Visualization of Deadlock and Wait-Notify Anomaly in Multithreaded Programs

Thesis submitted in partial fulfillment of the requirements for the award of degree of

Master of Engineering
in
Software Engineering

Submitted By
Aman Jyoti
(801231002)

Under the supervision of:
Mr. Vinay Arora
Assistant Professor

COMPUTER SCIENCE AND ENGINEERING DEPARTMENT
THAPAR UNIVERSITY
PATIALA – 147004

June 2014

Certificate

I hereby certify that the work which is being presented in the thesis entitled, "*Visualization of Deadlock and Wait-Notify Anomaly in Multithreaded Programs*", in partial fulfillment of the requirements for the award of degree of Master of Engineering in *Software Engineering* submitted in Computer Science and Engineering Department of Thapar University, Patiala, is an authentic record of my own work carried out under the supervision of *Mr. Vinay Arora* and refers other researcher's work which are duly listed in the reference section.

The matter presented in the thesis has not been submitted for award of any other degree of this or any other University.

(Aman Jyoti)

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(Mr. Vinay Arora)
Assistant Professor,
CSE Department

Countersigned by

(Dr. Deepak Garg)
Head
Computer Science and Engineering Department
Thapar University
Patiala

(Dr. S. K. Mohapatra)
Dean (Academic Affairs)
Thapar University
Patiala

Acknowledgement

First of all, I would like to thank the Almighty, who has always guided me to work on the right path of the life. Due to mercy of God, it has been possible for me to reach so far.

This work would not have been possible without the encouragement and valuable guidance of my supervisor **Mr. Vinay Arora**, Assistant Professor, Thapar University, Patiala. I thank my supervisor for his time, patience, discussions and valuable comments.

I am equally grateful to **Dr. Deepak Garg**, Associate Professor and Head, Computer Science and Engineering Department, for motivation and inspiration that triggered me for the thesis work. I am also thankful to the entire faculty and staff members of Computer Science and Engineering Department for their direct-indirect help, cooperation, love and affection, which made my stay at Thapar University memorable.

I will be failing in my duty if I don't express my gratitude to **Dr. S. K. Mohapatra,** Senior Professor and Dean of Academic Affairs the University, for making provisions of infrastructure such as library facilities, computer labs equipped with net facilities, immensely useful for the learners to equip themselves with the latest in the field.

Last but not least, I would like to thank my parents for their wonderful love and encouragement, without their blessings none of this would have been possible. I would also like to thank my brother for his constant support throughout the thesis.

Abstract

Multithreaded programs deal with simultaneous execution of multiple threads. There are various types of bugs that can occur during execution of multithreaded programs like deadlock, livelock, race condition and synchronization faults. These bugs are difficult to detect and correct due to non-deterministic nature of multithreaded program execution. This thesis work presents a critical analysis of these bugs by categorizing them on the basis of their detection and visualization techniques.

Here, deadlock due to lock acquisition and wait-notify dependency have been addressed. Dependencies have been visualized with four types of dependence graphs, namely data-control, lock acquisition, wait-notify and thread dependence graphs. A prototype tool Bug Visualizer has been proposed to provide a graphical representation for multithreaded programs and finally to locate the bugs. Suspicion of deadlock is detected due to the presence of cycle in lock dependence and thread dependence graphs. Algorithms have also been proposed for visualizing deadlock and wait-notify anomaly in multithreaded programs.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Multithreaded programs have simultaneous execution of multiple threads [1]. There are various types of bugs like deadlock, livelock, race condition and other synchronization faults that can occur in multithreaded programs. Visualization of these bugs helps in easy understanding of cause and location of the bug, which leads to effective debugging of the program. This chapter introduces various terms and bugs related to multithreaded programs along with their visualization tools.

1.1 Multithreaded Programs

Multithreaded programs deal with simultaneous execution of more than one thread. There is difference between parallelism and concurrency in multithreaded system [1]. Parallelism means at least two threads literally execute at the same time. This can be possible with multiple CPUs like in multiprocessor system. In this, threads run in parallel on separate processors whereas, there is illusion of parallelism in a concurrent multithreaded system. Some terms related to multithreaded programs are discussed in this section.

1.1.1 Concurrency

Concurrency can be defined as overlapped execution of processes/threads *i.e.* before one process completes its execution, other process starts executing [1]. In concurrent thread execution, threads make progress, however this progress needs not to be parallel. In this approach start and complete (fork and join) operations are used for specifying start and completion points of processes.

Figure 1.1 shows fork and join operations in a system having three sub-systems S_1, S_2 and S_3. Fork is used to represent concurrently executable sub-systems, whereas join is used to combine the result of those sub-systems, which can execute concurrently. S_1 and S_2 are two sub-systems, which execute concurrently as shown after fork operation. Sub-system S_3 is introduced after performing join operation over sub-systems S_1 and S_2.

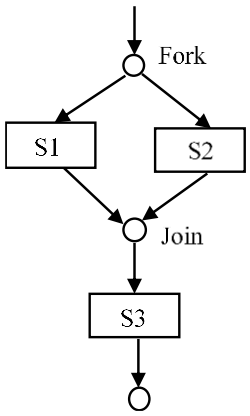


Figure 1.1 Fork and join operations [2]

1.1.2 Semaphore

Semaphore is a variable with integer value, which is used for signaling mechanism. Except its initialization, it can be accessed only through following two operations [3]:

i. Wait: In this operation, value of semaphore is decremented by 1. It is also known as *proberen* (P), down and lock.
ii. Signal: In this operation, value of semaphore variable is incremented by 1. It is also known as *verhogen* (V), up and unlock.

Figure 1.2 shows semaphore S for a critical section having entry and exit point. At entry point, there is wait operation over semaphore S. After that, critical section is executed and then signal operation is performed over semaphore S.

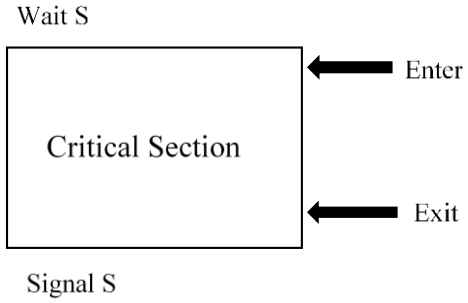


Figure 1.2 Wait and signal operation on semaphore S [4]

There are several types of semaphores, however basic idea for all of them is same. Two main types of semaphores are discussed below [3]:

i. Binary Semaphore: This is used for accessing single resource in a mutually exclusive way like serial port, hard disk drive *etc*. Here semaphore variable is

having only 0 or 1 value so it is called binary semaphore. Also, this can be stated as counting semaphore having maximum value 1.

ii. Counting Semaphore: When there are multiple resources to be handled in mutually exclusive way then counting semaphores are used. Counting semaphores are having maximum value equal to the number of resources like 4 printers or 5 memory buffers *etc*.

Semaphores can be used in the following two ways [5]:

i. For controlling access to shared devices among several tasks: Consider an example of printer used by two tasks. As two tasks cannot access printer simultaneously therefore, a binary semaphore is created for accessing printer in a mutually exclusive way. Now for using printer, it should be acquired by any task, if it is available, else that task has to wait for the printer till the signal comes for acquiring semaphore.
ii. For task synchronization: Concurrent programs produce unpredictable results due to non-deterministic nature of their execution. Therefore, there is a need to force the tasks for performing certain operations in a particular order. This work is also carried out by using semaphores.

1.1.3 Mutex

Mutex can be defined as mutual exclusion object that is used for locking shared resources among multiple threads [5]. It is mainly used by multiple threads to access re-entrant code turn by turn. Mutex is based on ownership concept *i.e.* if one thread acquires lock for a resource then only that particular thread can free that resource after finishing its task [6].

Figure 1.3 represents a section of Java code as an example for mutex. In this example, mutex.acquire() function is used for acquiring mutex lock. Mutex lock blocks the entry of all other threads for the section between lock acquiring and release operation. Thread, which has acquired the lock, can proceed further and complete its task. In this example, finally block is used for releasing mutex lock with mutex.release() operation.

```
try
{
        mutex.acquire();
        try
        {// do something
        }
        finally
        {mutex.release();
        }
}
catch(InterruptedException ie)
{ }
```

Figure 1.3 Java mutex example [6]

1.1.4 Critical Section

Critical section can be defined as a synchronized code segment that is shared among multiple threads. Critical section problem ensures that no more than one thread can enter in the shared section [2]. Solution to this problem involves mutual exclusion, progress and bounded wait [3].

Strucure of a process having critical section is shown in Figure 1.4. At the start of critical section there is an entry section. Before executing critical section, entry condition is checked in entry section. At the end of critical section, there is an exit section having some exit conditions which are needed to be fulfilled after executing the critical section.

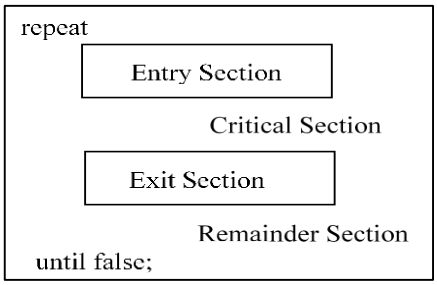


Figure 1.4 Structure of a process having critical section [3]

1.2 Bugs in Multithreaded Programs

Multithreaded programs are prone to many types of bugs like deadlock, livelock, starvation, race condition and wait-notify anomaly. Simultaneous execution of multiple threads and their interleavings make bug detection a trivial task. Brief description of these bugs is given in this section.

1.2.1 Livelock

In a multithreaded program, livelock is a state in which program neither becomes deadlocked nor it gets terminated, but it does not progress. It is also referred as busy waiting in a loop for a condition that is never going to be true [7].

In Figure 1.5, livelock is shown with an example of a road and two passengers. Initially both passengers A and B are on left lane of the road for crossing it. Then after inspecting blocked situation, both decided to move to right lane of the road. But now also lane is blocked for each other due to each other. Then they again move to left lane and continue like this. This leads to the livelock situation when both passengers A and B are moving and still having no progress.

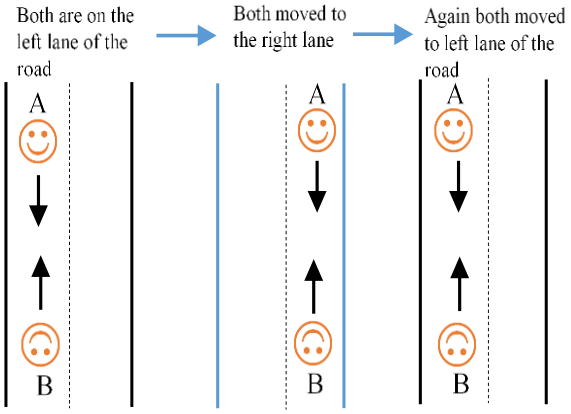


Figure 1.5 Livelock example [7]

1.2.2 Starvation

Starvation can be stated as waiting for infinite time for a non-sharable resource [8]. In this situation, all processes desire to execute in a mutually exclusive manner. A process is starved if it is waiting for a shared resource but other higher priority processes repeatedly access that resource.

There can be any of the following cause for starvation [9]:

i. Process priorities are fixed and strict: If priorities of processes are fixed then a lower priority process will always get less preference than higher priority processes and will get starved for shared resources.

ii. Random selection procedure is used instead of queue mechanism: If processes are selected randomly then there can be a chance that a process will never be chosen for resource allocation and it will get starved.

iii. Resources demand can exceed its supply: If supply of resources is not according to their demand then some processes will get starved due to their high demand of less available resources.

Figure 1.6 shows a starvation situation of timid driver in a one-lane bridge. Here timid driver waits for crossing of other cars, which are on opposite side of bridge, to cross a narrow bridge. This indefinite wait for oncoming cars to exit the bridge, leads to starvation condition.

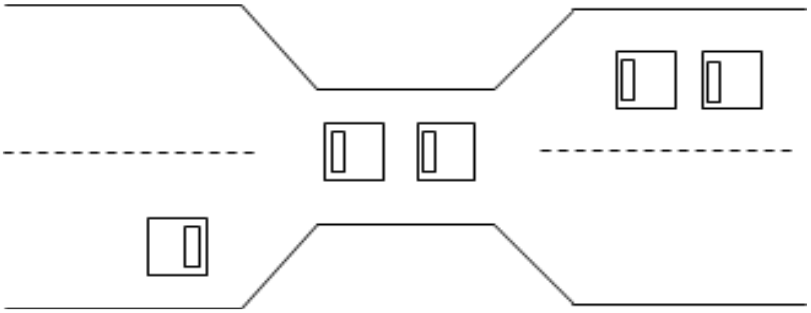


Figure 1.6 Timid driver example of starvation in one-lane bridge [10]

1.2.3 Race Condition

Race condition arrives when two or more threads access shared data simultaneously and change it [3]. This leads to unpredictable output for that shared data because the order in which threads access shared data/variable is unknown [8]. Race condition requires three things [11]:

i. Concurrency: System must have at least two control flows executing simultaneously in concurrent manner. Threads can be those control flows in multithreaded system.

ii. Shared Data: There must be some data which is shared among threads in concurrent system. That shared data is called as race object which is accessed by concurrent control flow.

iii. Change of State: At least one control flow must do some changes in the state of shared data or race object.

Race condition occurs in “check-then-act” scenario as shown in Figure 1.7. In this scenario, one thread checks some condition and then performs some action. However, between “check” and “act” step, some other thread changes the state of race object *i.e.* value of variable ‘m’.

```
if (m == 2) // “Check” step
{
z = m * 5; // “Act” step
// If other thread changed m in between “check” and “act” then z
                    //will not contain value 10.
}
```

Figure 1.7 Check-Then-Act example [12]

Lock can be acquired for shared data to prevent race condition shown in Figure 1.7. In Figure 1.8, lock is acquired over variable ‘m’ to prevent race condition. Locking of variable ‘m’ ensures that no one else can change its value while it is being accessed by one thread to manipulate the value of variable ‘z’.

```
// Obtain lock for m
if (m == 5)      // “Check” step
{
        z = m * 2;              // “Act” step
// Now, no one can alter m until the lock is released. Therefore z contain //value 10.
}
// release lock for m
```

Figure 1.8 Lock acquisition example to prevent race condition [12]

1.2.4 Deadlock

Deadlock is a state in which program execution gets blocked and it remains blocked forever [8]. Deadlock situation can be suspected with cyclic wait among tasks for non-sharable resources. Deadlock can be detected by showing dependencies of processes for acquiring resources with the help of wait-for graph. Presence of cycle in wait-for graph shows deadlock condition in the system.

Consider processes P_1 and P_2 as shown in Figure 1.9 for understanding deadlock condition. In this wait-for graph, process P_1 acquires resource B and waits for resource A, which is acquired by process P_2. Also, process P_2 is waiting for process P_1 for acquiring resource B. Here cyclic wait is present between P_1 and P_2, therefore it leads to deadlock situation [3].

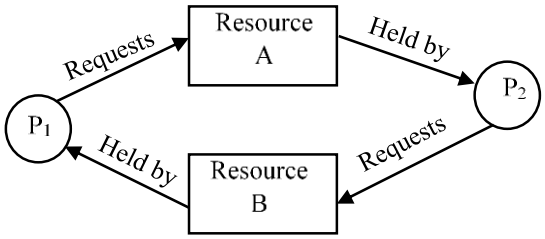


Figure 1.9 Deadlock example [3]

Following four conditions are necessary to hold simultaneously for a deadlock situation [3]:

i. Mutual Exclusion: There should be at least one resource that is not shared among processes. If any other process want that resource then that request must be delayed till previous process release that particular resource.
ii. Hold and Wait: There must be a situation when one process is holding some resource(s) and simultaneously waiting for other resource(s) acquired by other process.
iii. No Preemption: There should not be preemption of resources. Only the process which has acquired the resource can free that resource after completing its task.
iv. Circular Wait: There should be circular wait among a set of resources. If P_0, P_1 and P_3 form a set then, P_0 waits for P_1, P_1 waits for P_2 and P_2 waits for P_3.

1.3 Visualization

Card *et al.* [13] defined the term visualization as "The use of computer-supported interactive visual representations of data to amplify cognition". Visualization helps in easy understanding of essential facts to quickly see outliers and regularities.

There are various tools available for visualizing programs. Graphs and UML diagrams are the main tools used for this purpose. Basic details of these two are as follows:

1.3.1 Graph

A graph G (V, E) consists of a non-empty finite set of vertices, V and a finite set of edges, E. In any graph, edges are used to represent flow, dependency or any other relationship between vertices. Different types of graphs are discussed in this sub-section.

i. Control Flow Graph (CFG) [14]: In a control flow graph (or flow graph) G, an edge (i, j) joins two nodes n_i and n_j. G = (N, E) represents a flow graph G with nodes contained in a finite set N and a finite set E representing edges. In a CFG, edges represent flow of control between vertices.

Figure 1.10 shows control flow graph for an example program by providing numbers to each statement of the program. In the CFG, these statement numbers are used for showing control flow form one statement to other statement. Some statements can be grouped into a block and a block number is provided for its representation like A, B, C *etc*.

```
main()
{
        int number, index;
1.      printf("Enter a number");
2.      scanf("%d", &number);
3.      index=2;
4.      while(index<=number-1)
5.      {
6.              if(number%index==0)
7.              {
8.                      printf("Not a prime number");
9.                      break;
10.             }
11.             index++;
12.     }
13.     If(index==number)
14.             printf("prime number");
15.     }//end main
```

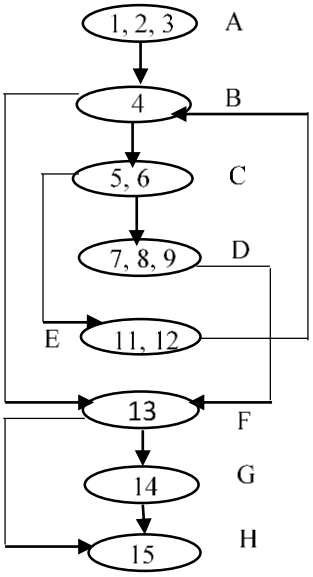


Figure 1.10 Control flow graph [14]

ii. Data Flow Graph (DFG) [14]: DFG is also known as def-use graph as it shows relationship between nodes where the variable is defined and used. It is a control flow graph with additional information of each node about definition and use of data.

Figure 1.11 shows an example data flow graph in which definition, computational use and predicate use of variables are shown. In a DFG, 'def' is used for definition, 'c-use' is used for computational use and 'p-use' is used for predicate use. This figure also shows grouping of statements into blocks and then providing labels to those blocks. Further, these labels are used for creating the data flow graph.

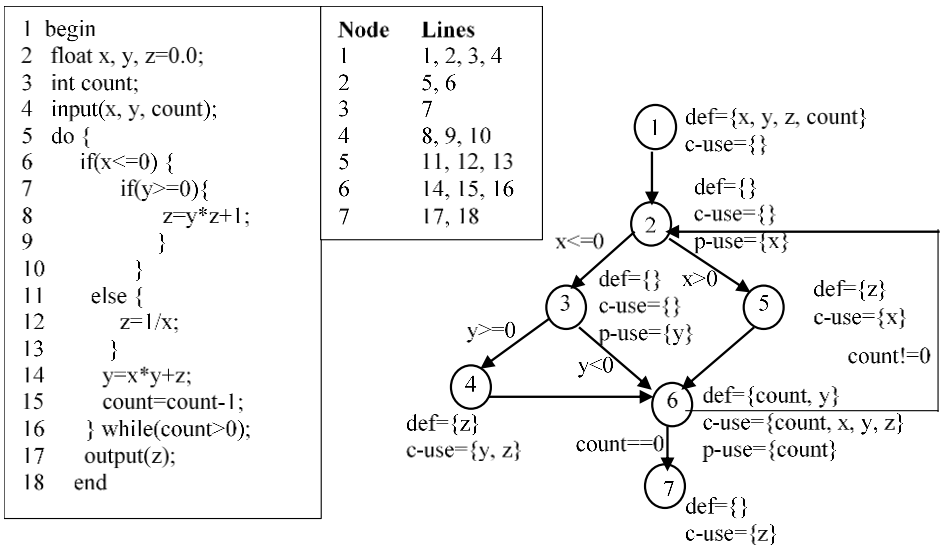


Figure 1.11 Data flow graph [14]

iii. Control Dependence Graph (CDG) [14]: A control dependence graph is a directed graph showing dependencies among nodes. Nodes m_2 is control dependent over node m_1 if there exist a path from node m_1 to program exit node including node m_2.

Figure 1.12 shows a control dependence graph by considering a sample code as shown in the figure. Node 7, 8 and 9 are control dependent over node 6 as node 6 is decision making node. Decision of node 6 will choose whether control will come over node 7, 8 and 9 or not.

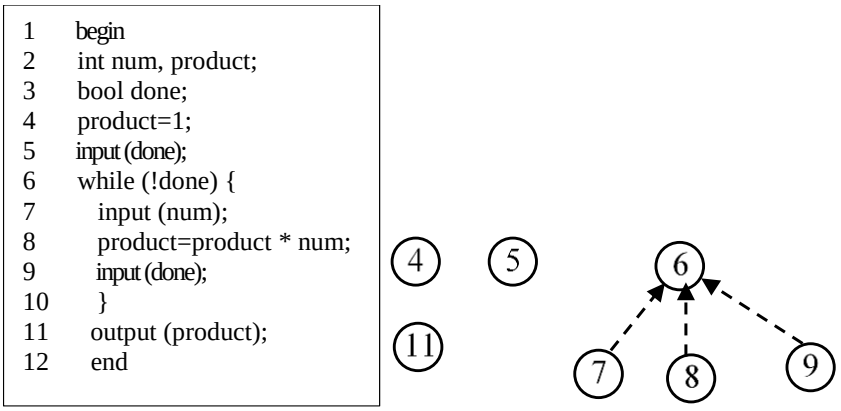


Figure 1.12 Control dependence graph [14]

iv. Data Dependence Graph (DDG) [14]: Similar to control dependence graphs, data dependence graph is also a directed graph showing data dependency between nodes. It shows the relation between those nodes which defines and uses value of a variable.

Figure 1.13 shows an example data dependence graph with the help of sample code. Here statement 8 is dependent over itself, statement 7 and 4 for definition of variable ‘product’. Similarly statement 6 is dependent over statement 5 and 9 for definition of variable ‘done’ and so on.

```
1    begin
2    int num, product;
3    bool done;
4    product=1;
5    input(done);
6    while (!done) {
7      input (num);
8      product=product * num;
9      input(done);
10   }
11   output (product);
12   end
```

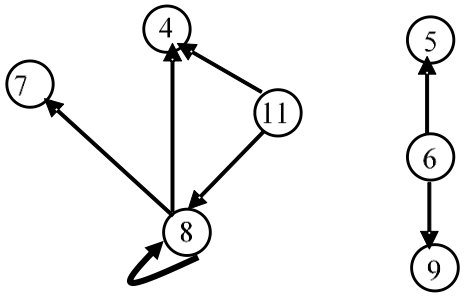


Figure 1.13 Data dependence graph [14]

1.3.2 Unified Modeling Language (UML):

UML diagrams are also useful for effective visualization of the programs. Multithreaded tasks can be visualized using following UML diagrams:

i. Activity Diagram [15]: This diagram shows flow of activities with in a system. Transitions are shown from one activity to another activity. In this diagram, fork and join are used to show concurrent tasks.

Figure 1.14 shows an activity diagram with fork and join to show concurrency in the system. This diagram shows two concurrently executing activities A_2 and A_3.

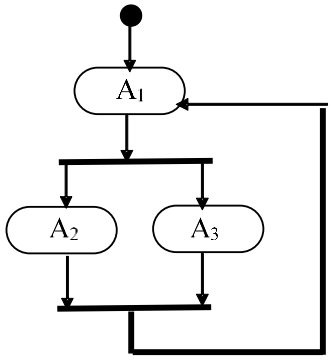


Figure 1.14 Activity diagram with fork and join [16]

ii. Collaboration Diagram [15]: In this diagram, all components which are going to interact, are collaborated together. Objects are connected with links and these

links are adorned with messages that are sent and received by the objects as shown in Figure 1.15. This makes a clear visualization of whole flow of control among objects.

Figure 1.15 shows a collaboration diagram having active objects and concurrent messages. Active object is shown by a rectangle with thick border and concurrent messages are having same label before dot separator like A.1 and A.2.

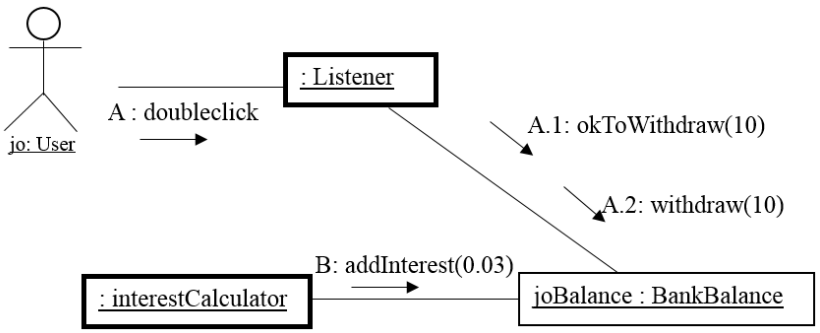


Figure 1.15 Collaboration diagram [16]

iii. Sequence Diagram [15]: This diagram shows messages according to their time ordering. This diagram gives a clear visualization of flow of control with time sequencing. There is an object lifeline shown with vertical dashed line in the diagram for each object. Also there is a focus of control that shows active period of an object with a thin, tall rectangle.

Figure 1.16 shows an example of sequence diagram with additional information of time ordering, which is not shown by collaboration diagram. This example shows that concurrent messages A.1 and A.2 execute after message B.

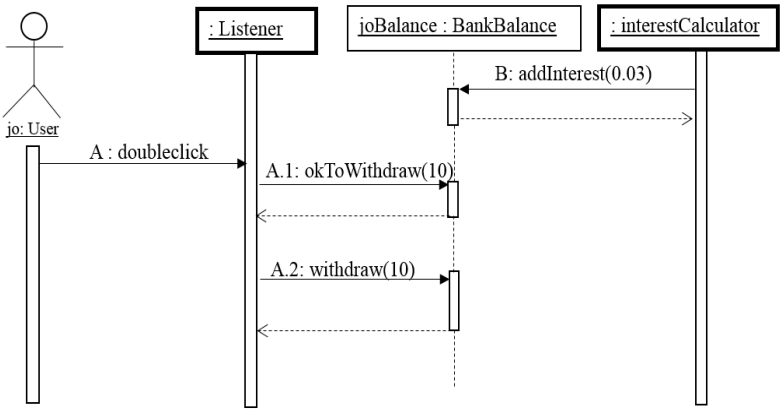


Figure 1.16 Sequence diagram [16]

iv. Statechart Diagram [15]: In statechart diagram, there is flow of control from one state to another state. This diagram shows sequence of states in response to the events an object goes through in its lifetime. Transition from one state to another state occurs if some event takes place and some conditions are satisfied.

In Figure 1.17, concurrent regions of a state are shown by splitting it with fork (left side vertical bar) and join (right side vertical bar). In a statechart diagram, concurrency can also be shown by splitting concurrent sub-states with a horizontal dash line instead of fork and join.

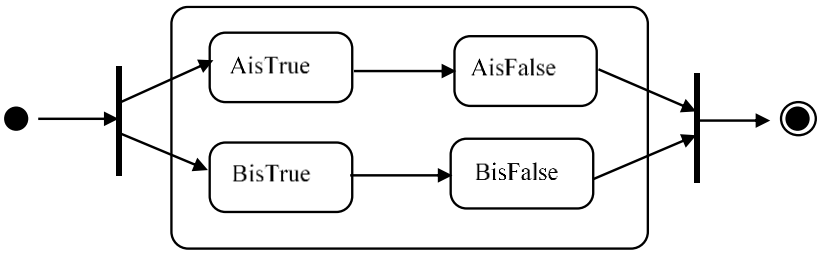


Figure 1.17 Statechart diagram [16]

Chapter 2
Literature Survey

This chapter categorizes the work that has been previously done for detecting various bugs of multithreaded programs. To better cognize the concept of multithreading and its debugging, multithreaded programs are visualized. This section also summarizes earlier proposed visualization techniques for multithreaded programs. At the end of the section, various tools for visualization and debugging of multithreaded programs, are summarized in a table.

2.1 Deadlock Detection Techniques

Deadlock is a situation in which a group of threads wait for some other event or task for some resource(s) in a cyclic manner and this wait never gets over [8]. Detailed categorization of deadlock detection techniques is discussed in this section.

2.1.1 Specification Based Detection

During early nineties, N.D. Francesco, and G. Vaglini [17] proposed a specification based approach for deadlock detection. In this approach, static and dynamic tools are used for checking specifications of a program. W. Araujo *et al.* [18] used Java modelling language (JML) as specification language for writing contracts with Java program. Java classes are instrumented with runtime assertion checking (RAC) code, which is generated by compiler of JML toolset after compiling the specifications. Race and deadlock conditions are detected with the help of contract based assertions. A methodology defined in research paper [19] captured the behavior of program in terms of type and effect system for deadlock detection. By using abstract system, cycles are detected in which processes wait for shared resources. In another approach, class under test (CUT) is given as input and a single output, bug report is generated by proposed tool [20]. Bug report for input thread safe class shows exception and deadlock. Later, for detecting race condition, deadlock and atomicity violation in concurrent program, concurrent predicates (CP) and concurrent predicate expressions (CPE) [21] are presented as instrumentation for developers. It can help in capturing the program state and necessary schedule for detecting specific bug.

2.1.2 Event Based Detection

Instead of using wait-for graph, D. Feitelson [22] detected deadlock when all events are in waiting stage. The only demerit of this algorithm is that there can be a chance that a process can be in busy waiting instead of deadlock. In paper [23], deadlock is detected dynamically by proposing three phases for its detection. In first phase, program is executed and critical events are monitored with their log creation for lock dependency. In second phase, Magiclock algorithm is used for detecting CycleSet of all possible lock dependency by depth first searching. Then in third phase, MagicFuzzer is provided with CycleSet as input and it executes program for detecting deadlock cycles in CycleSet. A methodology named counterexample-guided abstraction refinement (CEGAR), is also proposed in paper [24]. In this scheme, abstraction and compositional reasoning are combined for deadlock detection. Figure 2.1 shows components of a system represented in the form of state machine as labeled transition systems (LTS). Use of abstractions for verification reduces state space explosion problem.

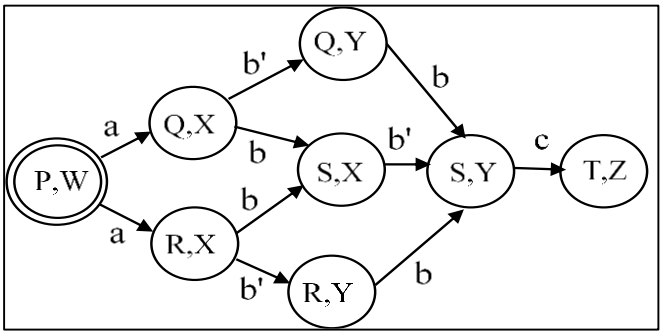


Figure 2.1 An example of labeled transition system [24]

2.1.3 Petri Net Based Detection

Proper termination, determinacy and deadlock detection for parallel programs written in occam language, is done using petri nets graph model [25]. Non-termination of a parallel program can be detected as infinite looping and infinite waiting for which four type of anomalies are detected. In this paper, main algorithm uses contraction procedure and static communication deadlock detection procedure. In paper [26], programmable logic controller (PLC) programs of automated machining cell (AMC) are tested for deadlocks and conflicts. Control model is developed using petri nets for PLC programs and deadlock detecting algorithm is applied over it. In papers [27, 28], structural analysis of system is proposed using a class of petri nets named as ES^3PR net model for detecting

deadlock. Three algorithms are used in this approach, one is supervisor synthesis for an ES^{3}PR and two algorithms are for rearrangement of output arcs of monitors.

2.1.4 Graph Based Detection

Deadlocks can also be determined by using geometric description for constructing partial order directed graph whose local maxima represents deadlock [29]. K.C. Tai [8] defined deadlock in terms of reachability graph and also proposed an algorithm for its detection. In research paper [30], program written in SHIM concurrent language, is sub-divided into a tree of tasks. Then each task is abstracted as an automation for performing communication, invoking and waiting for its children. Finally, these are converted into NuSMV symbolic model checker compatible form, which identifies deadlock. In research paper [31], dynamic synchronization dependence graph data structure is created in system level designs. Then loop detection algorithm is applied over dependence graph to detect deadlock. Lock graph is created in research paper [32] after capturing lock acquisition sequences and cycle in this graph shows presence of deadlock. However, false positive results can occur due to cycle inside one thread, cycle that is guarded by gate lock or may be due to cycle involving non-concurrent fragments. In research paper [33], bug locations are identified for single variable using Falcon tool. For detecting concurrency bugs UNICORN tool is used. Then Griffin is used for understanding the identified bugs using bug graph as shown in Figure 2.2 and clustering algorithm.

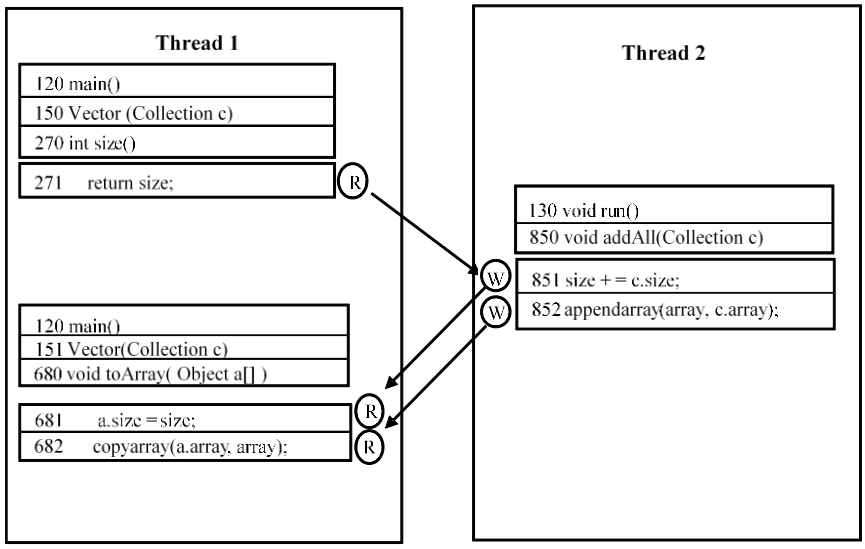


Figure 2.2 Bug graph [33]

2.1.5 Heuristic Based Detection

In the methodology presented in paper [34], heuristics are proposed to solve problems related to state space explosion for concurrent programs. Evaluation function finds out distance to goal node using heuristics to solve the problem. Searching algorithm A* guides to expand that state first which takes to the detection of deadlock present in the system. Other searching algorithms like greedy, IDA* are used for handling output interpretation problem in deadlock detection [35]. In another approach, constraints are imposed on total ordering of resources using resource ordering method (ROM) algorithm. Processes are required to request passive resources in equivalence classes based on ordering done by resource ordering algorithm [36]. Now resource requests generated by run time system fulfill AND-OR request condition and prevents deadlock.

2.2 Race Detection Techniques

Race condition occurs when two or more threads access shared data simultaneously and change it, which leads to unpredictable output for that shared data [3]. This section discusses various race detection approaches.

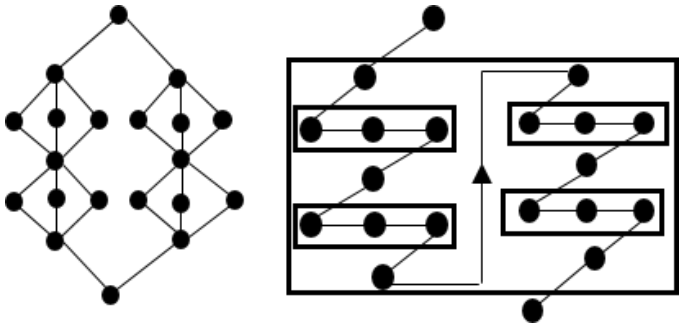
2.2.1 Logical Clock Based Detection

A race detection algorithm is defined for blocking as well as non-blocking send and receive message passing in parallel programs using logical clocks [37]. It is on-the-fly race detection algorithm so it saves memory for storing states of the program. This methodology can also be used for avoiding deadlocks in parallel programs. This paper also shows a scope for developing a tool in future to find out whether an execution of message passing program is portable and repeatable.

2.2.2 Monitoring Memory Accesses

For variable tracing of a block, all memory accesses are monitored including read and write accesses using address tracing [38]. In the given approach, a series parallel task graph is generated which represents sequential execution of concurrent program as shown in Figure 2.3 and then race is detected using variable access sets. A new approach for automatic identification of shared variables is stated using efficient dataflow algorithm [39]. Next step is to do pointer analysis to figure out aliases for

pointer locks. Finally, a new technique is proposed for warning reduction. Two algorithms, dataflow analysis for shared variable detection and summary computation for lock pointer analysis, are used in this approach.



(a) (b)

Figure 2.3 (a) Series-parallel task graph (b) Sequential execution task graph for race detection [38]

2.2.3 Graph Based Detection

Error detection is done through manipulation of event graph generated by recording run time information of program [40]. A tool for event manipulation (ATEMPT) is used for manipulating event graph. Race condition is detected with other communication errors in distributed share memory access, using this methodology. Monitoring of input/output access for performance is still to be done as an extension to the proposed work. In the approach given by M.-Y. Park *et al.* [41], unaffected races are detected and then visualized. The idea for detecting unaffected races came from the fact that mostly affected races are due to side effect of unaffected races. Affecter algorithm based on "happened before relation" is used for determining affecters. Then graph is created for affecter-relations and affected-relations for detecting races. Y. Chen *et al.* [42] generated race condition graph (RCG) for representing race condition in the system. It helps in visualizing race condition and program interaction. Then potential race condition graph (PRCG) is generated from RCG. Various classification of race condition is shown in Figure 2.4.

Data race in ARNIC-653 health monitor is detected dynamically using a visualization tool, which includes runtime monitor and visualizer [43]. In this approach, events are visualized by showing them with different colors. Two accesses to a shared variable are concurrent if they are not connected with each other with same color. In another paper of static analysis [44], transactions are captured in the form of transaction graph using

partial order reduction techniques and scheduling constraints. Then graph is refined by removing unreachable nodes iteratively which enhances scalability. This also reduces false warnings captured by initial lockset analysis. Two algorithms are used in this technique, namely transaction delineation based on static partial order reductions (POR) and iterative refinement of transaction graph.

2.2.4 Using Model Checker

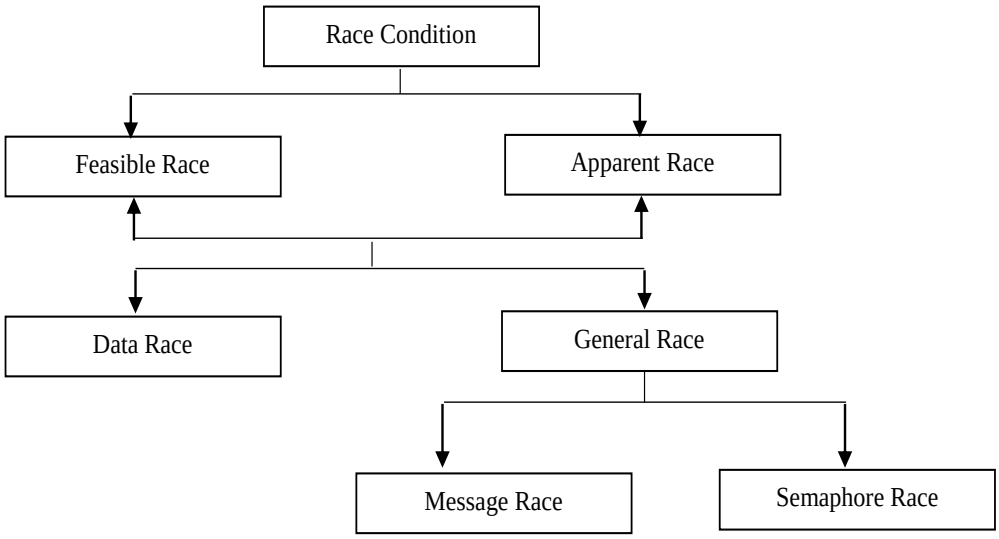


Figure 2.4 Race condition classification [42]

E. Yoshida and H. Kakugawa [45] defined an approach which converted assembly language program into process meta language (PROMELA) using a simulator. Then simple Promela interpreter (SPIN) model checker takes Promela as its input and detects mutual exclusion bugs in program. If any race, deadlock or starvation error is found, it is reported with that particular thread sequence which causes error, for understanding its cause. Aspect oriented programming language [46] is extended for three new pointcuts *i.e.* lock(), unlock(), and maybeshared() for detecting data races, which are not detected earlier by the tool Eraser. These new pointcuts are implemented as an extension to aspect bench compiler. New algorithm RACER is an adaptation of the Eraser algorithm.

2.2.5 UML Based Detection

In this approach, data race is detected at design phase so that it can be easily removed. Design model expressed in UML is analyzed for data race detection [47]. Then genetic algorithm is used for identifying concurrency problems like data race, deadlock and starvation.

2.2.6 Using Code Instrumentation

With Java programs, a specification language named as Java modeling language (JML) is used for writing contracts [18]. Compiler of JML toolset is used to translate specifications into runtime assertion checking (RAC) code. Therefore, instrumentation of Java classes is done by adding RAC code to them. Results of industrial case study show that contract based assertions are helpful in detecting race and deadlock in concurrent programs. Faults are at immediate vicinity of their assertion points as shown in Figure 2.5.

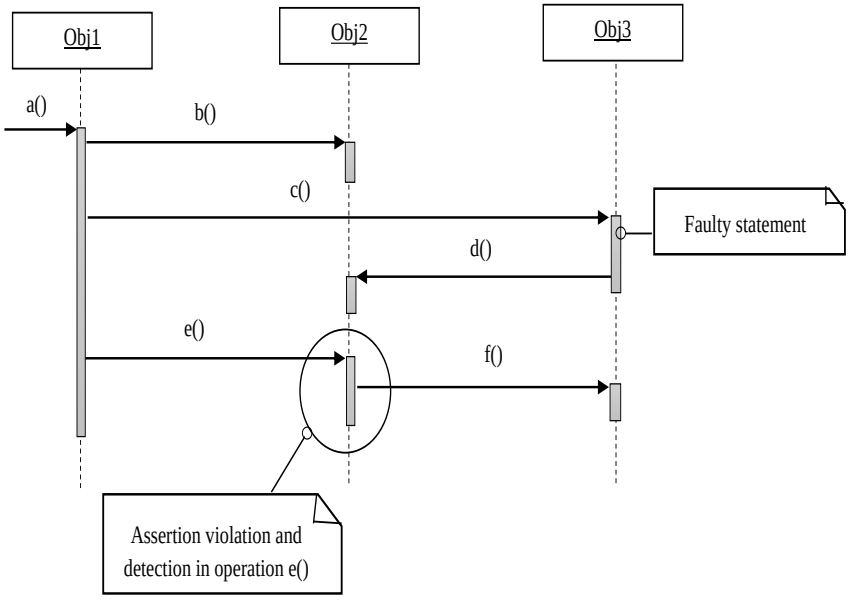


Figure 2.5 Diagnosis of concurrent faults [18]

2.2.7 Mix of Software and Hardware Approach

An approach having better performance and accuracy is proposed with RADISH race detector [48]. Already proposed *software* approaches are with high overheads and *hardware* approaches generally miss some data races. Therefore, RADISH detector is proposed with the benefits of both *software* and *hardware* approach without having their limitations.

A comparison of already proposed techniques is also done using parameters like additional synchronization, execution traces, data race conditions and fault recovery [49]. This survey identifies that little communication among processes diminishes data race condition.

2.3 Other Bug Detection Techniques

Multithreaded programs are prone to many bugs other than deadlock and race condition. Detection techniques for these bugs are discussed in this section.

2.3.1 Starvation

In terms of reachability graph of concurrent programs, starvation can be defined as infinite waiting of a process for a resource that is repeatedly acquired by other higher priority processes [8]. E. Yoshida and H. Kakugawa [45] proposed a methodology for starvation detection using SPIN model checker. A program written in assembly language is given as input to OS simulator which converts it into PROMELA. A SPIN model checker takes PROMELA as its input and detects mutual exclusion bugs in program. If any error is found then it is reported with that particular error causing thread sequence so that the cause of error could be understood. SPIN model checker is used for detecting mutual exclusion bugs like starvation. K.C. Tai [8] also defined an algorithm for detecting starvation.

2.3.2 Livelock

In terms of reachability graph of concurrent programs, livelock is defined as busy waiting with no progress [8]. For detecting livelocks, D. Lugato *et al.* [50] defined an approach based on UML. Using UML editor Objecteering, UML model (class diagram and state machine) is specified and translated into AGATHA extended input output labeled transition system (A-EIOLTS) specification by implementing two step specification generator. A-EIOLTS specification is given as input to AGATHA (atelier de g´en´eration automatique de tests holistiques pour automates) tool for developing symbolic execution tree. Symbolic test cases are generated by following each path of the symbolic tree which are further translated into sequence diagram for user visualization. After computing execution tree, livelocks become visible for the system. An algorithm defined by K.C. Tai [8] is also used for detecting livelocks.

2.3.3 Inconsistency and Order Violation Bugs

There are several other bugs, like correlated variable access bug which can occur in a multithreaded programs. In research paper [51], source code analysis and data mining

techniques are combined for automatic detection of correlated variable accesses and then detecting concurrency bugs. Multi-variable inconsistent updates and multi-variable concurrency bugs are detected by the proposed approach. This approach detected four new concurrency bugs in Mozilla.

Order violation bugs can also occur in multithreaded programs, which are needed to be detected and removed. A study done by S. Lu *et al.* [52] helps in developing tools for order violation bugs and multiple-variable bugs. It can also help in focusing small group of memory for the best utilization of testing efforts. Analysis and dynamic exploration techniques are combined in the approach proposed by E. Trainin *et al.* [53]. Results of this approach suggest that better models and better forcing algorithms improve efficiency of finding bugs.

2.4 Visualization of Multithreaded Programs

In the present era of multithreading, most of the tasks are executed in a concurrent manner. To better cognize the concept of multithreading, an expressive way is needed *i.e.* visualization. This can be used for many purposes [54]:

i. Error Detection: Visualization helps in tracing the errors by showing their exact location in the system.
ii. Better Comprehension: Understandability of program increases via visualization.
iii. Deadlock Detection: Visualization helps in deadlock detection by focusing every aspect of the program.
iv. Performance Analysis: Performance can be analyzed in better manner using visualization.
v. Race Condition Detection: Race conditions can be detected in better manner using visualization techniques for multithreaded programs.

2.4.1 Two Types of Visualization Systems [55]

i. Algorithm Animation: It animates the execution of an algorithm. Whole working of algorithm is animated and presented in this approach.
ii. Still Visualization: It displays the execution behavior of a program in one view. All events of the program are visualized all together.

2.4.2 Techniques for Visualizing Concurrency

i. Real Time Technique [56]: In this technique, concurrent programs are visualized during its execution time. This technique can depict status of thread execution at any time.

Advantages**:**

- This technique shows real time scenario of threads. It is a quick method of thread visualization.
- There is more comprehensibility due to clear visualization of execution steps.

Disadvantages:

- It is very hard to implement such scenario.
- This technique also requires updated thread status very frequently.

Tools implementing real time technique are Jive, JavaVis *etc*. [57].

ii. Postmortem Technique [58]: In this technique, the program behavior is visualized after its execution. In this technique, program execution traces are captured for visualizing it later on.

Advantage**:**

- All the significant data related to program execution are kept for future use.
- This approach is easier than real time approach because no run-time overhead is associated with it.

Disadvantages**:**

- Debugging is hard using only one example of program implementation.
- Due to unusual stopping of program execution, a large amount of incomplete data could be generated.

- Instrumentation of program could lead to change in original program behavior or any other error.

Tools implementing post-mortem approach are Xtango, Polka and Parade [58].

2.4.3 Problem with Multithreaded Program Visualization

Main problems associate with concurrent programs are [56]:

i. Probe Effect: Adding some extra information for better visualizing or understanding the program, changes behavior of the program. This change in behavior is known as probe effect.
ii. Non-repeatability: Any execution sequence of thread or any failure, is very tough to repeat in multithreaded programs.
iii. Non-determinism: Each execution of any multithreaded program results in diverse outputs. This non-deterministic nature of concurrent program produces many difficulties for its visualization.
iv. Execution Order: In a multithreaded program, execution order of operations is very hard to control, which makes testing difficult. Testing every branch for each input test case is also a challenge.

2.4.4 Tools for Multithreaded Program Visualization

Table 2.1 presents various tools for visualizing multithreaded programs, with their advantages and disadvantages.

Table 2.1 List of multithreaded program visualization tools

S. No.	Tools	Year	Advantages	Disadvantages
1	PARAVER [59]	1995	Visualization using traces.	Post-mortem approach.
2	VAMPIR[60]	1996	Performance analysis, graphical visualization, animation.	No deadlock detection.
3	JinSight[61]	1998	Open source, memory leaks and performance analysis.	No proper visualization.
4	Prism(Sun) [62]	2000	No separate debugger, easy to-use, flexible, and comprehensive set of tools.	Paid and no deadlock detection.

5	JaVis[63]	2002	Visualization based on UML sequence, collaboration diagrams extended for modelling thread synchronization and deadlocks.	Visualization based on other UML diagrams is not there.
6	RetroVue[64]	2002	Graphical debugger instead of replaying program execution.	Instrumentation to generate execution traces.
7	JAVAVIS[65]	2002	Main focus is to develop a tracing based analysis tool for deadlock detection.	Works for small program, no filters or automated analysis for errors.
8	Jacot[66]	2003	It uses the UML sequence, state chart and activity diagram paradigms.	Not much interactive.
9	Jink[57]	2004	Open source, interactive.	No performance analysis.
10	VisiVue[67] (VisiComp)	2004	Runtime visualization tool, creates animated diagrams.	Only for small applications of Java.
11	TraceVis[68]	2005	Interactive, anomaly detection and searching ability.	No support to visualize the memory behavior of the program.
12	CORE[69]	2010	Atomicity and sequence violation detection.	Performance issues.

Chapter 3
Gap Analysis and Problem Statement

This chapter introduces research gaps identified by analyzing previous work done in the area of multithreaded program and their bug detection. On the basis of gap analysis, problem statement of the research work is identified.

3.1 Gap Analysis

Based on the literature review of work related to bug detection techniques and tools for multithreaded program, following gaps have been identified:

- Till date, no approach has visualized deadlock due to wait-blockage and join-blockage [70].
- Effective visualization of large multithreaded programs is still a challenge due to management issues related to large number of elements [57].
- There is need to consider deadlock arising due to non-lexical lock primitives such as Lock.lock() and Lock.unlock() [70].

3.2 Problem Statement

After detailed review of literature related to anomalies and deadlock detection in multithreaded programs, this has been analyzed that an enhanced approach can be proposed for deadlock detection that uses a combination of lock graph, wait-notify dependence graph and thread dependence graph. Earlier, most of the approaches have considered only lock graphs for deadlock detection. Also, concurrent programs having wait expression and no corresponding notify expression, or having notify expression and no corresponding wait expression, are needed to be visualized for easy detection of wait-notify anomaly. Further, visualization of data and control dependencies helps in effective debugging of multithreaded programs.

Chapter 4
Methodology

This chapter gives a detailed view of proposed methodology by elaborating its every step with proposed algorithms. Architecture of the methodology is also explained in this section with five proposed algorithms.

4.1 Architecture of Bug Visualizer

As shown in Figure 4.1, the architecture of the proposed approach includes three components *i.e.* static parser module, matrix generator module, and graph generator with bug analyzer module. Bug Visualizer is developed using Java version 7.

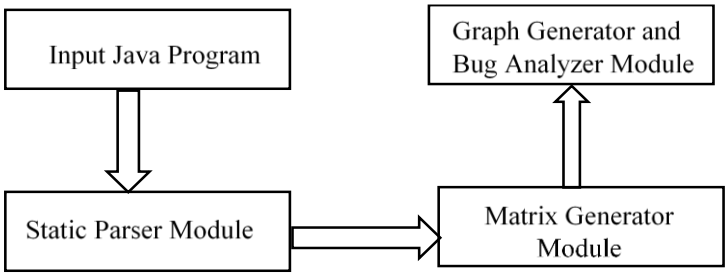


Figure 4.1 Architecture of proposed Bug Visualizer tool

Static analyzer module assigns line numbers to each statement of the program. Then it finds out various keywords, variables and corresponding dependencies between variables by statically analyzing and traversing the input Java program statement by statement.

Matrix generator module keeps earlier identified dependencies within adjacency matrices. Dependencies are stored according to their statement numbers. Various matrices are generated for different dependencies viz. control, data, lock, thread and wait-notify dependencies.

Graph generator module generates various graphs using the matrices generated by the matrix generator module. Data-control dependence graph is created using the data and control dependency matrices. Lock graph is created using lock dependency matrix. Wait-notify dependence graph is created using wait-notify dependency matrix. Thread dependence graph is created based on wait-notify thread dependencies. Finally, bug analyzer identifies bug(s) present in the Java program with the help of lock dependence

graph and thread dependence graph showing cycle for deadlock condition. It also shows wait-notify anomalies with wait-notify dependence graph.

4.2 Overview of Methodology

For visualizing thread blockage and wait-notify anomalies in a multithreaded Java program, the steps followed are shown in Figure 4.2.

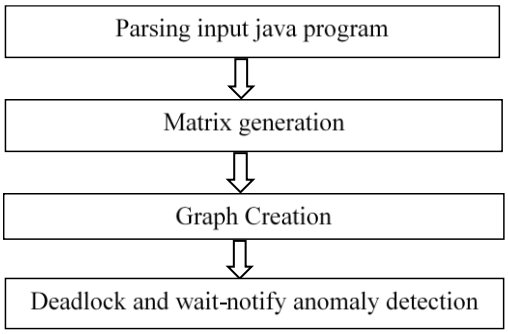


Figure 4.2 Overview of proposed methodology

i. Statically Parsing the Input Java Program

- Java program is taken as input file.
- Then whole file is numbered statement by statement.
- File is then traversed statement by statement for finding data, control, lock, wait-notify and thread dependencies.

ii. Matrix Generation

- Previously identified dependencies are mapped into an adjacency matrix corresponding to their line numbers.
- Different identification marks are stored for showing different dependencies in the matrices.

iii. Graph Generation

- Firstly, data dependence and control dependence matrixes are traversed for generating multithreaded program dependence graph.
- Lock dependence matrix is also traversed thoroughly for generating lock dependence graph.

- Then wait-notify dependence matrix is traversed for generating wait-notify dependence graph.
- Finally, thread dependence matrix, based on wait notify dependency, is traversed for generating thread dependence graph.

iv. Deadlock and Anomaly Detection

- Thread blockage is identified with the help of presence of cycle in lock dependence graph.
- Wait-notify anomaly is detected due to wait having no notify or notify having no wait dependency.
- Cycle in thread dependence graph shows deadlock due to cyclic wait-notify dependency among threads.

4.3 Proposed Algorithms

Five algorithms are proposed for the proposed tool Bug Visualizer as explained in this section. These algorithms can be categorized as follows:

4.3.1 Data-Control Dependence Graph

Matrices are generated by traversing input Java program statement by statement. Data dependence matrix is generated using proposed Algorithm 1. This algorithm gives 'data_matix' as output adjacency matrix showing data dependencies.

Algorithm 1: Data Dependence Matrix Generator

/* *variable_array* is used to store the name of variables, *define* array stores the line number of the variable definition, *data_matrix* is the adjacency matrix for storing data dependencies. *n* is total number of lines. */

Input: Java program file
Output: Data dependence adjacency matrix

1. Assign statement number 1,2,3,...,*n* to each statement of input Java program.
2. Initialize $i = 0$, *variable_array* = Φ, *define* = Φ and *data_matrix* = Φ.
3. Traverse the input Java file statement by statement.

4. For each statement *L* in input program

   do

   a. If variable *v* is declared, then

      Add variable *v* to *variable_array* at index *i*.

      Increment the index *i*.

   b. If *v* ∈ *variable_array* at index *i*, is assigned a value, then

      define[*i*] = *L*.

   c. If *v* ∈ *define* at index *i* is used at statement *L*, then

      data_matrix[*L*][*define*[*i*]] = true. /*variable usage statement *L* is dependent on the statement which defines the value of variable*/

   end

Control dependence matrix is generated using Algorithm 2. It gives *control_matrix* as output matrix. Then, *data_matrix* and *control_matrix* is given as input to proposed Algorithm 5 for generating data-control dependence graph. Data and control dependencies are visualized with different colors to differentiate between them.

Algorithm 2: Control Dependence Matrix Generator

/* *control_matrix* is the adjacency matrix for storing control dependencies. *n* is total count of statements in input program */

Input: Java program file

Output: Control dependence adjacency matrix

1. Assign line number 1,2,3,...,*n* to each statement of input Java program.
2. Initialize *control_matrix* = Φ.
3. Traverse the input Java file statement by statement.
4. For each statement *L* in input program

   do

   a. Search for control construct keywords. /*keywords such as ‘if’, ‘for’, ‘while’ *etc.*/

      i. Each statement s inside the keyword block is control dependent on starting statement of the block.

      ii. ‘else’ is control dependent on ‘if’ statement.

iii. Statements between ‘case’ and corresponding ‘break’ are dependent on case statement.

iv. Statements in ‘do-while’ block, depend on both ‘do’ and ‘while’ statements.

b. If any other block is started at statement *L*, then

/*each statement *s* inside this block is dependent on start statement of the block*/

control_matrix[*s*][*L*] = true.

end

5. Any nested statement is dependent on the innermost nested block statement for its control.

4.3.2 Lock-Acquisition Dependence Graph with Deadlock

Lock dependencies are captured in *lock_matrix* by traversing input Java program. Proposed Algorithm 3 is used for this purpose. Then lock acquisition graph is created using Algorithm 5 in which *lock_matrix* is given as input. This graph also detects thread blockage bug due to deadlock condition. Deadlock situation is suspected via presence of cycle in lock dependence graph.

Algorithm 3: Lock-Acquisition Dependence Matrix Generator

/* *lock_matrix* is the adjacency matrix used for storing lock acquisition dependencies. *Lock* array keeps variables which are locked. *Line* array keeps line number of locked variables. *n* is total number of statements in input program */

Input: Java program file

Output: Lock acquisition dependence adjacency matrix

1. Initialize $i = 0$, *lock_matrix*[][] = Φ.
2. For each run method in the input program

do

a. If ‘synchronized’ keyword is found for variable L_1, then /*lock is acquired for L_1 variable in r^{th} run method*/

$Lock_r[i] = L_1$. /* storing locked variable at i^{th} location of r^{th} run method in *Lock* array*/

$Line_r$ [i++] = s. /* storing line number of variable at i^{th} location of r^{th} run method in *Line* array*/

b. Each variable which is locked in one method is flow dependent on previously acquired lock in same method.

lock_matrix[$Line_r$[i]][$Line_r$[i-1]] = true ∀ i >= 1 and i <= n

/* lock acquiring statement is flow dependent on previous lock statement (in the sequence in which locks are acquired) in that method */

end

3. If $L_1 \in (Lock_u[i] \cap Lock_v[j])$, then /*variable L_1 is locked by thread u at index i and thread v at index j respectively*/

a. Add lock dependency between these two lock dependence statements such that,

if (i > j) then /* Statement which acquires lock later, is dependent on the statement which has acquired the lock previously*/

lock_matrix[$Line_u$[i]][$Line_v$[j]] = true.

else

lock_matrix[$Line_v$[j]][$Line_u$[i]] = true.

4.3.3 Wait-Notify and Thread Dependence Graph with Anomalies and Deadlock

Wait-notify and thread dependencies are captured by traversing input program statement by statement. *Thread* and *wait_notify_matrix* are generated as output adjacency matrices using Algorithm 4. Afterwards, *wait_notify_matrix* and *Thread* matrix are given as input to Algorithm 5 for graph generation. Wait-notify dependence graph not only shows wait-notify dependencies, however, it also shows wait-notify anomalies. Nodes, which are not connected to any other node in the graph, show wait-notify anomaly. Algorithm 4 is implemented by using different colors to show different anomalies. Thread blockage, due to wait-notify dependency, is identified with the help of thread dependence graph. Cycle among threads in this graph depicts deadlock suspicion in the program.

Algorithm 4: Wait-Notify and Thread Dependence Matrix Generator

/* *Wait* array is used to keep variables that are in waiting state. *Wline* array store line number of waiting variables. *Notify* array is used to store variables which are notified. *Nline* array is to store line number of notified variables. *Thread* matrix is used to store thread dependencies and *wait_notify_matrix* is used to store wait notify dependencies*/

Input: Java program file

Output: Wait-notify and thread dependence adjacency matrix

1. Search for 'wait', 'notify' and 'notifyAll' keywords in the input file.
2. If wait found for thread T_1 at line number L, then
   Add T_1 in wait list. Also keep its line number.
   Wait[j] = T_1.
   Wline[j++] = L.
3. If notify or notifyAll for T_1 is found at line number L, then
   Add T_1 in notify list. Also keep its line number.
   Notify[i] = T_1.
   Nline[i++] = L.
4. If $T_1 \in$ *Wait* at index j and $T_1 \in$ *Notify* at index i then
   wait_notify_matrix[*Wline*[j]][*Nline*[i]] = true.
5. If there is *Wait*[j] which is never notified, then
   wait_notify_matrix[*Wline*[j]][*0*] = true. /*wait without corresponding notify stored at 0^{th} column*/
6. If there is *Notify*[i] , for which no one is waiting, then
   wait_notify_matrix[*0*][*Nline*[i]] = true. /*notify without corresponding wait stored at 0^{th} row*/
7. If wait for resource R is in M thread and its notify part is in N thread, then
   Thread[M][N]=true. /*Showing M is dependent over N thread*/

Algorithm 5 generates each type of graph corresponding to the matrix given as input. *lock_matrix* is the input matrix for lock acquisition dependence graph. Presence of cycle in the graph, shows suspicion of deadlock. *wait_notify_matrix* is the input matrix for wait-notify dependence graph. Wait- Notify anomalies get clearly visible due to presence of unconnected nodes. *Thread* dependence matrix shows dependencies among threads due to wait-notify dependencies. Cycle in this graph shows suspicion of deadlock in the input program.

Algorithm 5: Graph Generator

/* *Matrix* is the adjacency matrix for which graph is to be drawn. For storing nodes, which are visited before, *visited* array is used. *visited* array also keeps the coordinates of visited node. */

Input: Adjacency matrix for which graph is to be drawn say *Matrix*[][]

Output: Directed graphs and bugs

1. Adjacency matrix is given as input.
2. Initialize $i = 0$, $j = 0$, *visited* = Φ.
3. Input matrix is traversed row by row for each cell.
4. If *Matrix*[*i*][*j*] = true, then /*true value means *i* is dependent on *j**/
   a. If ((*i* OR *j*) $\notin$ *visited*), then

      Draw corresponding node(s).

      Add node's coordinates in visited array.

   b. Draw arrow line from i^{th} node to j^{th} node using coordinates from visited array.
5. Use different colors to show arrow lines for different types of dependencies.

Chapter 5
Implementation

The proposed methodology is implemented in a prototype tool named as 'Bug Visualizer'. This tool is developed in Java version 7. This application identifies deadlock due to lock dependency and thread dependency due to wait-notify dependency. Wait-notify anomalies are also identified by this application. Snapshots of application are presented in this chapter.

5.1 Snapshots of Implementation

This section shows graphical user interface of 'Bug Visualizer' with various resultant graphs and bugs for the input Java program. Snapshots of the proposed approach are shown by categorizing them in following categories:

5.1.1 Interface for User Interaction

This application interface has been designed in Java7. As shown in the Figure 5.1, the main application consists of five sections.

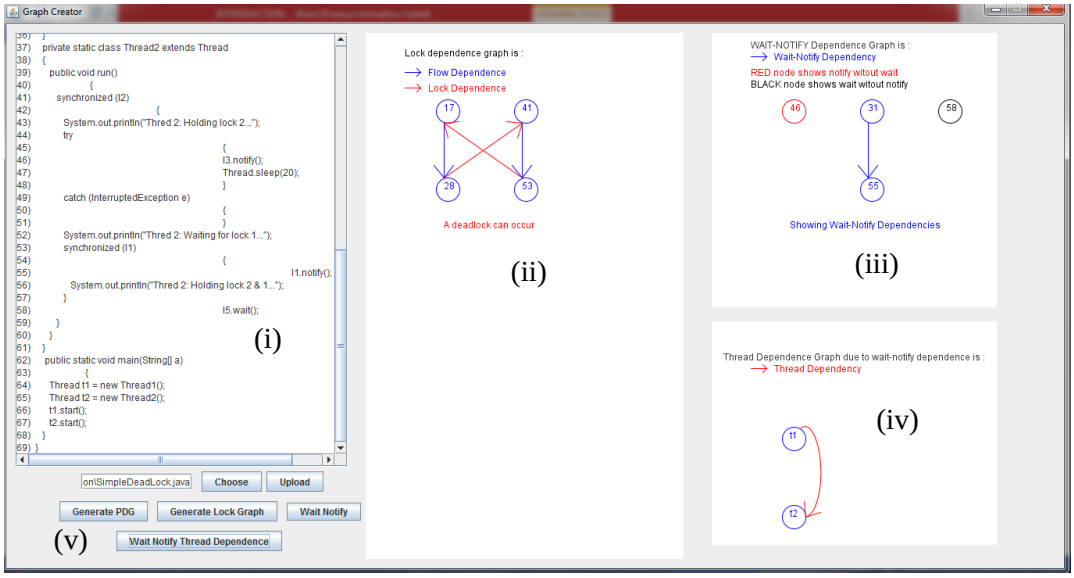


Figure 5.1 Graphical user interface of application

i. Text Section: This section contains one text area for showing the input file with the line numbers assigned to each statement.

ii. Lock Dependency Section: This is a panel attached to the main application which shows the lock dependence graph.

iii. Wait-Notify Section: This is another panel on the main application for showing the wait-notify dependency along with the anomalies like wait() without notify() or notify() without wait().

iv. Thread Dependency section: In this panel, thread dependence graph shows dependency among threads due to wait-notify dependencies.

v. Button Section: This section contains 6 buttons. On each of the button, appropriate event handling is applied. The details of buttons are as follows:

- Choose: This button helps the users in choosing the Java file to be given as input to the application.
- Upload: On clicking this button, file chosen by the user with their line numbers appears in the text area of the application.
- Generate PDG: On clicking this button, a new frame opens showing the program dependence graph of the selected file. Blue arrow represents control dependence and black arrow shows data dependence between the statements of the program chosen by the "Choose" button.
- Generate Lock Graph: When this button is clicked, the lock graph is shown in the lock acquisition section. Presence of a cycle in the lock graph shows the chances of deadlock in the input program.
- Wait Notify: When this button is clicked, the wait notify dependencies are visualized in the wait notify section. Wait and notify nodes without their counterparts are represented as isolated nodes.
- Wait Notify Thread Dependence: On clicking this button, thread dependence graph gets created showing thread dependency due to wait notify dependency. A cycle in this graph detects the suspicion of deadlock.

5.1.2 Selecting and Uploading the Input Java File

File is chosen on clicking "Choose" button as shown in Figure 5.2. Then selected file is uploaded in the text area by clicking "Upload" button. Uploaded file is adorned with line numbers associated with each statement. For referring any statement, corresponding line number is used in the graphs.

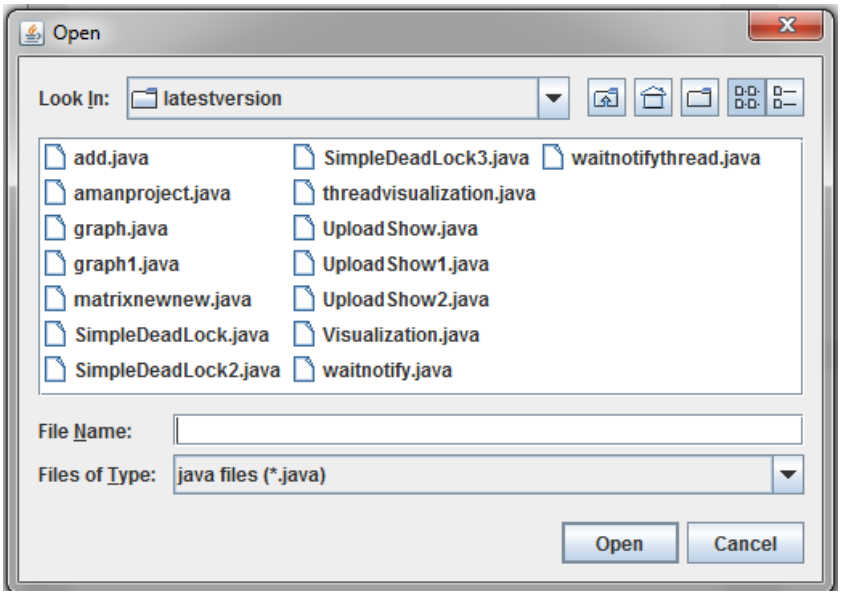


Figure 5.2 Selecting input java file

When "Upload" button is clicked, chosen file gets uploaded in the text area of the application with statement numbers as presented in the Figure 5.3. These statement numbers are further helpful while generating the different matrices for every dependency and drawing the corresponding graph.

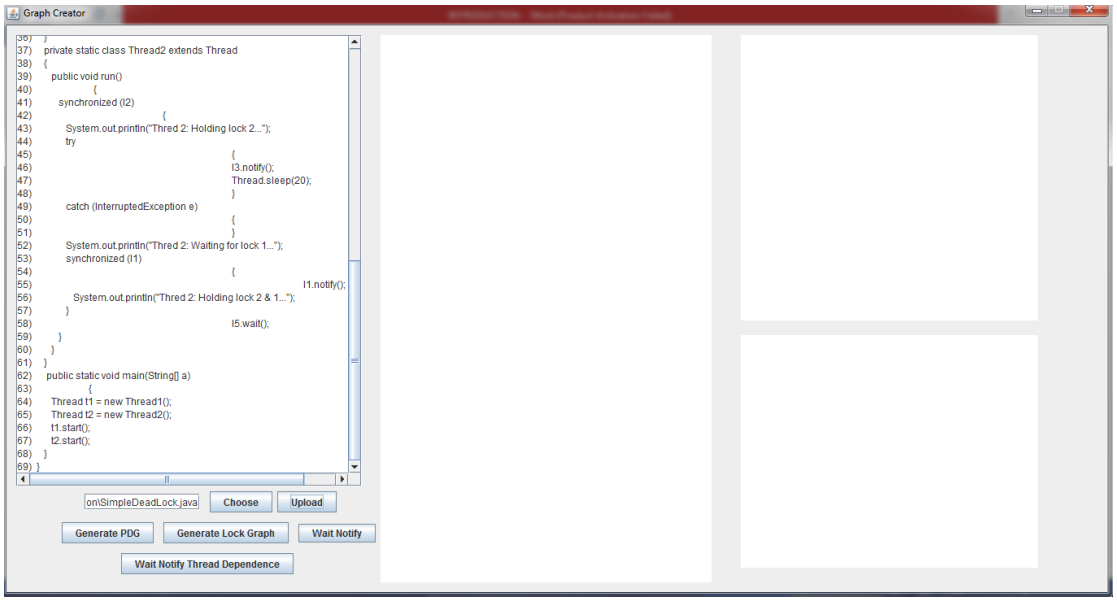


Figure 5.3 File uploaded in the application

Figure 5.4 shows the example Java program uploaded in Bug Visualizer. This example program is used for identifying multithreaded program bugs like deadlock and wait-notify anomalies.

```
1)  public class SimpleDeadLock extends Thread
2)  {
3)    public static Object l1 = new Object();
4)    public static Object l2 = new Object();
5)    private int index;
6)    private static class Thread1 extends Thread
7)    {
8)
9)                    //int c=a+b;
10)       public void run()
11)         {
12)
13)           int a=10;
14)           int b=15;
15)           int c=a+b;
16)           System.out.println("Value is " + c);
17)           synchronized (l1)
18)                   {
19)           System.out.println("Thred1: Holding lock 1...");
20)           try
21)                   {
22)                   Thread.sleep(20);
23)                   }
24)           catch (InterruptedException e)
25)                   {
26)                   }
27)           System.out.println("Thred 1: Waiting for lock 2...");
28)           synchronized (l2)
29)                   {
30)             System.out.println("Thred 2: Holding lock 1 & 2...");
31)                     l1.wait();
32)           }
33)
34)  }
```

```
35)       }
36)    }
37)    private static class Thread2 extends Thread
38)    {
39)      public void run()
40)                   {
41)        synchronized (l2)
42)                                 {
43)          System.out.println("Thred 2: Holding lock 2...");
44)          try
45)                   {
46)                   l3.notify();
47)                   Thread.sleep(20);
48)                   }
49)          catch (InterruptedException e)
50)                   {
51)                   }
52)          System.out.println("Thred 2: Waiting for lock 1...");
53)          synchronized (l1)
54)                   {
55)                   l1.notify();
56)             System.out.println("Thred 2: Holding lock 2 & 1...");
57)          }
58)                   l5.wait();
59)       }
60)     }
61)    }
62)    public static void main(String[] a)
63)                   {
64)      Thread t1 = new Thread1();
65)      Thread t2 = new Thread2();
66)      t1.start();
67)      t2.start();
68)  }
69)  }
```

Figure 5.4 Input Java example program

5.1.3 Generating Multithreaded Program Dependence Graph

Multithreaded program dependence graph (PDG) is generated on clicking “Generate PDG” button. At the back end, data and control dependence matrices are generated. These matrices are then traversed to generate the program dependence graph. In program dependence graph, different colors are used to distinguish data and control dependencies. Figure 5.5 shows data-control dependence graph generated for input example program. Here, data dependencies are shown with black color and control dependencies with blue color.

5.1.4 Generating Lock Dependence Graph

Lock dependence graph is created on clicking “Generate Lock Graph” button. Here deadlock is shown due to presence of cycle in lock dependence graph. Figure 5.6 shows lock dependence graph generated for input Java program. Figure 5.6 (a) shows the suspicion of deadlock for input program due to presence of cycle in lock dependence graph. Figure 5.6 (b) depicts the deadlock situation clearly.

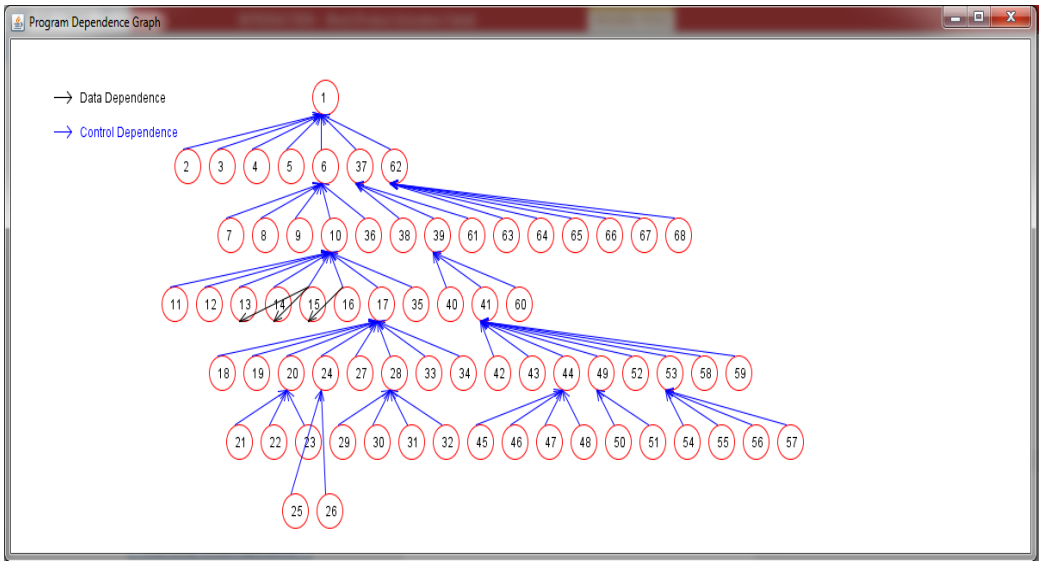
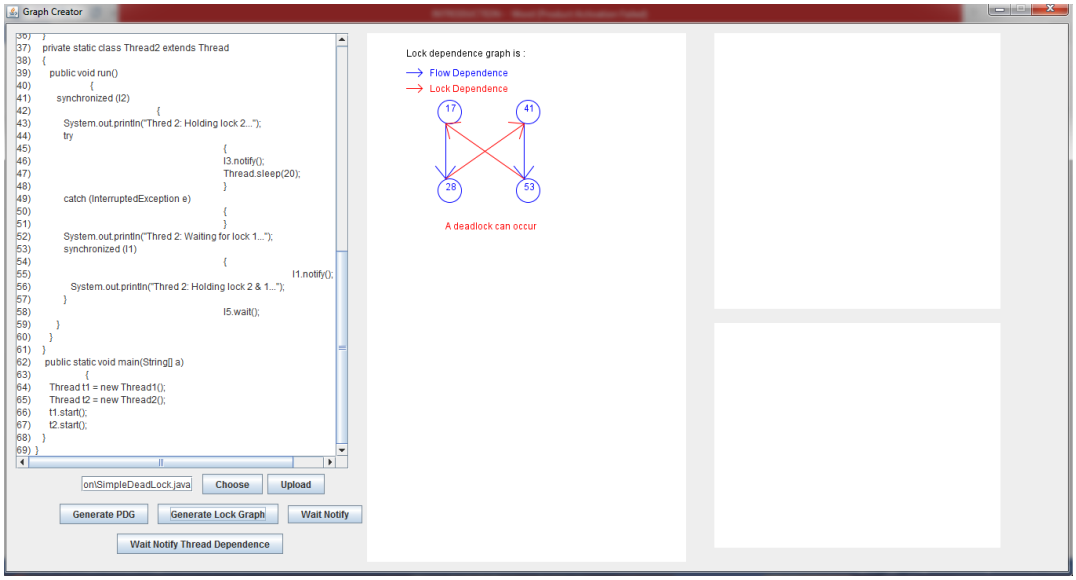
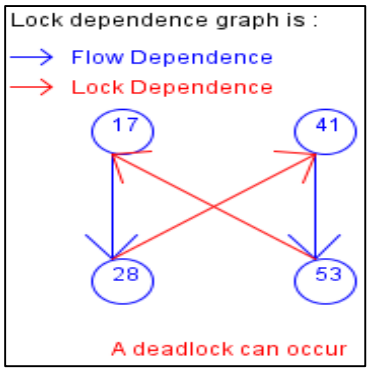


Figure 5.5 Data and control dependence graph



(a)



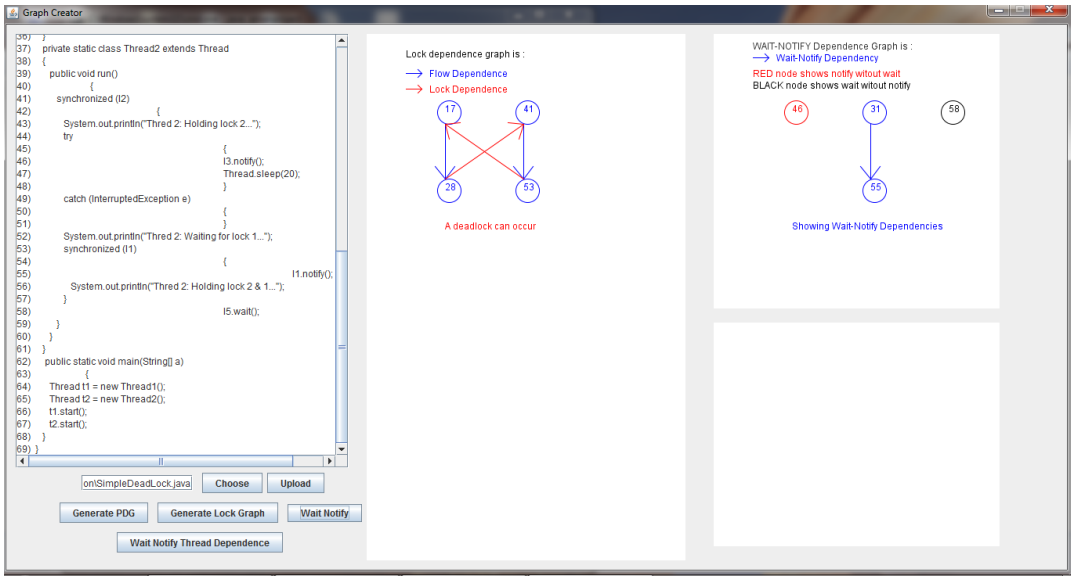
(b)

Figure 5.6 (a) Lock dependence graph. (b) Deadlock section of (a)

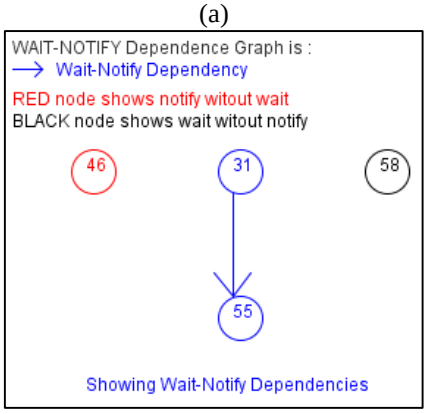
5.1.5 Generating Wait-Notify Dependence Graph

Wait-notify dependence graph is created on clicking “Wait Notify” button. This graph shows wait-notify dependencies and anomalies.

Figure 5.7 shows wait-notify dependence graph in Bug Visualizer. Wait notify dependencies are shown with blue color. Red node shows *notify without wait* anomaly *i.e.* notify node without corresponding wait node. Black node shows *wait without notify* anomaly *i.e.* wait node without corresponding notify node.



(a)

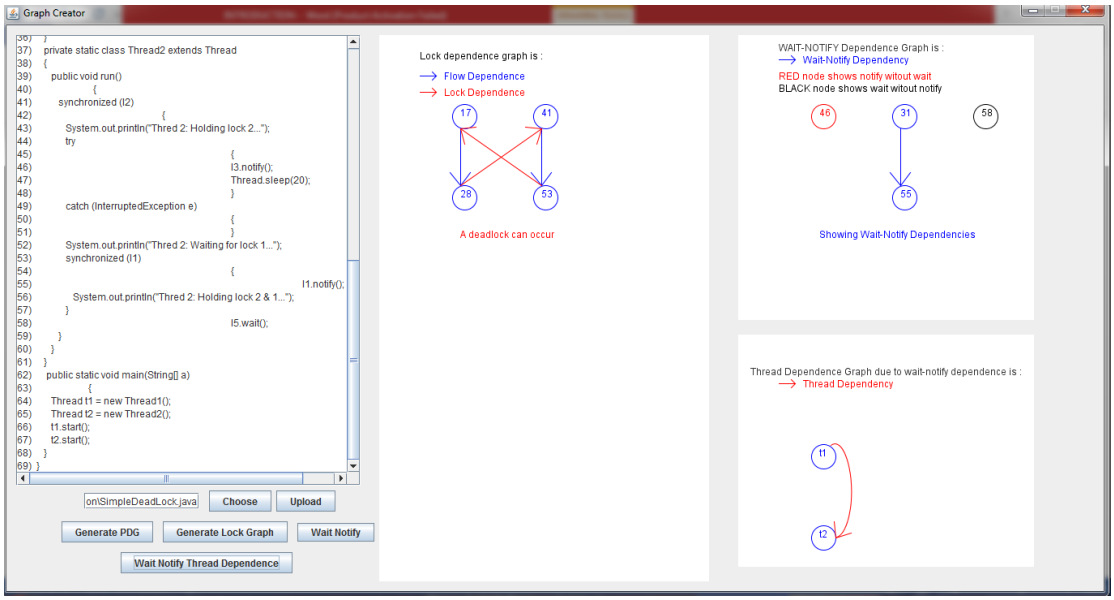


(b)

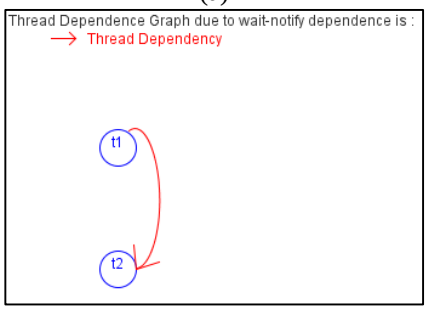
Figure 5.7 (a) Wait-notify dependence graph in Bug Visualizer. (b) Wait-notify section of (a)

5.1.6 Generating Thread Dependence Graph

Thread dependence graph is generated to show dependency among threads due to wait-notify dependency. Figure 5.8 shows thread dependence graph in Bug Visualizer.



(a)



(b)

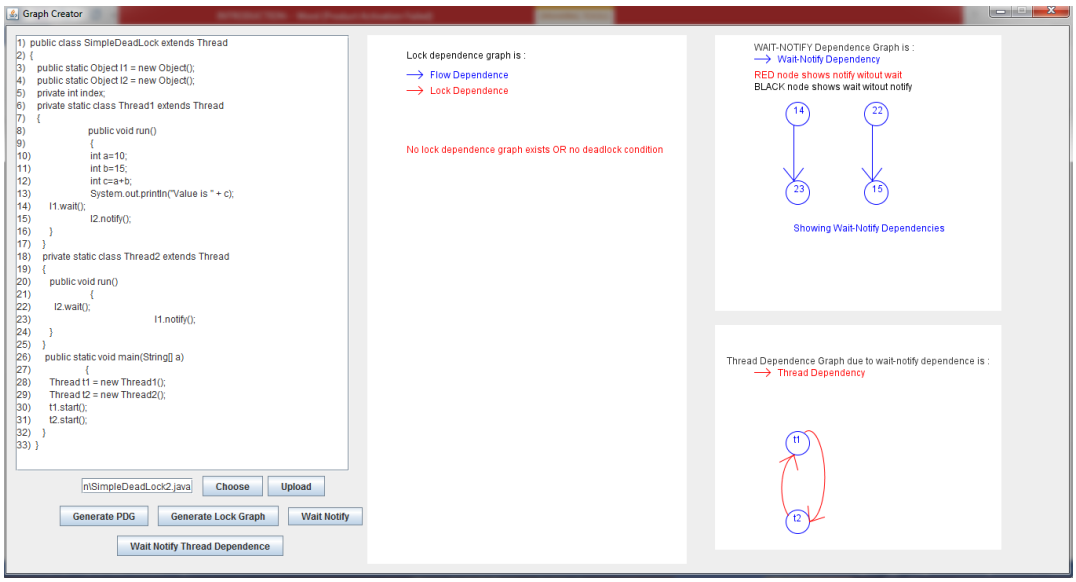
Figure 5.8 (a) Thread dependence graph in Bug Visualizer. (b) Thread dependence section of (a)

There can be a deadlock due to dependency among threads based on wait-notify dependency. Figure 5.9 shows an example program which is taken as input for identifying deadlock due to wait-notify dependency.

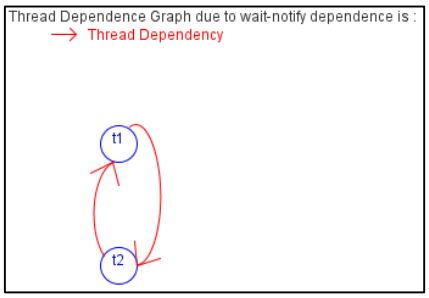
Figure 5.10 shows output of Bug Visualizer on giving Figure 5.9 example program as its input file. This graph shows wait-notify dependencies among threads in thread dependence graph. Cycle in thread dependence graph shows suspicion of deadlock due to wait-notify dependency as shown in Figure 5.10 (b). There is wait-notify dependency between thread t_1 and t_2 for the given input program. Dependency of wait node 14 on notify node 23, leads to dependency of thread t_1 on thread t_2. Similarly, dependency of wait node 22 on notify node 15, leads to dependency of thread t_2 on thread t_1. The mutual dependency between thread t_1 and thread t_2 leads to a cycle. Presence of cycle in this graph depicts the suspicion of deadlock in the system.

```
1)  public class SimpleDeadLock extends Thread
2)  {
3)    public static Object l1 = new Object();
4)    public static Object l2 = new Object();
5)    private int index;
6)    private static class Thread1 extends Thread
7)    {
8)                    public void run()
9)                     {
10)                    int a=10;
11)                    int b=15;
12)                    int c=a+b;
13)                    System.out.println("Value is " + c);
14)                    l1.wait();
15)                    l2.notify();
16)      }
17)    }
18)    private static class Thread2 extends Thread
19)    {
20)      public void run()
21)                    {
22)                  l2.wait();
23)                  l1.notify();
24)      }
25)    }
26)     public static void main(String[] a)
27)                    {
28)      Thread t1 = new Thread1();
29)      Thread t2 = new Thread2();
30)      t1.start();
31)      t2.start();
32)    }
33)  }
```

Figure 5.9 Java example program



(a)



(b)

Figure 5.10 (a) Thread dependence graph in Bug Visualizer. (b) Deadlock cycle in thread dependence graph

5.2 Results and Discussion

This section compares and discusses the results of proposed methodology with earlier proposed techniques. Proposed tool identifies deadlock due to lock acquisition and wait-notify dependencies. This tool also identifies wait-notify anomalies in multithreaded program. To debug multithreaded programs effectively, data-control dependence graph is created by the proposed tool Bug Visualizer.

Table 5.1 shows comparison of proposed methodology with earlier proposed tools. Also Bug Visualizer can detect and visualize wait-notify anomalies which were not visualized before by any other tool.

Table 5.1 Comparison of Bug Visualizer with JCAT and RacerX

S. No.	Techniques/Tools	Comparison
1	Java Concurrency Analysis Tool(JCAT) [71]	JCAT does not consider data and control flow analysis. However, Bug Visualizer shows both of these dependencies.
2	RacerX (Deadlock and race detection tool) [72]	RacerX can detect deadlock only due to lock acquisition however, Bug Visualizer can also detect deadlock due to cyclic wait-notify dependencies.

Chapter 6
Conclusion and Future Scope

This chapter concludes the proposed work and lists some points for further consideration as future work.

6.1 Conclusion

This thesis work detects deadlock and wait-notify anomalies by visualizing the multithreaded programs with the help of dependence graphs. This work is different from other approaches because of the following points:

- Proposed tool Bug Visualizer, presents a user interface for visualizing deadlock and wait-notify anomaly in multithreaded programs.
- Deadlock due to lock acquisition and wait-notify thread dependencies are identified using lock dependence and wait-notify thread dependence graphs respectively.
- Data and control dependence graphs are also shown to aid debugging of input Java program.
- This technique detects wait-notify anomalies with the help of wait-notify dependence graph for better fault localization.

6.2 Future Scope

The proposed approach detects deadlock and wait-notify anomaly by generating various graphs, however there are some points that can be elaborated further.

- This work can be further extended for detecting thread blockage due to join dependencies among threads by constructing join dependence graph.
- Further, the proposed tool Bug Visualizer can be enhanced to detect other multithreaded bugs like starvation, data race *etc*.

References

[1] "Concurrency vs parallelism - What is the difference?." Internet: http://stackoverflow.com/questions/1050222/concurrency-vs-parallelism-what-is-the-difference [May 2, 2014].

[2] P.B. Hansen, "Concurrent programming concepts," *ACM Computing Surveys*, ACM, vol. 5, no. 4, pp. 223-245, 1973.

[3] A. Silberschatz, P. B. Galvin and G. Gagne, *Operating System Concepts*, 7th ed. John Wiley & Sons Inc., 2005.

[4] "OS." Internet: http://imraan-prrec.blogspot.in/p/os.html [May 5, 2014].

[5] "General: What are semaphores?." Internet: http://www.keil.com/support/docs/466.htm, 01 Mar. 2007 [Nov. 25, 2013].

[6] Payne. "Is there a mutex in Java?." Internet: http://stackoverflow.com/questions/5291041/is-there-a-mutex-in-Java, 11 Nov. 2012 [Dec. 5, 2013].

[7] "What is livelock." Internet: http://www.guruzon.com/1/threading/multithreading-concepts/what-is-live-lock, 01 Jun 2013 [Dec. 27, 2013].

[8] K.C. Tai, "Definitions and Detection of Deadlock, Livelock, and Starvation in Concurrent Programs," *Proc. International Conference on Parallel Processing*, vol. 2, pp. 69-72, IEEE, Aug. 1994.

[9] "Starvation." Internet: https://www.cs.auckland.ac.nz/~alan/courses/os/book/6.Mana.13.starvation.pdf [May 4, 2014].

[10] "Concurrency: Deadlock and starvation." Internet: https://www.google.co.in/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&ved=0CDMQFjAC&url=http%3A%2F%2Fwww.cs.wustl.edu%2F~fredk%2FCourses%2Fcs422%2Fsp03%2FLectures%2Fdeadlock1.ppt&ei=5OZ-U7_-N8KfugSy1YLgCw&usg=AFQjCNFtitQ4I53RoSPR1I0F8mg-jRg-1g&bvm=bv.67720277,d.c2E&cad=rja [May 6, 2014].

[11] "Input output." Internet: https://www.google.co.in/url?sa=t&rct=j&q=&esrc=s&source=web&cd=7&cad=rja&uact=8&ved=0CFIQFjAG&url=http%3A%2F%2Fwww.cse.scu.edu%2F~

tschwarz%2FCOEN225_08%2FLectures%2FInputOutput.ppt&ei=Q9pgU8_CAdGSuASZ9oGQBA&usg=AFQjCNFMbEItFziWz_tUBRYZZ5vVlMAQhg&sig2=7urmorbaSfHNg25hTfgFpg&bvm=bv.65636070,d.c2E [May 4, 2014].

[12] "What is a race condition." Internet: http://stackoverflow.com/questions/34510/what-is-a-race-condition, 16 May 2013 [May 5, 2014].

[13] S.K. Card, J.D. Mackinlay and B. Shneiderman, *Readings in Information Visualization: Using Vision to Think*. San Francisco, CA: Morgan Kaufmann Publishers Inc., 1999.

[14] A.P. Mathur, *Foundations of Software Testing*, India: Pearson Education, 2010.

[15] G. Booch, J. Rumbaugh and I. Jacobson, *Unified Modeling Language User Guide*, Harlow:Addison Wesley,1998.

[16] P. Stevens, "UML and concurrency," *Abstract State Machines*, vol. 2589, pp. 151-166, Springer, 2003.

[17] N.D. Francesco and G. Vaglini, "Modular verification of correctness properties in environment for concurrent systems specifications: Deadlock case," *Information and Software Technology*, vol. 32, no. 2, pp. 133-148, Mar. 1990.

[18] W. Araujo, L.C. Briand and Y. Labiche, "On the effectiveness of contracts as test oracles in the detection and diagnosis of race conditions and deadlocks in concurrent object-oriented software," *2011 International Symposium on Empirical Software Engineering and Measurement*, pp. 10-19, 22-23 Sept. 2011.

[19] K.I. Pun, M. Steffen and V. Stolz, "Deadlock checking by a behavioral effect system for lock handling," *The Journal of Logic and Algebraic Programming*, vol. 81, no. 3, pp. 331-354, Apr. 2012.

[20] M. Pradel and T.R. Gross, "Fully automatic and precise detection of thread safety violations," *Proc. 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, vol. 47, no. 6, pp. 521-530, June 2012.

[21] J.E. Gottschlich, G.A. Pokam, C.L. Pereira and Y. Wu, "Concurrent predicates: A debugging technique for every parallel programmer," *Proc. 22nd International Conference on Parallel Architectures and Compilation Techniques*, pp. 331-340, 2013.

[22] D. Feitelson, "Deadlock detection without wait-for graphs," *Parallel Computing*, vol. 17, pp. 1377-1383, 1991.

[23] Y. Cai and W.K. Chan, "MagicFuzzer: Scalable deadlock detection for large-scale applications," *Proc. 2012 International Conference on Software Engineering*, pp. 606-616, 2012.

[24] S. Chaki, E. Clarke, J. Ouaknine and N. Sharygina, "Automated, compositional and iterative deadlock detection," *Proc. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design*, pp. 201-210, 23-25 June 2004.

[25] Z. Xu and O. de Vel, "Petri net modelling of occam programs for detecting indeterminacy, non-termination and deadlock anomalies," *Proc. Fourth International Workshop on Petri Nets and Performance Models*, pp. 116-124, 2-5 Dec. 1991.

[26] K.A. D'Souza, "A control model for detecting deadlocks in an automated machining cell," *Computers & Industrial Engineering*, vol. 26, no. 1, pp. 133-139, Jan. 1994.

[27] Z. Li, M. Uzam and M. Zhou, "Deadlock control of concurrent manufacturing processes sharing finite resources," *The International Journal of Advanced Manufacturing Technology*, vol. 38, no. 7-8, pp. 787-800, Sept. 2008.

[28] H. Liao, J. Stanley, Y. Wang, S. Lafortune, S. Reveliotis and S. Mahlke, "Deadlock-avoidance control of multithreaded software: An efficient siphon-based algorithm for gadara petri nets," *Proc. 50th IEEE Conference on Decision and Control and European Control Conference*, pp. 1142-1148, 2011.

[29] L. Fajstrup, E. Goubault and M. Raußen, "Detecting deadlocks in concurrent systems," *Proc. 9th International Conference on Concurrency Theory*, vol. 1466, pp. 332-347, 8-11 Sept. 1998.

[30] N. Vasudevan and S.A. Edwards, "Static deadlock detection for the shim concurrent language," *Proc. 6th ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, pp. 49-58, June 2008.

[31] E. Cheung, C. Xi, H. Harry, D. Abhijit, S.V. Alberto and W. Yosinori, "Runtime Deadlock analysis for system level design," *Design Automation for Embedded Systems*, vol. 13, no. 4, pp. 287-310, Dec. 2009.

[32] R. Agarwal, S. Bensalem, E. Farchi, K. Havelund, Y. Nir-Buchbinder, S. Stoller, S. Ur and L. Wang, "Detection of deadlock potentials in multithreaded programs,"

IBM Journal of Research and Development, vol. 54, no. 5, pp. 3:1-3:15, Sept.-Oct. 2010.

[33] S. Park, "Debugging non-deadlock concurrency bugs," *Proc. 2013 International Symposium on Software Testing and Analysis*, pp. 358-361, 2013.

[34] S. Gradara, A. Santone and M.L. Villani, "Using heuristic search for finding deadlocks in concurrent systems," *Information and Computation*, vol. 202, no. 2, pp. 191-226, Nov. 2005.

[35] S. Gradara, A. Santone and M.L. Villani, "DELFIN+: An efficient deadlock detection tool for CCS processes," *Journal of Computer and System Sciences*, vol. 72, no. 8, pp. 1397-1412, Dec. 2006.

[36] S. Davidson, I. Lee and V.F. Wolfe, "Deadlock prevention in concurrent real-time systems," *Real-Time Systems*, vol. 5, no. 4, pp. 305-318, Oct. 1993.

[37] R. Cypher and E. Leu, "Efficient race detection for message-passing programs with nonblocking sends and receives," *Proc. Seventh IEEE Symposium on Parallel and Distributed Processing*, pp. 534-541, 25-28 Oct. 1995.

[38] K. Audenaert and L. Levrouw, "Space efficient data race detection for parallel programs with series-parallel task graphs," *Proc. Euromicro Workshop on Parallel and Distributed Processing*, pp. 508-515, 25-27 Jan. 1995.

[39] V. Kahlon, Y. Yang, S. Sankaranarayanan and A. Gupta, "Fast and accurate static data-race detection for concurrent programs," *Proc. 19th International Conference on Computer Aided Verification*, vol. 4590, pp. 226-239, 2007.

[40] S. Grabner, D. Kranzlmuller and J. Volkert, "Debugging of concurrent processes," *Proc. Euromicro Workshop on Parallel and Distributed Processing*, pp. 547-554, 25-27 Jan. 1995.

[41] M.-Y. Park, S.Y. Kim and H.-R. Park, "Visualization of affect-relations of message races for debugging MPI programs," *Proc. IEEE International Conference on Granular Computing*, pp.745-745, 2-4 Nov. 2007.

[42] Y. Chen, Y.-H. Lee, W.E. Wong and D. Guo, "A race condition graph for concurrent program behavior," *Proc. 3rd International Conference on Intelligent System and Knowledge Engineering*, vol. 1, pp. 662-667, 17-19 Nov. 2008.

[43] G.M. Tchamgoue, L. Gan, O.-K. Ha, S.-W. Yang and Y.-K. Jun, "Visualizing concurrency faults in ARNIC-653 real-time applications," *Proc. IEEE/AIAA 31st*

Digital Avionics Systems Conference (DASC), pp. 9E6-1 - 9E6-9, 14-18 Oct. 2012.

[44] V. Kahlon, S. Sankaranarayanan and A. Gupta, "Static analysis for concurrent programs with applications to data race detection," *International Journal on Software Tools for Technology Transfer*, vol. 15, no. 4, pp. 321-336, 2013.

[45] E. Yoshida and H. Kakugawa, "A learning system for the problem of mutual exclusion in multithreaded programming," *Proc. IEEE International Conference on Advanced Learning Technologies*, pp. 2-6, 30 Aug.-Sept. 2004.

[46] E. Bodden and K. Havelund, "Aspect-oriented race detection in Java," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 509-527, 2010.

[47] M. Shousha, L.C. Briand and Y. Labiche, "A UML/MARTE model analysis method for detection of data races in concurrent systems," *Proc. 12th International Conference on Model Driven Engineering Languages and Systems*, vol. 5795, pp. 47-61, 2009.

[48] J. Devietti, B.P. Wood, K. Strauss, L. Ceze, D. Grossman and S. Qadeer, "RADISH: Always-on sound and complete race detection in software and hardware," *39th Annual International Symposium on Computer Architecture* , pp. 201-212, 9-13 June 2012.

[49] Y. Hafeez, M.A. Abbas and G. Mustafa, "Techniques for data-race detection and fault tolerance: A survey," *Proc. 2012 International Conference on Computer & Information Science*, vol. 2, pp. 958-961, 12-14 June 2012.

[50] D. Lugato, C. Bigot, Y. Valot, J.-P. Gallois, S. Gérard and F. Terrier, "Validation and automatic test generation on UML models: The AGATHA approach," *International Journal on Software Tools for Technology Transfer*, vol. 5, no. 2-3, pp. 124-139, 2004.

[51] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R.A. Popa and Y. Zhou, "MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 103-116, Dec. 2007.

[52] S. Lu, S. Park, E. Seo and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 2, pp. 329-339, Mar. 2008.

[53] E. Trainin, N.-B. Yarden, T.-B. Rachel, Z. Aviad, U. Shmuel and E. Farchi, "Forcing small models of conditions on program interleaving for detection of concurrent bugs," *Proc. 7th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, pp. 7:1-7:6, 2009.

[54] W.D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides and J. Yang, "Visualizing the execution of Java programs," *Software Visualization*, vol. 2269, pp. 151-162, Springer, 2002.

[55] C. Artho, K. Havelund and S. Honiden, "Visualization of concurrent program executions," *Proc. 31st Annual International Computer Software and Applications Conference*, vol. 2, pp. 541-546, IEEE, 24-27 July 2007.

[56] X. Wu, J. Wei and X. Wang, "Debug concurrent programs with visualization and inference of event structure," *Proc. 2012 19th Asia-Pacific Software Engineering Conference*, vol. 1, pp. 683-692, IEEE, 2012.

[57] R. Capuano, "Interactive visualization of concurrent programs," *Proc. 19th International Conference on Automated Software Engineering*, pp.418-421, IEEE, 20-24 Sept. 2004.

[58] M. Bedy, S. Carr, X. Huang and C. Shene, "A visualization system for multithreaded programming," *ACM SIGCSE Bulletin*, vol. 32, no. 1, pp. 1-5, Mar. 2000.

[59] V. Pillet, J. Labarta, T. Cortes and S. Girona, "Paraver: A tool to visualize and analyze parallel code," *Proc. WoTUG-18: Transputer and Occam Development*, vol. 44, pp. 17-31, 1995.

[60] W.E. Nagel, A. Arnold, M. Weber, H.-C. Hoppe and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," *Supercomputer*, vol. 12, no. 1, pp. 69-80, 1996.

[61] W.M. Pauw and J. Vlissides, "Visualizing object-oriented programs with jinsight," *Object-Oriented Technology: ECOOP'98 Workshop Reader*, vol. 1543, pp. 541-542, Springer, 1998.

[62] "Prism™ 6.1 User's Guide." Internet: http://www.unics.uni-hannover.de/zzzzwg1/SunWS6/Prism-61-UG.pdf , Mar. 2000 [Jan. 15, 2014].

[63] K. Mehner, "JaVis: A UML-based visualization and debugging environment for concurrent Java programs," *Software Visualization*, vol. 2269, pp. 163-175, Springer, 2002.

[64] J. Callaway, "Visualization of threads in a running Java program," Master's thesis, University of California, June 2002.

[65] R. Oechsle and T. Schmitt, "JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java debug interface (JDI)," *Software Visualization*, vol. 2269, pp. 176-190, 2002.

[66] H. Leroux, C. Mingins and A. Réquilé, "JACOT: A UML-based tool for the runtime-inspection of concurrent Java programs," *Proc. 2nd International Symposium on Principles and Practices of Programming in Java*, vol. 42, 16-18 June 2003.

[67] J. Joshi, B. Cleary and C. Exton, "Application of helix cone tree visualizations to dynamic call graph illustration," *Proc. Third Program Visualization Workshop*, pp. 68-75, 2004.

[68] J. Roberts, "TraceVis: An execution trace visualization tool," *Proc. MoBS,* 2005.

[69] D. Jayasinghe and P. Xiong. "CORE: Visualization tool for fault localization in concurrent programs." Internet: http://www.cc.gatech.edu/grads/i/ijayasin/resources/core_falcon.pdf, 2010 [May 8, 2014].

[70] T. Shimomura and K. Ikeda, "Waiting blocked-tree type deadlock detection," *Proc. Science and Information Conference (SAI), 2013* , pp. 45-50, 7-9 Oct. 2013.

[71] C. Demartini, R. Iosif and R. Sisto, "A deadlock detection tool for concurrent Java programs," *Softw: Pract. Exper*, vol. 29, no. 7, pp. 577–603, June 1999.

[72] D. Engler and K. Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 237-252, Oct. 2003.

List of Publications

Accepted and Presented:

[1] Aman Jyoti and Vinay Arora, "Visualization of Deadlock and Wait-Notify Anomaly in Multithreaded Programs," Presented at IEEE International Conference on Advanced Communication Control and Computing Technologies (ICACCCT), Tamilnadu, India, 2014.

[2] Aman Jyoti and Vinay Arora, "Debugging and visualization techniques for multithreaded programs: A survey," Presented at IEEE International Conference on Recent Advances and Innovations in Engineering (ICRAIE), Jaipur, India, 2014.

Accepted:

[1] Vipin Verma, Aman Jyoti and Vinay Arora, "Deadlock detection with dependence graph due to wait-notify dependency in multithreaded programs," Second International Conference on Emerging Research in Computing, Information, Communication and Applications (ERCICA), Bangalore, India, Elsevier, 2014.