World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:7, No:6, 2013

CScheme in Traditional Concurrency Problems

Nathar Shah and Visham Cheerkoot

Open Science Index, Computer and Information Engineering Vol:7, No:6, 2013 publications.waset.org/7402/pdf

***Abstract*—**CScheme, a concurrent programming paradigm based on scheme concept enables concurrency schemes to be constructed from smaller synchronization units through a GUI based composer and latter be reused on other concurrency problems of a similar nature. This paradigm is particularly important in the multi-core environment prevalent nowadays. In this paper, we demonstrate techniques to separate concurrency from functional code using the CScheme paradigm. Then we illustrate how the CScheme methodology can be used to solve some of the traditional concurrency problems – critical section problem, and readers-writers problem - using synchronization schemes such as Single Threaded Execution Scheme, and Readers Writers Scheme.

***Keywords*—**Concurrent Programming, Object Oriented Programming, Environments for multiple-processor systems, Programming paradigms.

I. Introduction

WHILE Moore's Law [1] is still applicable today, computer engineers have shifted their focus from increasing transistor density for the boosting of clock rate to instead putting more execution units (CPU cores) on a single CPU die [2]. Consequently, this adds extra burden on developers because of the need to explicitly parallelize their applications in order to take full advantage of the increasing number of cores that each successive multi-core generation will provide [3]. However, the conventional approaches to parallel programming are error prone that it results in problems like race condition, deadlock, livelock, and starvation.

CScheme [8] is a scheme based programming paradigm that is designed to alleviate concurrency problems. In the approach, programmers express concurrency semantics through a set of predefined schemes that can be configured through its inherent synchronization units. Effective separation of concurrency from the functionality is achieved through this approach. This thereby leaves the engineering of software functionality to the application developers, and the concurrency deployment to the parallel programming experts.

In the CScheme, it makes use of aspect-oriented programming [4] as the main approach to separation. This technique has proved to increase software modularity in practical situations where object-oriented programming does not offer an adequate support. As such, we use AspectJ, a Java based aspect oriented language. A detailed description of AspectJ is presented in many papers [5] at the AspectJ site.

Using AspectJ, all the synchronization concerns of a particular object-oriented program can be separated thereby alleviating the need for a programmer to take care of synchronization issues while developing an object-oriented application. Such synchronization concerns can be configured using AspectJ's aspects. By using CScheme tool, these synchronization aspects do not need to be developed by the programmer. Instead, we provide Synchronization Schemes, which are pre-configured templates that define specific thread coordination and communication mechanisms. Moreover, each Synchronization Scheme comprises of several Synchronization Units that can be further configured to take care of specific synchronization issues. Furthermore, the provided Synchronization Schemes can be composed to build custom developer-defined Synchronization Schemes based on the synchronization needs of the object-oriented software that he/she is developing.

The technique is further explained in the next section. Our objective in this paper is to demonstrate techniques to apply CScheme paradigm on popular concurrency problems like critical section problem and readers-writers problem. The techniques make use of the Single Threaded Execution Scheme, and Reader-Writer Scheme. Those schemes contain set of synchronization units that can be manipulated by different techniques. Our contributions in the paper are as follows:

a) Illustrate an effective mechanism to separate functional code from the concurrency constructs.
b) Device a technique using the CScheme paradigm for the critical section problem.
c) Device a technique using the CScheme paradigm for the readers-writers problem.

Section II describes Scheme-based concurrent programming while Section III describes how the CScheme paradigm can be used in some of the traditional concurrency problems. Section IV discusses related work. Finally, in Section V, we will make some future work remarks.

II. CScheme – A Concurrent Scheme Based Concurrent Programming

CScheme is a paradigm for concurrent programming based on schemes [8]. A scheme is a generalization of a pattern of same semantic. CScheme encapsulates a combination of synchronization semantics in a generic manner such that those semantics can be reused over-and-over again and be composed in programs of similar need. The building blocks of a synchronization scheme are called synchronization units, which encapsulate synchronization as well as thread

Nathar Shah is with the Faculty of computing and informatics, Multimedia University, 63100 Cyberjaya, Malaysia (e-mail: nathar@mmu.edu.my).

Visham Cheerkoot was with the faculty of computing and informatics, Multimedia University, 63100 Cyberjaya, Malaysia (e-mail: visham.cheerkoot@mmu.edu.my).

Open Science Index, Computer and Information Engineering Vol:7, No:6, 2013 publications.waset.org/7402/pdf

interaction mechanisms.

In any concurrent applications, the risks are in concurrent access to shared data and thread coordination. Among the risks in shared data accesses are sharing mutable static variables across threads, changing the instance on which we synchronize on one part of the program, synchronizing on string literals and autoboxed values, improper guarding of non-atomic operations, etc. Thread coordination risks are in improper use of coordination primitives like wait() and notify(). As a result of these risks, performance issues like deadlock, starvation, and livelock arise. If one thread invokes a method frequently, other threads that also need frequent synchronized access to the same object will often be blocked [6].

The CScheme approach alleviates the risks by enabling the programmers to deal with the threads in a controlled manner through synchronization units' manipulation. For example, Shah et. al.[8] have developed synchronization schemes like Single Threaded Execution Scheme, Readers Writers Scheme, Guarded Suspension Scheme, and Thread Coordination Scheme where their respective synchronization units can be configured to the specifics of an application. The configuration then regulates how thread coordination and access to shared variables/resources happens. On the other hand, in CScheme, a customized synchronization scheme is created using a GUI Scheme builder tool that allows multiple synchronization units or even compatible synchronization schemes to be composed and checked using CheckThread [7] engine, a thread checker. The thread checker makes use of static analysis to find concurrency bugs at compile time.

Complete detail on the CScheme design and its design logic are presented in [8].

III. CScheme for Concurrency Applications

This section demonstrates the usage of several synchronization schemes and synchronization units presented in the previous section. We consider a few case studies which are well-known illustrative examples of common computing problem in concurrency.

There are several classic synchronization problems that have been invented to demonstrate synchronization primitives. However, these problems are not necessarily analogous to real-world problems but they do illustrate principles real solutions should use.

A. CScheme for Critical Section Problem

The Critical Section Problem is a classic synchronization problem that demonstrates a set of instructions that must be controlled so as to allow exclusive access to a one process only. This problem highlights the fact that execution of the critical section by processes must be mutually exclusive in time.

Consider the code sample in Fig. 1 below:

```
public class TrafficSensorController implements TrafficObserver{

    private int vehicleCount = 0;
    public void vehiclePassed(){
        vehicleCount++;
    }
    public int getAndClearCount(){
        int count = vehicleCount;
        vehicleCount = 0;
        return count;
    }
}
```

Fig. 1 Not Synchronized Code

This code shows that the instance variable "vehicleCount" is subject to non-atomic changes in methods vehiclePassed() and getAndClearCount(). If multiple threads attempt to execute these methods simultaneously, there is a high possibility of data corruption as well as lost updates.

Using Java, the critical section problem in the above case can be solved as follows by making use of the "synchronized" synchronization construct (Fig. 2):

```
public class TrafficSensorController implements TrafficObserver{
    private int vehicleCount = 0;
    public synchronized void vehiclePassed(){
        vehicleCount++;
    }
    public synchronized int getAndClearCount(){
        int count = vehicleCount;
        vehicleCount = 0;
        return count;
    }
}
```

Fig. 2 Error prone synchronization

Below, we show the approach of using our tool.

First, mark any field that will be subject to concurrent access by threads with the "@SharedField" annotation (Fig. 3).

```
import annotations.SharedField;
public class TrafficSensorController implements TrafficObserver{
    @SharedField
    private int vehicleCount = 0;
    public void vehiclePassed(){
        vehicleCount++;
    }
    public int getAndClearCount(){
        int count = vehicleCount;
        vehicleCount = 0;
        return count;
    }
}
```

Fig. 3 Annotating shared field

Then launch the CScheme GUI tool. The next step is to choose the desired resource file and scheme and its associated synchronization unit that we want to apply to the chosen resource file (Fig. 4).

Open Science Index, Computer and Information Engineering Vol:7, No:6, 2013 publications.waset.org/7402/pdf

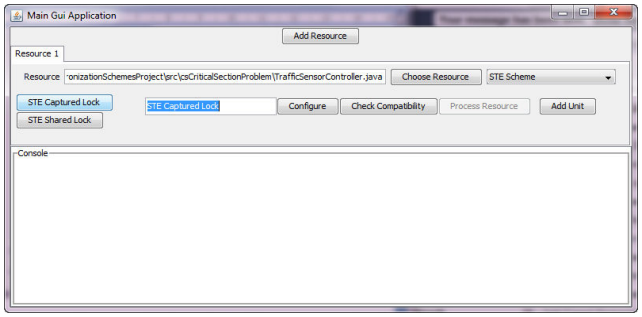


Fig. 4 Configuring the scheme with its synchronization units

The next step is to configure the selected scheme as shown below (Fig. 5). Here choose the methods in the source code that would be guarded by the Captured Lock unit.

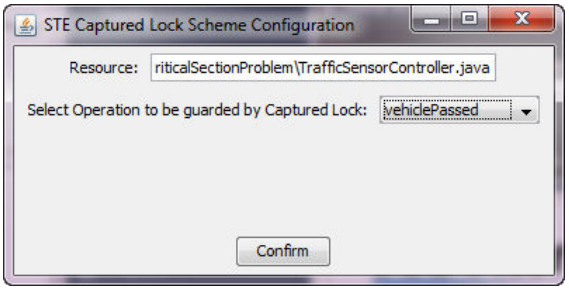


Fig. 5 Configuring Captured Lock Synchronization Unit

We add one more Captured Lock unit for the other methods that need to guard the shared instance variable "vehicleCount" from concurrent access.

After the scheme has been applied to the resource file, the latter looks as follows (Fig. 6):

```
import my.com.checkthread.engine.annotations.ThreadSafe;
import annotations.CapturedLock;
import annotations.SharedField;
import annotations.SingleThreadedSchemeCapturedLockManaged;

@SingleThreadedSchemeCapturedLockManaged
public class TrafficSensorController implements TrafficObserver{
    @SharedField
    private int vehicleCount = 0;
    @CapturedLock
    @ThreadSafe
    public void vehiclePassed(){
        vehicleCount++;
    }
    @CapturedLock
    @ThreadSafe
    public int getAndClearCount(){
        int count = vehicleCount;
        vehicleCount = 0;
        return count;
    }
}
```

Fig. 6 The generated code after the configuration

As it can be noted, vehiclePassed() and getAndClearCount() are now guarded by the "CapturedLock" from the Single Threaded Execution Scheme. This will ensure that threads access these methods to read/modify vehicleCount atomically.

The Shared Lock synchronization unit of the Single Threaded Execution scheme can also be applied to this client code. The Captured Lock unit uses the monitor of the resource object as lock. However, the Shared Lock will use the monitor of an external object to do this guarding against concurrent thread access. Therefore, the Shared Lock unit is more appropriate in cases where more than one shared resources need to be guarded in the critical section.

The previous code sample will now be used with the Shared Lock unit. The scheme and unit selection is shown below (Fig. 7).

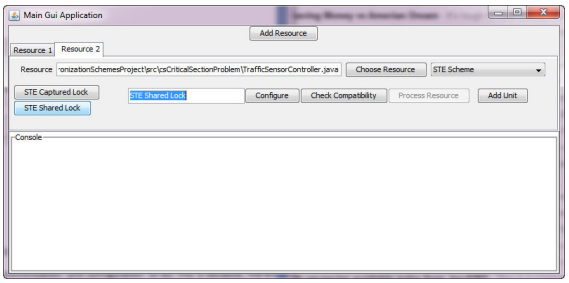


Fig. 7 Applying the second resource with shared lock synchronization unit

In this case, we do not have any synchronization unit configuration to do. This is because, the engine will check for all instance variables marked with the "@SharedField" annotation. Then any methods which make use of these shared fields will automatically be guarded with the Shared Lock.

The annotated client-code is shown below (Fig. 8) after processing:

```
import my.com.checkthread.engine.annotations.ThreadSafe;
import annotations.SharedField;
import annotations.SharedLock;
import annotations.SingleThreadedSchemeSharedLockManaged;
@SingleThreadedSchemeSharedLockManaged
public class TrafficSensorController implements TrafficObserver{
    @SharedField
    private int vehicleCount = 0;
        @SharedLock
        @ThreadSafe
    public void vehiclePassed(){
        vehicleCount++;
    }
        @SharedLock
        @ThreadSafe
    public int getAndClearCount(){
        int count = vehicleCount;
        vehicleCount = 0;
        return count;
    }
}
```

Fig. 8 Generated code after applying the scheme and synchronization units

Open Science Index, Computer and Information Engineering Vol:7, No:6, 2013 publications.waset.org/7402/pdf

B. CScheme for Readers Writers Problem

Our second case study deals with the famous Readers Writers problem which shows a common computing problem in concurrency. It deals with the situation where there are several threads that need to access the same shared resource one at a time, some to perform read operations while others to perform write operations. The main constraint here is that no two threads must access the resource for reading or writing while another thread is performing a write operation on it. However, two or more threads can access the shared resource if their purpose is only for reading.

Consider the code sample below where we have an instance variable "bid". There are two methods, getBid(), to read the bid value and setBid(), to return the bid value. In a multithreaded environment, several threads might access a "bid" object at the same time, some of which might be modifying the state of the object while others might just be reading the object's state. Simultaneous access to an object by both reader and writer threads may result in lost updates as state changes made to an object by writer threads may not be visible to reader threads or other writer threads. Hence, reader threads and writer threads should not be allowed to access the same object at the same time.

The original bid class is shown below (Fig. 9):

```
public class Bid {
    private int bid = 0;
    public int getBid(){
        int bid = this.bid;
        return bid;
    }
    public void setBid(int bid){
        if(bid>this.bid){
            this.bid = bid;
        }
    }
}
```

Fig. 9 The Bid class

The code sample below (Fig. 10) shows the implementation of the Bid class using the *java.util.concurrent.locks.ReadWriteLock*.

```
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
public class BidOriginal {

   private ReadWriteLockrwLock = new
ReentrantReadWriteLock();
   private int bid = 0;
   public int getBid(){
      int bid=0;
      rwLock.readLock().lock();
      try {
         bid = this.bid;
      } finally {                    rwLock.readLock().unlock();
      }
      return bid;
   }
   public void setBid(int bid){
      rwLock.writeLock().lock();
      try {
         if(bid>this.bid){
            this.bid = bid;
         }
      } finally {
      rwLock.writeLock().unlock();
      }
   }
}
```

Fig. 10 The bid class implementation with java's read write lock

Below we show how the Reader Writer Scheme provided by our tool can be applied to the Bid class in order to provide the same reader-writer thread synchronization mechanism as shown in the previous code sample.

First, we mark the instance variables that will be subject to concurrent access by threads in our target class with the "@SharedField" annotation (Fig. 11).

```
import annotations.SharedField;
public class Bid {
   @SharedField
   private int bid = 0;
   public int getBid(){
      int bid = this.bid;
      return bid;
   }
   public void setBid(int bid){
      if(bid>this.bid){
         this.bid = bid;
      }
   }
}
```

Fig. 11 Annotating the shared field of bid class

Then, we select a scheme for the resource class. After that, we choose the synchronization unit. In this case, we choose the Reader Writer Fair unit (equal preference is given to both reader and writer threads thereby preventing starvation) (Fig. 12).

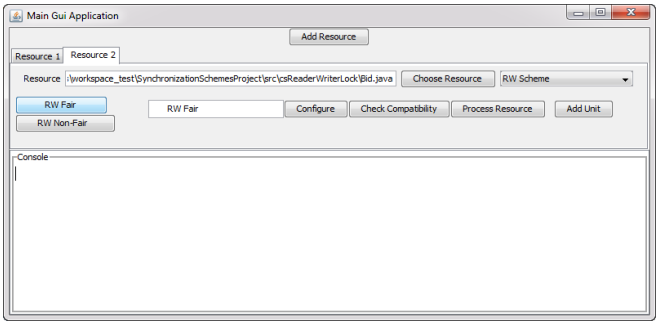


Fig. 12 Applying scheme and synchronization units to Bid class

After the "Process Resource" button is pressed, our target class is processed with the applied synchronization scheme as shown below (Fig. 13):

```
import annotations.ReadOnly;
import annotations.ReaderWriterFairSchemeManaged;
import annotations.SharedField;
import annotations.WriteOnly;
@ReaderWriterFairSchemeManaged
public class Bid {
    @SharedField
    private int bid = 0;
    @ReadOnly
    public int getBid(){
        int bid = this.bid;
        return bid;
    }
    @WriteOnly
    public void setBid(int bid){
        if(bid>this.bid){
            this.bid = bid;
        }
    }
}
```

Fig. 13 Applying scheme and synchronization units to Bid class

This scheme marks the getBid() method as "ReadOnly", that is it will only be accessed by reader threads while the setBid() method is marked as "WriteOnly" implying that it will only be accessed by writer threads. This scheme ensures that "ReadOnly "methods can be simultaneously accessed by reader threads while no writer thread is allowed to enter the "WriteOnly" methods. "WriteOnly" methods can only be accessed by one writer thread at a time while no reader thread is executing a "ReadOnly" method. The Aspects that ensure that a class follows a Fair Reader Writer mechanism are shown below (Figs. 14 and 15):

```
import annotations.ReadOnly;
import annotations.ReaderWriterFairSchemeManaged;
import annotations.WriteOnly;
/**
 * The Class ReaderWriterSchemeFairImplAspect.
 * Implements ReaderWriterReentrantFairAbstractAspect.
 */
public aspect ReaderWriterSchemeFairImplAspect extends
ReaderWriterReentrantFairAbstractAspect {
    //concrete annotation-based pointcuts specification
    public pointcut readerWriterSchemeFairManaged() : execution(*
(@ReaderWriterFairSchemeManaged *).*(..));
    public pointcut readOperation()  : execution(@ReadOnly * *(..))
    &&readerWriterSchemeFairManaged();
    public pointcut writeOperation(): execution(@WriteOnly * *(..))
    &&readerWriterSchemeFairManaged();
}
```

Fig. 14 The aspect code that does the weaving task between a class and a Fair Reader-Writer synchronization unit

```
import java.util.concurrent.locks.ReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock;
/**
 * The Class ReaderWriterReentrantFairAbstractAspect.
 * Abstract Aspect for RW Fair Scheme processing
 */
public abstract aspect ReaderWriterReentrantFairAbstractAspect
perthis(readOperation() || writeOperation()) {
    /**
     * Read operation abstract pointcut.
     */
    public abstract pointcut readOperation();
    /**
     * Write operation abstract pointcut.
     */
    public abstract pointcut writeOperation();

    private ReadWriteLock rwLock = new   ReentrantReadWriteLock(true);
    /**
     * Advice for Read operation pointcut.
     * Binds the pointcut with a read lock of the Fair
ReentrantReadWriteLock.
     */
    Object around() : readOperation() {
        rwLock.readLock().lock();
        try {
            returnproceed();
        } finally {
            rwLock.readLock().unlock();
        }
    }
    /**
     * Advice for Write operation pointcut.
     * Binds the pointcut with a write lock of the Fair
ReentrantReadWriteLock.
     */
    Object around() : writeOperation() {
        rwLock.writeLock().lock();
        try {
            return proceed();
        } finally {
            rwLock.writeLock().unlock();
        }
    }

}
```

Fig. 15 The abstract aspect used in Fig. 14

Open Science Index, Computer and Information Engineering Vol:7, No:6, 2013 publications.waset.org/7402/pdf

Open Science Index, Computer and Information Engineering Vol:7, No:6, 2013 publications.waset.org/7402/pdf

The "perthis" construct is used in order to ensure that a new aspect is created for each object that matches a "readOperation()" or a "writeOperation()"pointcut.

If the "RW Non-Fair" synchronization unit of the Reader Writer synchronization scheme is chosen, the steps to process a resource class with this unit is similar to those of the "RW Fair" unit. However, the target class, after applying the scheme, will look as follows:

```
import annotations.ReadOnly;
impor tannotations.ReaderWriterNonFairSchemeManaged;
import annotations.SharedField;
import annotations.WriteOnly;
@ReaderWriterNonFairSchemeManaged
public class Bid {
    @SharedField
    private int bid = 0;
    @ReadOnly
    public int getBid(){
        int bid = this.bid;
        return bid;
    }
    @WriteOnly
    public void setBid(int bid){
        if(bid>this.bid){
            this.bid = bid;
        }
    }
}
```

Fig. 16 The generated class after applying RW Fair unit

The Aspects that ensure that a class follows a Reader Writer Non-Fair mechanism are shown below (Figs. 17 and 18):

```
import annotations.ReadOnly;
import annotations.ReaderWriterNonFairSchemeManaged;
import annotations.WriteOnly;
/**
 * The Class ReaderWriterSchemeNonFairImplAspect.
 * Implements ReaderWriterReentrantNonFairAbstractAspect.
 */
public aspect ReaderWriterSchemeNonFairImplAspect extends
ReaderWriterReentrantNonFairAbstractAspect {
    //concrete annotation-based pointcuts specification
    public pointcut readerWriterSchemeNonFairManaged() : execution(*
(@ReaderWriterNonFairSchemeManaged *).*(..));
    public pointcut readOperation()  : execution(@ReadOnly * *(..))
    &&readerWriterSchemeNonFairManaged();
    public pointcut writeOperation(): execution(@WriteOnly * *(..))
    &&readerWriterSchemeNonFairManaged();
}
```

Fig. 17 The aspect weaved for Non-Fair Reader Writer synchronization unit

```
import java.util.concurrent.locks.*;
/**
 * The Class ReaderWriterReentrantNonFairAbstractAspect.
 * Abstract Aspect for RW Non-Fair Scheme processing
 */
public abstract aspect ReaderWriterReentrantNonFairAbstractAspect
perthis(readOperation() || writeOperation()) {
    /**
     * Read operation abstract pointcut.
     *
     */
    public abstract pointcut readOperation();
    /**
     * Write operation abstract pointcut.
     */
    public abstract pointcut writeOperation();
    private ReadWriteLock rwLock = new ReentrantReadWriteLock();
    /**
     * Advice for Read operation pointcut.
     * Binds the pointcut with a read lock of the Non-Fair
ReentrantReadWriteLock.
     */
    Object around() : readOperation() {
        rwLock.readLock().lock();
        try {
            return proceed();
        } finally {
            rwLock.readLock().unlock();
        }
    }
    /**
     * Advice for Write operation pointcut.
     * Binds the pointcut with a write lock of the Fair
ReentrantReadWriteLock.
     */
    Object around() : writeOperation() {
        rwLock.writeLock().lock();
        try {
            return proceed();
        } finally {
            rwLock.writeLock().unlock();
        }
    }
}
```

Fig. 18 The abstract aspect of the aspect in Fig. 17

IV. Related Work

The paper Aspects of Synchronization [9] discusses about how Aspect Oriented Programming encourages the separation of the different aspects of a system and how synchronization is a very important aspect in a concurrent object-oriented program. It also describes how synchronization itself has different aspects and by separating the latter, flexible, generic implementations of common synchronization constraints can be derived.

The dissertation referred in [10] presents an approach for constructing concurrent programs which separately handles functional and nonfunctional concerns. The approach defines solutions for each non-functional concern related to concurrency, such as object synchronization, and integrates those solutions as well as their composition in an incremental development process of concurrent programs. In addition, the approach allows the most appropriate solution for each concern to be customized by the programmer to take into account the specific needs of the program being built.

V. Future Work

The next plan is to add more Synchronization Schemes to our proposed paradigm. The two main schemes of interest that we want to add are the Guarded Suspension scheme as well as the Scheduler scheme. The Guarded Suspension scheme is to be used when there exists a condition that prevents a method from doing what it is supposed to do. The scheme will solve this issue by suspending the execution of the method until that condition no longer exists. The aim of the Scheduler scheme is to provide a mechanism to implement a scheduling policy by controlling the order in which threads are scheduled to execute single-threaded code.

We also plan to extend the capability of the bug detection mechanism that we are using in order to be able to find more concurrency related bugs in the client-code. Moreover, we plan to create an eclipse plug-in version for our GUI tool so that so that development will be easier for programmers who utilize the widely used Eclipse IDE.

References

[1] Gordon, E.M., 1998. Cramming more components onto integrated circuits. *Proc. IEEE*, 86: 82-85.

[2] Bryan, C. and B. Jeff, 2008. Real-world concurrency. *Queue Concurrency Prob.,* 6: 17-25..

[3] Ali-Reza, A., K. Christos and S. Bratin, 2006. Unlocking concurrency. *Queue Comput. Archit.,* 4: 25-33.

[4] Kiczales, G., J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.M. Loingtier and J. Irwin, 1997. Aspect-Oriented Programming. In Proc European Conference on Object-Oriented Programming, Finland, June 9-13, 1997, pp.220- 240.

[5] Xerox Corporation, 2002. The aspect J™ programming guide. Palo Alto Research Center, Inc., USA. http://www.eclipse.org/aspectj/doc/released/progguide/index.html

[6] Brian, G., P. Tim, B. Joshua, B. Joseph, H. David and L. Doug, 2006. *Java Concurrency in Practice*. Addison Wesley Professional, India

[7] Joe, C., 2009. Cool Concurrency with CheckThread, eclipse CON, 2009 http://checkthread.org/checkthread_eclipsecon2009.pdf.

[8] Shah, N., Cheerkoot, V, 2011. A Scheme Based Paradigm for Concurrent Programming. Journal of Software Engineering, vol. 5, no. 3, pp. 108-115.

[9] David, H., N. James and P. John, 1997. Aspects of synchronisation. Technol. Object-Oriented Languages Syst., 2: 14-14.

[10] Ant'onio, M.F., 1999. Concurrent object-oriented programming: Separation and composition of concerns using design patterns. Ph.D Thesis, Pattern Languages and Object-Oriented Frameworks.

Open Science Index, Computer and Information Engineering Vol:7, No:6, 2013 publications.waset.org/7402/pdf