Yet, Another Method for Detecting API Deadlock

Suvarin Ploysri and Wanchai Rivepiboon
Department of Computer Engineering, Chulalongkorn University, Bangkok, Thailand

Keywords: Static Analysis, Deadlock Detection, Multithreading, Java API.

Abstract: Currently, developing a multithreading Application Programming Interface (API) for special use is extensive. Defects that are the most concern are deadlocks since the deadlock causes the application developed on top of the API stops working. The deadlock detection algorithm has been developed widely. We present another deadlock detection algorithm using the concept of static analysis. The algorithm detects potential deadlocks in the source code of the multithreading API. Finally, it reports deadlock sites. The result assists the developer to be aware of code synchronization that potentially encounters deadlocks. Moreover, it is related information to find the root cause of the deadlock in the future.

1 INTRODUCTION

Deadlock is a general problem for concurrence programming. Developing the API that has multi-threads also confronts with the deadlock problem. Lacking of understanding of developers in the architecture of the multithreading API for development and testing coverage causes the deadlock defects. If defects are not fixed and the API is used to develop the application, the application will encounter the deadlock later. Amy Williams, William Thies, and Michael D. Ernst (2005) quoted that finding and fixing deadlock was difficult. If the deadlock occurs in the application, it is possible that fixing deadlock in the API layer may affect to the behaviour of the API. Deadlock prevention is the first solution; however, it is still not sufficient and inevitable that we can avoid this problem. Deadlock detection is also another alternative to detect and later solve deadlocks in early of the Software Development phase of the API. After we know the deadlock site and even though we cannot fix the deadlock because it leads to the design of the API changed, knowing it early can help us to manage adding on the limitation of the API usage in the document to caution developers about defect parts.

Using static analysis for detecting the deadlock in the multithreading API is more suitable than using other analysis; dynamic analysis or hybrid analysis, because we still don't know the exactly scenario of the customer's application implementation when using the multithreading API and we should consider all potential deadlock in the API source code.

For the deadlock detection algorithm, we use six necessary conditions for the deadlock proposed by Mayur Naik, Chang-Seo Park, Koushik Sen and David Gay (2009) and two code patterns presented by Frank Otto and Thomas Moschny (2008) as conditions for developing the deadlock detection algorithm. We also have to consider synchronization objects, synchronization statements and wait-notify methods in the source code of the multithreading API.

Finally, our algorithm can detect the potential deadlock in the multithreading API. The reported result of the deadlock detection algorithm matches with the reported deadlock that was investigated by the support consultant of the multithreading API.

The remainder of this paper is organized as follows. In section 2, we provide information about background knowledge for more understanding in the term of the deadlock in the multithreading API. In section 3, we discuss related works for the deadlock detection algorithm. In the section 4, we present our deadlock detection algorithm. In section 5, we provide the result from the deadlock detection algorithm. In section 6, it is the limitation of the deadlock detection algorithm. And finally, in the section 7, it is the conclusion and future work.

Ploysri S. and Rivepiboon W..
Yet, Another Method for Detecting API Deadlock.
DOI: 10.5220/0004417801320139
In *Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2013), pages 132-139
ISBN: 978-989-8565-62-4

2 BACKGROUND KNOWLEDGE

2.1 Deadlock in Multithreading API

Before we explain about the definition of the deadlock in the multithreading API, let see the definition of the deadlock and the multithreading API as follows.

2.1.1 Deadlock

The deadlock is a general term in several fields. Pallavi Joshi, Chang-Seo Park, Koushik Sen and Mayur Naik (2009) give the definition that a deadlock is a liveness failure that happens when a set of threads blocks forever because each thread in the set is waiting to acquire a lock held by another thread in the set. Qichang Chen, Liqiang Wang, Ping Guo and He Huang (2011) completely provides the definition that a deadlock occurs when a chain of processes/threads are involved in a cycle in which each process is waiting for resources/locks that are held by some other processes. When a deadlock happens, none of the processes/threads can proceed, which in turn causes the whole or part of the program to halt. In summary, the deadlock occurs when the resources are blocked by more than two threads and causes the program stops working.

The general root cause of the deadlock problem is incorrect synchronization of a pair or more objects (Holt, 1972), incorrect ordering of lock acquisitions said by Tong Li, Carla S. Ellis, Alvin R. Lebeck and Daniel J. Sorin (2005) or incorrect implementation of the API or the application.

2.1.2 Multithreading API

The multithreading API is implemented on top of the general API such as Java for specific use. The multithreading API is an API designed to have several threads to process their tasks. There are some threads communicate with each others to process data. Threads will be created when the application

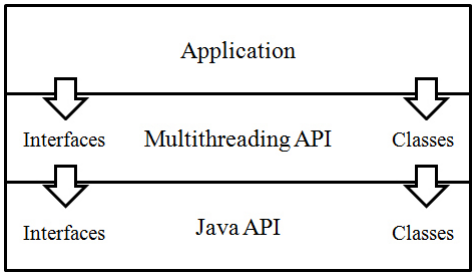

Figure 1: The application layer, the multithreading API layer and the Java API layer.

multithreading API is an API designed to have several threads to process their tasks. There are some threads communicate with each others to process data. Threads will be created when the application calls into the API layer at runtime. Figure 1 shows the building block of the application including underlying layers that are the multithreading API layer and the Java API layer.

2.1.3 Deadlock in Multithreading API

The deadlock occurs in the multithreading API when the API creates several threads calling by the application layer at runtime. There are some threads locks and waits for the same objects. Figure 2 shows the deadlock occurs in the multithreading API.

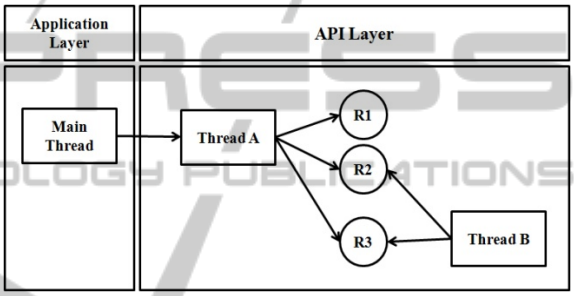

Figure 2: The deadlock occurs in the multithreading API.

Figure 2 demonstrates that the main thread of the application layer calls to the multithreading API layer and it creates 2 threads that are Thread A and Thread B. R1, R2 and R3 are represented as resources used by threads. Thread A locked for R1 and R2 and Thread B locked for R2 and R3. In runtime, there will be only one thread that locks for R2 and R3 at a time. Therefore if Thread A locks R2 and Thread B locks R3 and then Thread A requires to lock R3 and Thread B requires to lock R2, the deadlock will occur. The application will stop working because each thread cannot continue its task.

2.2 Static Analysis

Static analysis makes predictions about a program's runtime behaviour based on analyzing its source code (Chen et al., 2011). Static analysis focuses on the structure of the program and does not require its execution. All possible (but also infeasible) execution paths can be taken into consideration (Otto et al., 2008). Static analysis is more suitable for detecting deadlock in the multithreading API because we still don't know the exact scenario that will be used to develop the application.

The root cause of the deadlock problem is incorrect synchronization of pair or more objects (Holt, 1972), incorrect ordering of lock acquisitions (Li et al., 2005) or incorrect implementation of API or application. Using static analysis can help us to detect all synchronized statements, synchronized methods and wait-notify methods in the source code of the multithreading API.

3 RELATED WORKS

There are algorithms developed on the concept of static analysis such as the work of Naik et al. (2008). They used 0-CFA call graph, flow-insensitive k-object-sensitive analysis, thread-escape analysis and may-happen-in-parallel analysis to get the result of the deadlock site. Otto and Moschny (2008) used the point-to and may-happen-in-parallel analysis. Agarwal and Stoller (2006) proposed operations to detect the deadlock that are acquired and released of locks, wait-notify on condition variables, up and down operations on semaphores, accesses to shared variables and thread start and join and termination operations. Williams et al. (2005) used flows-sensitive and context-sensitive analysis for static deadlock detection in Java libraries and lock-order graphs to represent locking. They focused on deadlocks occurred by synchronized statements and the wait-notify methods of Java. Jyotirmoy Deshmukh, E. Allen Emerson and Sriram Sankaranarayanan (2009) used symbolic deadlock analysis. They used lock-order graph analysis, logical formulae for symbolic enumeration of alias patterns, Soot framework, may-aliases for tracking locked objects across methods. They detected deadlocks in Java, concurrent libraries and multithreaded client applications. Gianpiero Francesca, Antonella Santone, Gigliola Vaglini and Maria Luisa Villani (2011) used the Calculus of Communicating System (CCS) to detect deadlocks. CCS is temporal-logic formula representing the requirement verified by the concept of Model Checking for complex system and proving correctness of a system. Engler and Ashcraft (2003) developed a static tool named RacerX using flow-sensitive, interprocedural analysis to detect both race conditions and deadlocks in the code. Mikhail Moiseev, Alexey Zakharov, Ilya Klotchkov and Sergey Salishev (2011) presented an approach of deadlock detection in the SystemC design based on static code analysis and implemented in the Deadlock Analyzer tool.

4 DEADLOCK DETECTION ALGORITHM

For the deadlock detection algorithm, we use six conditions of the deadlock (Naik et al., 2009) and a code pattern (Otto et al., 2008). We adapt them to implement our own algorithm to detect the potential deadlock in the source code of the multithreading API. Six conditions of the deadlock (Naik et al., 2009) consider locked objects whether they can be accessed by several threads or not. A code pattern (Otto et al., 2008) provides information about the Cyclic Lock Dependency that if the lock order of two fragment of code is reverse order, it shows the cyclic and causes a deadlock. In summary, the deadlock detection algorithm considers synchronized statements, synchronized methods and wait-notify methods in the source code of the multithreading API. Then the algorithm will find the relation of locked objects, methods and threads from all these statements, methods and classes in the source code to get information for locked objects in each thread. Using six conditions of deadlock (Naik et al., 2009) the algorithm will verify whether there are more than 2 threads lock the same object or not. Using the Cyclic Lock Dependency (Otto et al., 2008) to verify whether there are more than 2 threads locks the same object in reverse lock order. If it is in this case, it is possible that the deadlock occurs. For easier to implement the deadlock detection algorithm, we will split the deadlock detection algorithm up to 3 parts that are 4.1 Source Code Information that gets information of object names, method names and thread names in the source code of the multithreading API, 4.2 Source Code Analyzing that analyzes the relation of objects, methods and threads and which threads call methods and lock which objects. And the last part is 4.3 Deadlock Detection that detects the deadlock using information from prior parts.

4.1 Source Code Information

Firstly, the deadlock detection algorithm should have source code information of the multithreading API. We will use java.lang.reflect package of the Java API to get Meta data of the class, synchronized methods, synchronized objects, fields of class, wait-notify methods, Thread names and names of method callers and callees. The algorithm collects information into the container for providing relation and detecting the deadlock in next steps.

The following part of example code is used to demonstrate our algorithm for getting source code

information.

```
   public class CodeDummy extends
SuperDummy implements InterfaceDummy{
    CodeDummy _cd = new CodeDummy();
    CodeDummy _cd2 = new CodeDummy();
    ...

    public CodeDummy(){
        this.push();
        synchronized(this){
            //...
        }
        doSomething(_cd);
        synchronized(_cd2){
            //...
   }
    }

    synchronized void push(){
        //...
    }
    String doSomething(CodeDummy cd){
        _cd2.wait();
        push();
        //...
   }
   }
```

The algorithm will collect following information of the class into the container as initial information.

4.1.1 Class

The class name, superclass extending and the interface implementation are information to get exact locked objects that are used in the synchronized statements, synchronized methods and wait-notify methods of a class of the source code of the multithreading API. Sometimes the developer uses 'this' or 'super' or instant objects that are not relevant the exact locked object. We have to collect this information to uncover the objects that are in the synchronized statements and synchronized methods and call wait-notify methods. The above example code shows the CodeDummy class extends from the SuperDummy class and implements the InterfaceDummy interface. The locked object can be the CodeDummy object or the SuperDummy object. When it calls this.push();, 'this' is the CodeDummy object that is locked.

4.1.2 Synchronized Methods

We will get the information which methods are synchronized in a class of the source code of the multithreading API to get the locked object from the class name or the extended class. From above example code, it shows that the push() method is synchronized. Therefore we will collect the push() method. In runtime, when the API calls this point the CodeDummy object will be locked.

4.1.3 Synchronized Objects and Fields of Class

The synchronized objects or synchronized statements are blocks of object synchronization. Normally, the object variable doesn't relevant the information of the object type locked therefore we have to collect all variable declaration of the class. In above example code, when calling synchronized(this), 'this' refers to the CodeDummy object. There is one more example on synchronized(_cd2); line, we will not know the type of _cd2 at this point. In addition, there is another example of code that calls doSomthing(_cd);, we do not know the exact type of _cd as well. Therefore we have to collect all variable declaration of the class to define the type of the object later. If there are more than a method calls the same synchronized object, it is possible that the deadlock occurs on this object.

4.1.4 Wait-notify Methods

Normally, calling the wait-notify method have to call via the object. In above example code, _cd2 calls the wait() method. We will collect information that _cd2 calls the wait() method and which method calls the wait() methods. From above example code, the doSomething() method calls _cd2.wait(). For the notify() method, the algorithm has to collect this information in the same way of the wait() method.

4.1.5 Thread

We have to check which class extends the Thread class and implements the Runnable interface to find all threads in the multithreading API.

4.1.6 Method Caller and Callee

To know exact method calling of the multithreading API, we also have to collect which methods are callers and callees.

After we know what information we have to collect from the source code, then we will implement the algorithm to get all information from the source code of the multithreading API.

The following is the source code to get the package name and the class name of several java files of the source code of the multithreading API. We will add package names and class names to the containers to get information from several java files.

```
   scanner = new Scanner(fr[i]);
   while(scanner.hasNext()){
    temp1 = scanner.next();
    if(temp1.equals("package"))
    temp2 = scanner.nextLine();
    temp3 = new String(temp2.trim());
        temp3 =
temp3.substring(0,temp3.length()-1);
        packageName.add(temp3);
        scanner.nextLine();
   }
   if(temp1.equals("class")){
    temp1 = scanner.next();
    if(temp1.contains(keyword) &
!temp1.contains("{")){
        scanner.next();
        continue;
    }
    else if (!temp1.contains(keyword) &
!temp1.contains("{")){
        classname = new String(temp1);
        className.add(classname);
        break;
    }
   }
```

After we get package names and class names, we will use java.lang.reflection to get exact information of the each class that are superclass name, its methods and fields. The code is shown as follows.

```
   myClass =
Class.forName(packageName.get(j)+"."+
className.get(j));
   System.out.println(myClass.getName()
);

   for(Type type :
myClass.getGenericInterfaces()){
   System.out.println(type.toString());
   }
   Method [] methods =
myClass.getDeclaredMethods();
   for(Method methodname : methods){
   System.out.println(methodname.toStri
ng());
   }
   Field [] fields =
myClass.getFields();
   for(Field fieldname : fields){
   System.out.println(fieldname.toStrin
g());
   }
```

The following source code is the algorithm to get the inner class of the source code of the multithreading API.

```
   Class [] declaredClasses =
myClass.getDeclaredClasses();
   for(Class itsClass :
declaredClasses){
    System.out.println(itsClass.getCano
nicalName());
    for(Type type :
itsClass.getGenericInterfaces()){
        System.out.println
(type.toString());
    }
    for(Method itsClassmethod :
itsClass.getDeclaredMethods()){
        System.out.println
(itsClassmethod.toString());
    }
   }
```

Source code information will be collected with the form of the table for each class. Table 1 is the synchronized object table to collect object names and types of objects that are synchronized by statements or methods or called by wait-notify methods. Table 2 is the synchronized method table. It will collect synchronized method names and the object names that are synchronized. Table 3 will collect the class names that extend the Thread class or implement the Runnable interface and their fields. Table 4 collects class names and their methods. Table 5 collects method callers and callees which are called by the same thread or not.

The following are tables that will be used for analyzing the source code.

Table 1: The synchronized object table.

Synchronized object types	Synchronized object names
CodeDummy	_cd
CodeDummy	_cd2
CodeDummy	this
...	...

Table 2: The synchronized method table.

Synchronized method names	Synchronized method objects
push	CodeDummy
doSomething	_cd
wait	_cd2
...	...

Table 3: The class name extending Thread or implementing Runnable and their fields.

class names	fields
EventQueue	LinkedList _queue, Runnable _event
Table 3: The class name extending Thread or implementing Runnable and their fields. (Cont.)2	
CodeDBFactory	CodeDBFactory facDb, Hashtable manager
...	...

Table 4: The class names and their methods.

class names	methods
CodeDummy	push, doSomething
...	...

Table 5: The method callers and callees.

method callers	method callees
doSomething	push
...	...

4.2 Source Code Analyzing

After the algorithm gets all information of the source code, it will find the relation of information in each table which threads calls which methods and which methods lock which objects. And the algorithm will rearrange information into the following structure shown in Figure 3.

```
Thread.name,
Methodname, locked objectnameA, line# of method,
Synchronized objectnameB, line# of synchronized objectnameB,
objectC.notify, line# of objectC,
objectD.wait, line# of objectD,
...
```

Figure 3: Information restructuring.

The structure in Figure 3 is used for representing the name of thread in line 1, the synchronized method in line 2, the synchronized statement in line 3, object that calls the notify() method in line 4 and the object that calls the wait() method in line 5. It is possible that each line of detection result is arranged randomly since the programming technique and logic of the multithreading API.

The result of this part will get all threads and the flow of the each thread when calling methods and locked objects. Figure 4 provides visualization of threads in the multithreading API from the result of the algorithm.

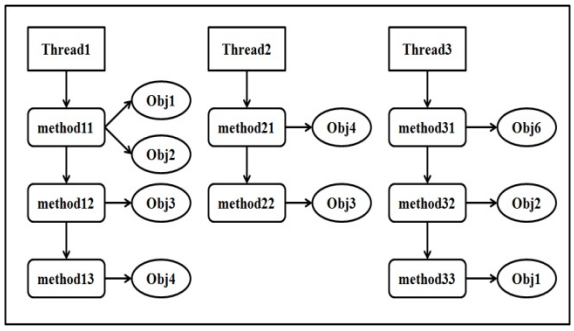

Figure 4: Threads from the result of the algorithm.

In Figure 4, it shows 3 threads that are Thread1, Thread2 and Thread3. Thread1 calls method11 locking Obj1 and Obj2 and then calls method12 locking Obj3 after that calls method13 locking Obj4. Thread2 calls method21 locking Obj4 and calls method22 locking Obj3. And Thread3 calls methods31 locking Obj6, calls methods32 locking Obj2 and calls methods33 locking Obj1.

4.3 Deadlock Detection

After getting all Threads, called methods, locked objects and their paths as structure shown in Figure 3 and thence the algorithm will check which objects are locked by another thread with reverse lock order. If there is more than one object called by another thread with reverse lock order, it is possible that the deadlock occurs. The algorithm of deadlock detecting is shown in Figure 5.

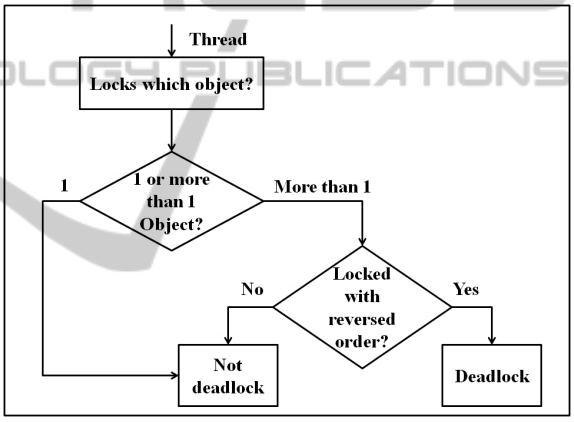

Figure 5: The deadlock detection algorithm.

The algorithm is designed using deadlock definition, concept of six conditions of deadlock (Naik et al., 2009) and a code pattern; Cyclic Lock Dependency (Otto et al., 2008). The deadlock occurs when there are two or more than two objects blocked by 2 threads in reverse order. Therefore Figure 5 shows that the algorithm will get locked objects of a thread. After get all objects, it will check whether there are 2 objects is locked by another thread or more than one thread or not. If 2 objects are locked by another thread or more than one thread, it will check whether the lock order is in reverse or not. If the lock order is a reverse order, it means that the deadlock can occur.

5 RESULT

Our deadlock detection algorithm is able to detect

the potential deadlock site in the source code of the multithreading API.

The result reports threads that occurs deadlock sites, objects that are locked, methods that call these objects and line numbers of the source code of the multithreading API for the deadlock site. Figure 6 shows the reported result of the algorithm.

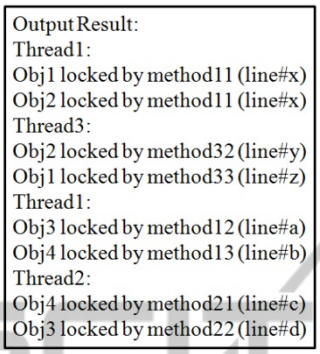

Figure 6: The report of deadlock site.

We compare the result of the deadlock detected by the algorithm with manual. And for other deadlocks that haven't been reported, we try to manually reproduce them and deadlocks encounter.

Table 6: The result of the detection algorithm compares with manual.

API version	Manual	Algorithm
1.0	3	9
2.0	4	6
2.1	2	2

Table 6 shows the result of the deadlock comparing between manual and the deadlock detection algorithm. We have tested the result with several versions of the multithreading API. API version 2.0 has addition features and was fixed defects from API version 1.0. API version 2.1 was fixed defects that were found in API version 2.0.

For the result, in API version 1.0, deadlock sites that are found by manual reproduction are 3 and by the deadlock detection algorithm are 9. In API version 2.0, deadlock sites that are found by manual reproduction are 4 and by the deadlock detection algorithm are 6. In API version 2.1, deadlock sites that are found by manual reproduction and by the deadlock detection algorithm are 2.

In summary, using manual reproduction is not sufficient to find all deadlocks in the multithreading API. Using the deadlock detection algorithm can help us to find more deadlock sites that have not been reported by customers, support consultants, developers or testers.

6 LIMITATION

In this research we do not focus on the performance of the deadlock detection algorithm in speed and CPU consumption and the false positive.

7 CONCLUSIONS AND FUTURE WORK

In summary, using static analysis to develop the deadlock detection algorithm can help us to detect deadlock sites in the source code of the multithreading API. Static analysis is appropriate to detect deadlocks in the multithreading API because we do not know exactly scenarios that the customer will use to develop their own application.

Our deadlock detection algorithm considers enough conditions to detect the deadlock from synchronized statements, synchronized methods, and wait-notify methods. The result is accurate and we can find more deadlock sites that have not been reported by manual reproduction before.

Therefore, we should add deadlock detection as a process in the Software Development, Software Testing or Software Maintenance phase of the Software Development Life Cycle of the multithreading API. Since detecting deadlock early can help we reduce defects before the application is developed using the multithreading API.

For the future work, it is possible to add more conditions to detect the deadlock for other conditions and programming designs such as using the Lock class and the Semaphore class in java.util.concurrent package for implementing the multithreading API. In addition, we can focus on the performance of the deadlock detection algorithm in speed and CPU consumption and the false positive to get more effective algorithm. Moreover, we can use other algorithms to find the root cause of the deadlock, not just only detect sites of the deadlock.

REFERENCES

Amy Williams, W. T., and Michael D. Ernst. (2005, 25-29th July). *Static Deadlock Detection for Java Libraries.* Paper presented at the ECOOP2005, Glasgow UK.

Ashcraft, D. E. a. K. (2003). *RacerX: Effective, Static Detection of Race Conditions and Deadlocks.* Paper presented at the SOSP '03.

Frank Otto, T. M. (2008). *Finding Synchronization Defects in Java Programs Extended Static Analyses*

and Code Patterns. Paper presented at the IWMSE'08, Leipzig, Germany.

Gianpiero Francesca, A. S., Gigliola Vaglini and Maria Luisa Villani. (2011, 18-22 July). *Ant Colony Optimization for Deadlock Detection in Concurrent Systems.* Paper presented at the 2011 35th IEEE Annual Computer Software and Applications Conference, Washington, DC, USA.

Holt, R. C. (1972). Some Deadlock Properties of Computer Systems. *Computing Surveys, 4*(3), 18.

Jyotirmoy Deshmukh, E. A. E. a. S. S. (2009). *Symbolic Deadlock Analysis in Concurrent Libraries and Their Clients.* Paper presented at the 2009 IEEE/ACM International Conference on Automated Software Engineering, Washington, DC, USA.

Mayur Naik, C.-S. P., Koushik Sen and David Gay. (2009). *Effective Static Deadlock Detection*. Paper presented at the ICSE'09, Vancouver, Canada.

Mikhail Moiseev, A. Z., Ilya Klotchkov and Sergey Salishev. (2011). *Static analysis method for deadlock detection in SystemC designs*. Paper presented at the SoC 2011, Tampere, Finland.

Pallavi Joshi, C.-S. P., Koushik Sen and Mayur Naik. (2009). *A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks.* Paper presented at the PLDI '09, New York, USA.

Qichang Chen, L. W., Ping Guo, and He Huang. (2011). Analyzing Concurrent Programs for Potential Programming Errors. *Modern Software Engineering Concepts and Practices: Advanced Approaches*, 1-43. doi: 10.4018/978-1-60960-215-4.ch016

Tong Li, C. S. E., Alvin R. Lebeck, and Daniel J. Sorin. (2005, April 10–15). *Pulse: A Dynamic Deadlock Detection MechanismUsing Speculative Execution.* Paper presented at the 2005 USENIX Annual Technical Conference, Anaheim, California.