World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:6, No:1, 2012

Application of Java-based Pointcuts in Aspect Oriented Programming (AOP) for Data Race Detection

Sadaf Khalid, Fahim Arif

Open Science Index, Computer and Information Engineering Vol:6, No:1, 2012 publications.waset.org/8088/pdf

***Abstract*—**Wide applicability of concurrent programming practices in developing various software applications leads to different concurrency errors amongst which data race is the most important. Java provides greatest support for concurrent programming by introducing various concurrency packages. Aspect oriented programming (AOP) is modern programming paradigm facilitating the runtime interception of events of interest and can be effectively used to handle the concurrency problems. AspectJ being an aspect oriented extension to java facilitates the application of concepts of AOP for data race detection. Volatile variables are usually considered thread safe, but they can become the possible candidates of data races if non-atomic operations are performed concurrently upon them. Various data race detection algorithms have been proposed in the past but this issue of volatility and atomicity is still unaddressed. The aim of this research is to propose some suggestions for incorporating certain conditions for data race detection in java programs at the volatile fields by taking into account support for atomicity in java concurrency packages and making use of pointcuts. Two simple test programs will demonstrate the results of research. The results are verified on two different Java Development Kits (JDKs) for the purpose of comparison.

***Keywords*—**Aspect Bench Compiler (abc), Aspect Oriented Programming (AOP), AspectJ, Aspects, Concurrency packages, Concurrent programming, Cross-cutting Concerns, Data race, Eclipse, Java, Java Development Kits (JDKs), Pointcuts

I. Introduction

CONCURRENT programming errors including data races, deadlocks, starvation and live locks affect the results and performance of concurrent applications. Data race is a concurrent programming error in which more than one threads try to access the same memory location at the same time without acquiring any lock and at least one of the accesses is for writing to memory [1]. Various race detection algorithms have been proposed in the past to address the issue of data race. ERASER the lockset based algorithm is basic contribution to the field of data race detection [2]. RACER algorithm enhances the phenomenon of data race detection in java based programs by making use of AOP [3], [4]. AOP is the modern programming paradigm that facilitates the handling of crosscutting concerns through provision of pointcuts.

Sadaf Khalid is with the Department of Computer Software Engineering, Military College of Signals, National University of Sciences and Technology, Islamabad, Pakistan (e-mail: makkah-madina@hotmail.com).

Fahim Arif is with the Department of Computer Software Engineering, Military College of Signals, National University of Sciences and Technology, Islamabad, Pakistan (e-mail: fahimarif@gmail.com).

Pointcuts and joinpoints help to intercept the events of interest at runtime.Amongst different programming languages, java provides huge support for concurrent programming by introducing various concurrency packages and improving them with every new release of JDK. Java concurrency utilities include java.util.concurrent package containing classes useful for concurrent programming, java.util.concurrent.locks package and java.util.concurrent.atomic package which is actually a small toolkit of classes allowing lock free programming on atomic variables.Use of volatile keyword indicates that application is multithreaded. Volatile variables are usually considered thread safe. But volatile variables become the possible candidates of data races if non-atomic operations are performed concurrently upon them. Volatile variables are considered self-synchronized but the need of synchronization for volatile variables cannot be completely neglected especially if some non-atomic operation is being performed upon the volatile field, since some other thread can intervene and may affect the final results.The aim of this research is to address the issue of volatility and atomicity by making use of java concurrency packages and AOP. This research work proposes some suggestions for incorporating certain conditions for data race detection at the volatile fields in java programs by taking into account support for atomicity provided in java concurrency packages and making use of pointcuts. By considering RACER algorithm as base, an additional state to the RACER state machine is suggested to address atomicity issue for data race detection. It is also described that how the atomicity issue can also be handled by introducing new pointcuts only without modifying the RACER state machine. The results of research are demonstrated by using two test programs. The results are tested on two different JDKs for the purpose of comparison.The rest of the paper is organized as follows. Section II is a brief description of related work, Section III highlights some important concepts of AOP, Section IV demonstrates the proposed research solution, Section V illustrates and analyzes the results of research, Section VI presents a comparison of proposed solution with that of other data race detection techniques and finally paper concludes by suggesting some future work.

II. Related Work

Savage proposed the ERASER algorithm for detecting potential data races in lock-based multithreaded programs. ERASER makes use of lockset algorithm whose refinement at appropriate times leads to the detection of data races. ERASER was used by many researchers as the basis for their

Open Science Index, Computer and Information Engineering Vol:6, No:1, 2012 publications.waset.org/8088/pdf

race detection algorithms, but ERASER had some shortcomings. ERASER could not handle the issue of object initialization properly because it cannot guess when initialization phase is complete and when variable becomes actually shared, thus missing out many data races as well as generating many false alarms. Harrow proposed an extension to ERASER that used thread segments to model lock free handover of objects between parent and child threads [5]. But this technique used the same state machine as proposed by ERASER algorithm, so it also missed out the issue of object initialization leading to false alarms.O'Callahan and Choi proposed another improvement to ERASER algorithm [6]. They refined lockset algorithm implemented by ERASER by using the happens-before graph of the program under test. This happens-before graph contains happens-before edges at calls to Thread.start (), Thread.join (), Object.wait () and Object.notify (). They used some low level bytecode instrumentation toolkit to add instrumentation to the bytecode and did not include any concept of providing instrumentation at higher level of abstraction as possible with AOP paradigm.RACER is the most refined aspect oriented version of ERASER algorithm for detecting data races in java based programs. It refines the lockset algorithm of ERASER by appropriately addressing the object initialization issue. Moreover, RACER also introduced the concept of access periods for lock free handover of object between parent and child threads, but it differs from the scheme proposed by Harrow in the way that it modifies the state machine of ERASER by adding two new states thus handling object initialization by considering reads and writes from the very beginning. The biggest advantage of RACER above all of the previous race detection techniques is that it provides instrumentation at higher level of abstraction by making use of AOP paradigm. Three new pointcuts lock (), unlock () and maybeShared () are introduced as language extension to AspectJ to intercept the events of locking and unlocking and accessing shared memory locations in multithreaded programs.

III. ASPECT ORIENTED PROGRAMMING

AOP is the modern programming paradigm simplifying the implementation of cross-cutting concerns [7]. Concern space is multi-dimensional, so in order to avoid tangling, redundancy and complications caused by scattered functionality, these cross-cutting concerns are modularized in the form of aspects. Aspects are modular implementation of cross-cutting functionality. Aspects can be developed independent of the target application even by some third party reducing the application development time and then woven into the base program through a process called weaving.

Aspect comprises pointcuts, joinpoints and advice which are the fundamental concepts of AOP. Joinpoints are actually the execution points for advice. Pointcuts select certain joinpoints on which certain advice has to be executed, whereas advice is the additional functionality needed to be performed when certain execution point in the program has been reached.

AspectJ is an aspect oriented extension to java [8]. It facilitates the implementation of fundamentals of AOP in java. In AspectJ, aspects are simple class like structures implementing the desired functionality through advice execution on certain execution points selected by pointcuts. Advice in AspectJ is much like a simple function in normal java programming, but here advice is not called explicitly but implicitly invoked by pointcuts and joinpoints.

AOP facilitates application development independent of writing various aspects, so any application can be made aspect oriented by writing suitable aspects for desired situation. The aim of this research is to merge AOP with concurrency support provided by java in order to handle the issue of data race detection effectively.

IV. PROPOSED ENHANCEMENTS FOR DATA RACE DETECTION

Multithreaded programs may exhibit non-deterministic behavior giving rise to data races. A lot of work has been done in the past to cope with the concurrent programming errors especially with the situations giving rise to data race. Data races can be minimized by strict adherence to certain programming disciplines. Use of synchronized blocks reduces memory consistency errors. Volatile variables are often used in multithreaded programs for data sharing among threads. Though considered self-synchronized, they cannot be completely neglected for being the candidate of data races. Volatile variables become the possible candidates of data races if non-atomic operations are performed concurrently upon them. Java provides support for concurrency by introducing concurrency package containing atomic classes. Use of these atomic classes can reduce the chances of data race. The authors aim to use RACER algorithm as base and propose some suggestions for data race detection at volatile fields in java programs by availing the benefits of java concurrency packages for atomicity and by using AOP.

A. Observations

Though considered to be self synchronized, volatile fields are not completely thread safe. If some non-atomic operation like increment (i++) operation is being performed on certain volatile field, then there is greater chance of thread interference. Because the increment operation comprises of get, increase and set sub-operations, so there is a chance that another thread might get the incorrect result if volatile field is concurrently accessed by more than one threads. Similar is the case with other non-atomic operations. Continuous testing revealed that besides unary operations, expressions making use of conditional operators also become non-atomic. So, in this case volatile field become the possible candidate of data race. This research work proposes suggestions to handle the race conditions arising from these kinds of situations in java based programs.

B. Proposed Solution for Data Race detection at Volatile Fields Undergoing Non-atomic Operations

Three different ways to detect and avoid data races at volatile fields undergoing non-atomic operations proposed in this research work include:

i. Use of synchronization block for accessing volatile fields.
ii. Use of atomic classes introduced by java concurrency packages to accommodate atomicity issue.

Open Science Index, Computer and Information Engineering Vol:6, No:1, 2012 publications.waset.org/8088/pdf

iii. Suggesting a new state machine for handling atomicity.

1) Use of Synchronization Block for Accessing Volatile Fields

Use of synchronized block guarantees safe operation with no chance of data race. It is one of the ways to perform atomic operation without using any concurrency package for atomicity, since there is no chance of thread interference. Use of synchronized block for accessing volatile fields is shown in Fig. 1. In this case, volatile variable *counter* is being accessed for performing two non-atomic operations, one the increment operation at line 13 and another conditional expression at line 14. Performing these non-atomic operations on volatile fields within synchronized block eliminate the chance of data race.

```
1  public class TaskForVolatile implements Runnable
2  {
3      int test=0;
4      private volatile int counter = 0;
5      public void run () {
6                  while(test < 5) {
7                      try {
8                  Thread.sleep(100);
9              } catch (Exception e) {
10                 // TODO: handle exception
11             }
12             synchronized(TaskForVolatile.class){
13                 System.out.println("counter:" + counter++);
14                 if(counter >= 5) {
15                 stopMe();
16                 counter=0;
17                 }
18             }
19         }
20                 }
21     private void stopMe() {
22             test++;
23     }
24     public static void main(String [] args) {
25     TaskForVolatile t1 = new TaskForVolatile();
26         for (int i = 0; i < 3; i++) {
27             Thread thread1 = new Thread(t1);
28             thread1.start ();
29         }
30     }
31 }
```

Fig. 1 Use of Synchronized Block for Accessing Volatile Fields

We introduced a new aspect *VolatileChk* keeping track of accesses to volatile fields by making use of two additional pointcuts *volatileFieldSet()* and *VolatileFieldGet()* as shown in Fig. 2. Without a synchronized block, data race will be fired for volatile fields undergoing non-atomic operations.

```
public aspect VolatileChk
{
pointcut volatileFieldSet(): set (volatile * *) && set(!Atomic* *) &&maybeShared();
pointcut volatileFieldGet(): get (volatile * *) && get(!Atomic* *) &&maybeShared();
}
```

Fig. 2 Aspect Introducing Pointcuts for Detecting Reads and Writes to Volatile Fields

Maybe, shared () pointcut developed by E. Bodden and K. Havelund is also used. Results of this test program will be discussed in Section V.

2) Use of Atomic Classes Introduced by Java Concurrency Packages to Accommodate Atomicity Issue

With evolution of JDKs, java introduced concurrency package java.util.concurrent.atomic for safe handling of atomic operations avoiding data races. A simple increment operation is non-atomic, if performed on volatile fields without using synchronization mechanism, there is a great possibility of data race in case of concurrent access of volatile field by more than one threads leading to incorrect results. But if atomic classes introduced by java concurrency package are used then volatile fields can undergo simple non-atomic increment operation without making use of synchronized blocks. A test program in Fig. 3 shows the usage of atomic classes for performing non-atomic operations atomically. We only make use of AtomicInteger class for the purpose of demonstration in this test program.

```
1  import java.util.concurrent.atomic.*;
2  public class AtomicTestClass implements Runnable
3  {
4      private volatile  AtomicInteger counter= new AtomicInteger
5      int test = 0;
6      int temp;
7      public void run () {
8          while(test < 5) {
9              try {
10                 Thread.sleep(100);
11             } catch (Exception e) {
12                 // TODO: handle exception
13             }
14             System.out.println("counter =" + counter);
15             counter.incrementAndGet();
16             temp= counter.get();
17
18             if(temp >= 5) {
19                 stopMe();
20                 counter = new AtomicInteger(0);
21             }
22         }
23     }
24     private void stopMe() {
25             test++;
26 }
27     public static void main(String [] args) {
28         AtomicTestClass t1 = new AtomicTestClass();
29         for (int i = 0; i < 5; i++) {
30             Thread thread1 = new Thread(t1);
31             thread1.start ();
32             }}}
```

Fig. 3 Test Class Showing the Usage of Atomic Test Class Introduced by java.util.concurrent.atomic Concurrency Package

Fig. 4 shows a new aspect *AtomicityChk* containing two pointcuts *AtomicityCheck_Read ()* and *AtomicityCheck_Write ()*. These new pointcuts are created to detect the accesses to the volatile fields making use of atomic test classes. If such a field is accessed outside synchronization block, no data race will be fired since atomicity is guaranteed by the usage of atomic concurrency package. Fig. 4 also shows a new pointcut *neverShared ()*. Its purpose is to create an impact that fields can never become shared if atomic operations are performed. Section V explains the results of the test program in detail.

```
public aspect AtomicityChk
{
    pointcut atomicityCheck_Read(): get(volatile Atomic* *) && neverShared();
    pointcut atomicityCheck_Write(): set(volatile Atomic* *) && neverShared();
    pointcut neverShared(): if(false);

}
```

Fig. 4 Aspect Introducing Pointcuts for Detecting Reads and Writes to Volatile Fields Making Use of Atomic Test Classes

3) Suggesting a New State Machine for Handling Atomicity

We suggest a new state machine with two states *Virgin* and *Atomic* for handling atomic operations. In this case, the

Open Science Index, Computer and Information Engineering Vol:6, No:1, 2012 publications.waset.org/8088/pdf

immediate state of the variable just after its creation is Virgin. If the newly created field is undergoing atomic operations by making use of java atomic classes, then once the field is read or written by any thread, its state is transitioned to A*tomic* and then no data race will be fired for such volatile variables undergoing atomic operations. Rest of the transitions and working will be same as that of RACER state machine [3]. Fig. 5 shows the proposed state machine for handling atomic operations.

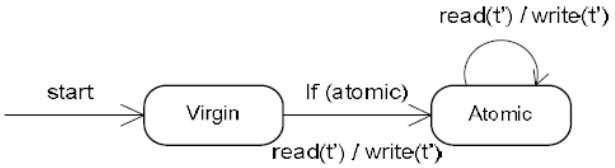


Fig. 5 Proposed State Machine for Handling Atomicity Issue

V. Analysis

Test results are obtained using AspectJ in Eclipse and providing runtime environment of abc. In this section, results obtained using JavaSE 1.6 as JRE System Library (SUN JDK) and ibm_sdk60 as JRE System Library (IBM JDK) are described in detail.

A. Testing Proposed Solution with SUN JDK

Volatile variables are used to share the values among threads in multithreaded programs. They are usually not cached in thread local memory and contain the most recent value updated by any thread. Volatile variables are usually considered thread safe and are accessed without strict adherence to locking discipline. However, this is not the case with all of the accesses to volatile fields. In the test program of Fig. 1, volatile variable *counter* is undergoing two different kinds of non-atomic operations. In line 13, value of *counter* is incremented whereas in line 14, *counter* is undergoing another non-atomic operation in conditional expression. We classify this operation within the conditional expression as non-atomic because through continuous testing, it was observed that new value of *counter* is misread by some threads because it has already been changed during the condition testing, which is not detected. But if this operation is performed atomically within the synchronized block then such kind of errors can be avoided. The increment operation is already non-atomic since there is the chance of thread interference, so it is better to perform such operations atomically within the synchronized block. Executing this test program using AspectJ and providing runtime environment of abc containing newly created pointcuts yields the output shown in Fig. 6. Here, no data race is reported at the volatile field *counter*, since it is accessed within the synchronized block. We enhanced RACER by making it sensitive to such volatile field accesses via pointcuts introduced in newly created aspect shown in Fig. 2. However, the data race is being reported at another instance variable *test* which is quite true because this variable is actually shared among the threads.

```
==========================
Race condition found!
Field 'int TaskForVolatile.test' is accessed unprotected.
Owner object: 18751079
==========================
```

Fig. 6 Output with NO Data Race at Volatile Field Accessed within the Synchronized Block

If we execute the test program in Fig. 1 without using synchronized block, output shown in Fig. 7 is produced. Here, newly created pointcuts detect the unprotected access to volatile fields undergoing non-atomic operations to fire data race on it.

```
==========================
Race condition found!
Field 'private volatile int TaskForVolatile.counter' is accessed unprotected.
Owner object: 27196165
==========================

==========================
Race condition found!
Field 'int TaskForVolatile.test' is accessed unprotected.
Owner object: 18751079
==========================
```

Fig. 7 Output Showing Data Race at the Volatile Field undergoing Non-atomic Operations

With the evolution of java programming language, new concurrency package *java.util.concurrent.atomic* supporting atomicity is introduced. Atomic operations are safe and cannot become victims of thread interference. So, there are no chances of data race in case of atomic operations. So, if support provided by java.util.concurrent.atomic package is availed, then chances of data race are eliminated and volatile variables can be accessed without any synchronization mechanism. In order to adjust support for atomicity in existing race detection algorithm RACER, we introduced a new aspect shown in Fig. 4. Two pointcuts *atomicityCheck_Read ()* and *atomicityCheck_Write ()* detect if any volatile field making use of atomic classes introduced by java.util.concurrent.atomic is used. If such classes are used, then the purpose of *neverShared()* pointcut is to implement the assumption that such fields can never be shared among the threads, so there is no chance of data race at volatile fields making use of atomic classes. Result of executing the test program of Fig. 3 is shown in Fig. 8.

```
==========================
Race condition found!
Field 'int AtomicTestClass.temp' is accessed unprotected
Owner object: 28606871
==========================

==========================
Race condition found!
Field 'int AtomicTestClass.test' is accessed unprotected
Owner object: 28606871
==========================
```

Fig. 8 Output with NO Data Race at Volatile Field using Java Atomic Classes (Detected via Pointcut neverShared ())

Open Science Index, Computer and Information Engineering Vol:6, No:1, 2012 publications.waset.org/8088/pdf

In test program shown in Fig. 3, our main concern is with the usage of atomic classes for creating atomic variables. Volatile variable *counter* is created as an instance of AtomicInteger class introduced by concurrency package java.util.concurrent.atomic. Since use of atomic classes guarantee safe operations without thread interference, so here no data race is being reported at the volatile field *counter* even it is not accessed within the synchronized block. This is because newly developed pointcut *neverShared ()* made the assumption that fields on which atomic operations are being performed can never become the shared fields. The other two data races being reported as shown in Fig. 8 are true one, as the instance fields *temp* and *test* are actually shared among threads.If the pointcut *neverShared ()* is not created, then in order to deal with atomic operations, we suggest a new state machine shown in Fig. 5. After a new variable is created, its state is declared as *Virgin*, after checking that whether it is created by making use of java atomic classes, whenever a thread reads or write this variable, its state is switched to *Atomic*. No data race is fired in *Atomic* state because this state is introduced as an indication of atomic operations. Output of the same test program shown in Fig. 3 when tested with the implementation of new state machine is shown in Fig. 9. This state machine only handles the atomic operations on volatile fields; the rest of the implementation we used is same as that of RACER state machine.

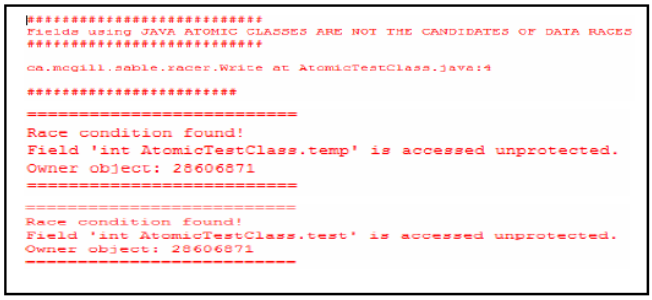


Fig. 9 Output showing NO Data Race at the Volatile Field using Java Atomic Classes (Detected via Proposed State Machine)

Output in Fig. 9 shows that when a volatile field *counter* declared using java atomic classes is accessed at line 4 for writing purpose by thread, its state is transitioned to *Atomic* and a message saying 'Fields using JAVA ATOMIC CLASSES ARE NOT THE CANDIDATES OF DATA RACES' is displayed. After this, whichever thread accesses this field *counter* for either read or write purpose, its state will remain the same and no data race will be reported for such volatile fields on which atomic operation is being performed.

B. Testing Proposed Solution with IBM JDK

IBM JDK provides utility classes containing support for concurrency same as that in SUN JDK. We propose some suggestions to handle volatility and atomicity issue for data race detection and tested the results on smaller test programs. So no change in results is obtained. We obtain the same results for both SUN and IBM JDKs. Differences in thread management in SUN and IBM JDK might be observed for some larger programs.

VI. Comparison Of Data Race Detection Techniques

Data race is a serious programming error in multithreaded programs. A lot of work has been done on fixing various concurrent programming errors especially data races in concurrent programs. Various race detection algorithms have been proposed in past to address the issue of data race detection in different programming languages. ERASER is the lock-set based race detection algorithm for C and C++ programs. Harrow proposed an extension to ERASER by introducing the concept of thread segments implemented in Visual Threads. Mayur Hiru Naik presented his algorithm for static race detection in java programs [9], [10]. RACER on the other hand is the most recent aspect oriented race detection algorithm for java based programs. All of these algorithms address the race detection issue in their own style. In this research work, we propose suggestions to address a unique issue of data race detection at the volatile fields, and how concurrency utilities for atomicity provided by evolution of java programming language can be utilized to handle the issue effectively. This research work demonstrates how the application of common programming practice normally used for writing concurrent programs can help address the race detection issue at the volatile fields. Volatile variables being considered as self-synchronized have never become the focus of the data race issue. But through careful usage of volatile variables and taking into account the atomicity issue, another aspect of data race detection issue can be resolved. Table I highlights the comparison among different race detection techniques. None of the existing race detection techniques particularly focused on the volatility and atomicity issue, so we propose suggestions to address this issue for detecting data races in multithreaded programs.

TABLE I
Comparison Between Different Data Race Detection Techniques

Data Race Detection Technique	Object Initialization	Aspect Oriented	Accesses to Volatile Variables	Atomicity Issue
ERASER	N	N	N	N
RACER	Y	Y	N	N
Static Race Detection Tech. by Mayur	N	N	N	N
Visual Threads by Harrow	N	N	N	N

We have presented the results of two different test programs making use of static, volatile and atomic variables. Fig. 10 summarizes the results by highlighting the total number of data races in test program of Fig.1 while accessing the volatile field outside the synchronized block. The total numbers of data races existing in the test program of Fig. 1 are two. Data race at the simple instance field *test* is detected via RACER logic while data race at the volatile field is being reported through the addition of pointcuts *volatileFieldSet()* and *volatileFieldGet().*

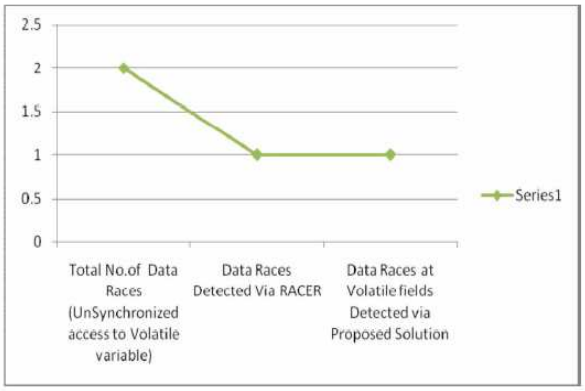


Fig. 10 Data Race Detection

Test program in Fig. 3 contains two data races at the instance fields *test* and *temp*, while the data race at the volatile field *counter* is avoided by using java atomic classes. Our newly created pointcuts *atomicityCheck_Read*, *atomicityCheck_Write()* and *neverShared()* help to resolve the issue. *atomicityCheck_Read* and *atomicityCheck_Write()* detect the accesses to atomic volatile fields and *neverShared()* pointcut checks that these fields are never shared among threads and can never become the possible candidates of data races, thus avoiding false alarms.

VII. Conclusion

AOP is modern programming paradigm facilitating the implementation of cross-cutting concerns in modular style. Its simplicity, ease of use and ability to intercept events of interest at runtime increases its usage for solving various issues. For the same reason, it can also be used for solving concurrent programming errors like data races, starvation etc. In this research work, we have used AspectJ to address data race issue by combining the benefits of AOP and java concurrency utilities. We have presented the race detection issue at the volatile fields and atomicity concept for java programming language, but this concept can be extended to other programming languages by considering the concurrency utilities and aspect-oriented support provided by them. By using AspectC, Aspect# etc, concept of data race detection by using aspects and pointcuts can be extended to C and C# programming languages. Moreover, we can devise aspect oriented algorithms for addressing other concurrent programming errors like starvation, live locks etc.

Acknowledgment

We thank Eric Bodden for providing us with the implementation of RACER algorithm which helped us understanding the algorithm and facilitate implementing our proposed modifications in the algorithm. We also thank National University of Sciences and Technology, Islamabad, Pakistan for facilitating us in carrying out this research work and for provision of equipment required for implementation and testing of our results.

References

[1] "Oracle Solaris Studio 12.2 Thread Analyzer User's Guide," Internet: download.oracle.com/docs/cd/E18659_01/pdf/821-2124.pdf [6 Sep. 2010].

[2] S.Savage, M.Burrows, G.Nelson, P.Sobalvarro and T.Anderson, "*Eraser: A Dynamic Data Race Detector for Multithreaded Programs*," ACM Trans. Computer Systems, Vol.15, No.4, pp. 391-411, 1997.

[3] Eric Bodden and Klaus Havelund, "*Aspect-Oriented Race Detection in Java*", IEEE Trans. on Software Engineering, Vol.36, N0.4, July/August 2010.

[4] Eric Bodden and Klaus Havelund, "*Racer: Effective Race Detection Using AspectJ*", Proc. Int'l Symp. Software Testing and Analysis, pp. 155-165, July 2008.

[5] J.Harrow, "*Runtime Checking of Multithreaded Application with Visual Threads*", SPIN Model Checking and Software Verification, Springer, pp. 331-342, 2000.

[6] R. O'Callahan and J-D. Choi, "*Hybrid Dynamic Data Race Detection*", Proc. ACM SIGPLAN Symp. Principles and Practice of Parallel Programming, pp. 167-178, 2003.

[7] Pouria Shaker and Dennis K. Peters, "*An Introduction to Aspect-Oriented Software Development*", Proc. Newfoundland Electrical and Computer Engineering Conference, October 2005.

[8] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold, "*An Overview of AspectJ*", ECOOP'01 Proc. of 15th European Conference on Object Oriented Programming, 2001.

[9] Mayur Hiru Naik, "Effective Static Race Detection for Java", Ph.D Dissertation, March 2008.

[10] Mayur Naik and Alex Aiken, "*Conditional Must Not Aliasing for Static Race Detection*", Proc. of the 34th annual ACM SIGPLAN, 2007.

Open Science Index, Computer and Information Engineering Vol:6, No:1, 2012 publications.waset.org/8088/pdf