computation eventually progresses, i.e., each process is eventually scheduled and are allowed to perform part of its computation.

ACTORS are characterized by:

- Identity (*never changes once an actor created*)
- Current behaviour (*indicates actor's action on next message*)

**Immutable actor:** This is a special kind of ACTOR having same "action behaviour" throughout its lifetime.

### 5.1.1 Actor Classes

Primitive actor: This kind of ACTOR corresponds to atomic types such as *characters*. The primitive actors are sent directly in messages.

Non-primitive actor: This kind of ACTOR has *mail-address*, which represents actor's *identity*. The *current behaviour* of such ACTOR is composed of a set of instance variables or local state of the actor.

The example of an ACTOR is given below [11].

```
(defBehaviour simple-check-acc [balance]
  (script
    [[:deposit amount customer]
      (become check-acc[(send balance[+amount] ) ] )
      (send customer [:deposited amount] ) ]
     [[:withdraw amount customer]
      (become check-acc[(send balance [-amount] ) ] )
      (send customer [:withdraw amount] ) ]
     [[:balance customer]
      (send customer [: balance balance] ) ] ) )
```

## 5.2 The ABCL model

**5.2.1 ABCL/1 model:** ABCL/1 has evolved from the ACTOR but, more pragmatic; adopts more eclectic view toward object coexistence.

**5.2.2 Object in ABCL/1:** The object and its behaviour is specified entirely by its script, which is a collection of message patterns and associated methods.

**5.2.3 Message passing in ABCL/1:** There are two kinds of communication primitives are there such as, asynchronous and point to point.

### 5.2.4 Message types in ABCL/1:

-*past* ( meaning, send a message and resume the activity)

-*now* (meaning, send a message and wait for the response)

-*future* (meaning, send a message, mention the future action which should be carried out by the receiver and resume the sender's current activity.)

**5.2.5 Message priorities:** In the ACTOR model no such message priority is defined. But, as an improvement, the ABCL/1 contains such message priorities [11]. There are two types of priorities in ABCL/1 such as, *express\_message* and *ordinary\_message* [11]. On the reception of an *express\_message* will preempt the *ordinary\_message* processing temporarily and the ABCL/1 object will process the express message. When the *express\_message* processing is over, the object will resume from its preemption point.

**5.2.6 Message queues and Object states:** The ABCL/1 contains a set of message queues. There exist two message queues per object such as, *express\_message\_queue* and *ordinary\_message\_queue* [7][11]. In addition, there are three possible object states such as,

-**dormant** (object currently idle, will do something on message arrival)

-**active** (object currently processing a message, executing a method)

-**waiting** (while processing some message, a message is sent and waiting for the reply)

## 6. The inheritance anomaly

An object is a representation of encapsulated state and the code in the form of instance variables and methods respectively. In object-oriented languages types and names of instance variables and the signatures of the methods are typically declared in a class definition. The concurrent behaviour of an object is captured in part by the static class definition of the object and in part by the dynamic mechanism employed by the method interface to guarantee synchronization. The inheritance anomaly occurs when an attempt is made to specialize concurrent behaviour using an inheritance mechanism. The anomaly occurs when a subclass violates the synchronization constraints assumed by the base class [5]. Ideally, all the methods of the base class should be reusable. But, if the synchronization constraints are defined by the superclass in a manner prohibiting incremental modifications through inheritance, the methods cannot be reused, they must be re-implemented to reflect the new constraints. Hence, the inheritance mechanism becomes useless [5].

## 6.1 Specifying object behaviour

The behaviour of an object can be defined as a set of behavioural equations that capture the states of an object and the subset of methods that are visible when the object is in a particular state. The behaviour of an object that maintains some prescribed linear order over a collection of items, is defined. The observable behaviour of an object representing a bounded linear order is completely described by the following equations.

Let,  $V = \{x_1, x_2, \dots, x_n\}$  be a set of variables and an object  $A$  is defined as,  $A = \{A_0, A_1, \dots, A_n\}$  where,  $\forall A_i \in A$ ,  $A_i$  represents object transformations and  $\Gamma = \{a, b, \dots, z\}$  be a set of ports through which objects can import or output values in  $V$ . Hence [5],

$$\begin{aligned} A_0 &= a(x_1).A_1(x_1) \\ A_1(x_1) &= a(x_2).A_2(x_1, x_2) + \hat{a}x_1.A_0 \\ A_n(x_1, x_2, \dots, x_n) &= \hat{a}x_1.A_{n-1}(x_1, x_2, \dots, x_{n-1}) \end{aligned}$$

The equations capture precisely the states that an object representing a bounded linear order may occupy during its lifetime. In the behaviour defined for  $A_1$  object state, it states that  $A_1$  object state can perform an input and can perform an output to its client. The behaviour equation of an object also defines the replacement behaviour of an object. The notion of replacement behaviour is a fundamental aspect of the ACTOR model. Hence, it is reasonable to use the behaviour equations as a formal means for specifying the behaviour of individual objects with ACTOR like semantics.

## 6.2 Object states and behaviour sets

In this section a model is defined that captures the essential elements used in developing a programming abstraction to represent a collection of behaviour equation [5]. Let us assume that every object is associated with an object state  $\delta_i$  and a set  $\beta_i$  termed as a set of observable behaviour of the object. The collection of all object states is  $S = \{\delta_1, \delta_2, \dots, \delta_n\}$ . The set of all possible observable behaviour sets is the power set,  $B = P(M)$ , where  $M = \cup_{i=0, n} \beta_i$ . The function  $f: S \rightarrow B$  maps the objects states into the observable behaviour of an object at that state. The function  $f$  is called behaviour function.

## 6.3 Inheriting concurrent behaviour

The types of concurrent object-oriented systems of interest are composed of objects with concurrency properties consistent with ACTOR model [3][5][7]. Each object possesses its own thread of control and communicates with other objects via message passing. Concurrency in a system is limited to inter-object concurrency [5]. In order to overcome the inheritance anomaly, one needs to modify the required methods in the inherited subclass, which is the source of conflict. As an example, suppose, in inherited object  $A$ , the  $\hat{a}x_i$  is the source of synchronization problem. In this case the  $\hat{a}x_i$  method can be replaced by another similar method termed as  $\hat{b}x_i$ . Hence, the behavioural equation of the object  $A$  will become [5][10],

$$\begin{aligned} A_0 &= a(x_1).A_1(x_1) \\ A_1(x_1) &= a(x_2).A_2(x_1, x_2) + \hat{a}x_1.A_0 \\ A_2(x_1, x_2) &= a(x_3).A_3(x_1, x_2, x_3) + \hat{a}x_1.A_1(x_1) + \hat{b}x_1|x_2.A_0 \\ A_n(x_1, x_2, \dots, x_n) &= \hat{a}x_1.A_{n-1}(x_1, x_2, \dots, x_{n-1}) + \hat{b}x_1|x_2.A_{n-2}(x_1, x_2, \dots, x_{n-2}). \end{aligned}$$

Hence, we have incorporated two refinements in the system. First, the  $\hat{b}$ xi operation is added to the appropriate observable behaviour sets and a new power set  $B$  is computed. Second, the  $A1(x1)$  behaviour is clearly a distinguishing behaviour. So, we may define a new mapping function  $f_{\sim}$  as [5],

$$\begin{aligned} f_{\sim}(\delta 0) &= f(\delta 0) \\ f_{\sim}(\delta 1) &= f(\delta 1) \\ f_{\sim}(\delta 2) &= \{a, \hat{a}, \hat{b}\} \end{aligned}$$

## 7. Summary

This thesis has described the basic concept of the object-oriented programming paradigm in a formal way for the understanding of the concurrency issues and its problem. This paper also describes the basic formalism to model concurrency and presents the ACTOR model as the practical examples in the domain of object-oriented concurrent system. Later, we describe the improvement made in ABCL programming paradigm, which is a sophisticated extension of basic ACTOR model. However, it appears that relationship between concurrent object behaviour and inheritance is the source to few anomalies. It is shown that proper emphasis should be offered to the relationship between the states of an object and subsets of methods that are visible in the interface to the object. It is called as behaviour sets of an object. If the inheritance anomaly has to be avoided, the behaviour sets and the functions of the objects should be inheritable and mutable. As a final note, it can be stated that the present day object-oriented programming languages, that is C++, has addressed this issue with sufficient expression.

## References

- [1] Institute of Information Science and Technology, April 2000, Massey University, New Zealand, [www-ist.massey.ac.nz/csnotes/355/lectures](http://www-ist.massey.ac.nz/csnotes/355/lectures).
- [2] Yonezawa. A., Tokoro. M., Neirstrasz. O., Wegner. P., 1992, Towards an Object Calculus, In proceedings of the ECOOP, LNCS 612, Springer-Verlag, pp. 1-20.
- [3] Pfenning. F., November 2002, Supplementary Notes on Concurrent Processes, Lecture 23, pp. 15-312.
- [4] Guerraoui. R. et al., 1996, Strategic Directions in Object-Oriented Programming, ACM Computing Survets, Volume 28, No. 4.
- [5] Kafura. D., Lavender. G., 1994, Concurrent Object-Oriented Languages and the Inheritance Anomaly, in Parallel Computers: Theory and Practice, ed: T.L. Cassavant, IEEE Press, pp. 165-198.
- [6] Budd. T., 2002, An Introduction to Object-Oriented Programming, 3<sup>rd</sup> edition, Addison Wesley, pp. 4-7, 26-34, 161-182.
- [7] Oscar. N., 1989, Survey of Object-Oriented Concepts, ACM press.
- [8] Grimaldi. R., 1994, Discrete and Combinatorial Mathematics, 3<sup>rd</sup> Edition, Addison Wesley, pp. 349, 353-356, 374-376.
- [9] Papatomas. M., 1989, Concurrency Issues in Object-Oriented Programming Languages, In Object-Oriented Development, Univ. of Geneva, pp. 207-245.
- [10] Parrow. J., 2001, An Introduction to the pi-Calculus, Handbook of Process Algebra, ed. Bergstra, Ponse and Smolka, Elsevier Science, pp. 479-543.
- [11] Chris. T., Mark. S., 1989, Concurrent Object-Oriented Programming, ACM press.