International Journal of Engineering & Technology, 7 (3.4) (2018) 6-12

International Journal of Engineering & Technology

Website: www.sciencepubco.com/index.php/IJET

Research paper

Concurrent Bug Detection Using Invariant Analysis

Bidush Kumar Sahoo[1]*, Mitrabinda Ray[2]

[1]*Research Scholar, Department of Computer Science and Engineering, ITER, SOA University*
[2]*Associate Professor, Department of Computer Science and Engineering, ITER, SOA University*
**Corresponding author E-mail: bidush.sahoo@gmail.com*

Abstract

In concurrent programs, bug detection is a tedious job due to non-determinism and multiple thread control. The bug detection is done by checking the interleaving of threads which is not available in operational phases. So, static analysis is one of the preferred approaches for detection of concurrent bug. Invariant based testing technique is one approach of static analysis used for detecting the concurrent bugs. In this paper, we discuss an invariant based testing approach using three steps: (i) the invariants of a given concurrent program are generated using Daikon tool. (ii) The bad invariants are removed by using the static call graph of the source code, where the static call graph is generated by the javacg tool. (iii) The reduced invariant set is obtained by eliminating the bad and redundant invariants which is used for testcase generation. Using the reduced invariant set, we generate the testcases that are used to find the various concurrent bugs such as Deadlock, Atomicity violation and Bad composition. We conducted an experiment on a well-known concurrent program called the Dining Philosopher Problem. The experimental results show that, the testcases obtained from the reduced invariant set is able to detect more types and no. of concurrent bugs than the existing approach on invariant based testing.

Keywords: *Dynamic Invariant, Concurrent Bugs, Static Call Graph, Redundant Invariant, Invariant Analysis*.

1. Introduction

In order to fulfill hardware resource requirements, the current technology is moving towards concurrent system. There are various advantages of concurrent system like utilization of less memory and resources in comparison with sequential system. But as the concurrent system depends upon the thread interleaving, so, there is a chance of error while data or resource sharing. The bug detection in concurrent program is difficult because it depends on the occurrence of certain bad interleaving of threads [18]. Data race bug is a common problem in concurrent programs [13]. These bugs occur when the memory access rules are violated in read-write pattern. Various concurrent bugs like deadlock, atomic violation etc occurs while running a concurrent program having thread interaction issues [18]. Due to this the tester faces a huge challenge while detection of concurrent bugs in the source code. In concurrent programs, through static analysis it is possible to detect the concurrent bugs.

One of the static analysis approaches is *Invariant Analysis*. Invariants are the conditions which remain true during the execution of a program. It is a property which is able to capture the specific program behaviors through a specific program point. These are very essential for the verification tools for checking the loops and the function calls. The invariants for a program can be generated by Daikon tool [3]. For detection of concurrent bugs [1], the function call graph known as static call graph [21] is used for finding all possible fault positions. The static call graph helps in providing all possible routes of program execution. The static call graph of the source code is generated by tools such as javacg [21], Doxygen [16] etc.

Our objective is to find out the concurrent bugs using the program invariants of the concurrent program and the static call graph of the source code. Daikon tool is used to obtain the program invariants. We propose an algorithm to reduce the redundant program invariants in order to get the effective testcases. The javacg tool is utilized for finding the static call graph in order to find all possible call traces. Compared with the previous approaches this technique provides several advantages. First, it detects more concurrent bugs than previous approaches. Second, it checks the negative failure invariants which are actually positive but treated as negative Invariants. Third, the numbers of test cases are reduced.

The section description is provided as follows. Section 2 describes the related works regarding different invariant based testing works. Section 3 describes the background study of different bug types and invariants. Section 4 describes the invariant based bug detection and different methods along with the implementation using static call graph and Daikon tool. Last but not the least in section 5, it concludes the paper with some future work.

2. Related Work

In the field of invariant analysis, both static analysis and dynamic analysis for invariant generation is discussed by various authors. Using static analysis, the authors in [2] [3] [9] generate invariants for detecting concurrent bugs. Some authors in [5] [6] use dynamic analysis for invariant generation to detect various concurrent bugs. Using bit-mask invariants and range invariants, different authors in [4] tried to find out concurrency bugs. Some authors use different tools like UDON [6], IODINE [5] for extracting the dynamic invariants and DAIKON [3], HOUDINI [9] for static invariants. Some of the works are discussed below.

Wang et al [2], describes about the concurrency bug detection using program invariant method using the function call graph of the components. Then it uses a reduction technique for the invariants to reduce the bad and redundant invariants. Function call graph and invariants are used to find the suspicious positions of the concurrency bugs. It obtains the program invariants through Daikon tool. Then by using the interaction among the components of the program/source code it finds the different concurrency bugs. This method has some merits like detection of various types of bugs like deadlock, atomicity violation and bad composition etc. Comparing to this method, our approach can detect many types of concurrent bugs.

According to **Ernst et al** [3], the program invariants are generally used for identifying the program properties used for modification of code. It uses the dynamic techniques for getting the invariants from trace files. In this work, it gives idea on various techniques of finding dynamic invariants. It provides the report of finding the predefined invariants. It also checks for scalability issues like invariant detection and accuracy of test cases. However, this approach is able to generate less number of effective testcases compared to our approach.

Alberto [4] has used the error detection technique on the embedded systems. He has checked the performance of the technique by dynamic invariant method. In his work he has used two types of invariants like bit-mask and range invariants. In his work, he has checked 60% - 80% of the program, coming under the scanner for fault detection. It has found that some of the errors present within the code were going unnoticed. Bit-mask invariants are fast learners in comparison to range invariants but the range invariants can find more number of false positive invariants. Comparing with our approach, it fails to detect the false negative invariants.

According to **Hangal et al** [5], the dynamic invariant detection can be inferred in hardware designs. IODINE, a tool for dynamic invariant generation is used for extracting the dynamic invariants in case of hardware design. It is also used for property checking like identification, validations etc. Moreover, this approach is not suitable for the software designing process unlike our approach.

Kusano et al [6], has used a novel method for generating the likely invariants from a multithreaded program. This method is an enhancement of dynamic invariant generator, which customizes the invariant inference engine. They have compared their unique approach with the Daikon tool for multithreaded programs. In the new tool UDON they have incorporated the concept of capturing the relation among the shared variables, which are called transition invariants. Using this approach, it shows the Daikon tool is less effective in terms of number of true invariant generation. It also improves by leveraging the selective interleaving exploration strategy. Through this approach we get the idea of relevance of transition invariants. It also demonstrates the efficiency and effectiveness through some multithreaded programs. Moreover, some dynamic invariants that are generated may not represent the concurrency related behavior.

Marina et al, in [7] described class invariants as a condition on the shared memory. In this paper the validity of class invariants in a multi-threaded system is defined. This approach supports the breaking of invariants which is invisible to other threads. Here, the class invariants maintain a token which indicates the inspection of class invariants. The token can be splited or combined according to the situation. Splitting the invariant is done through executing an unpack statement. The invariant is re-established when the token which is required to inspect the invariant is available again. The behavior is designed by the pack statement. So, the unpacked statement which is present in a thread is free to do what it wants with the class invariants. It happens because in parallel the class invariants cannot be observed in a thread. For checking the class invariants whether it is broken or not, other threads aren't allowed to obtain the permission. However, this approach is having the limitation of considering only the permission based programming languages which is not a barrier in our approach.

According to **Ellsworth** [8], the dynamic invariant detection is done on all observed runs of a program at a fixed point in execution. The dynamic invariant detection is helpful for better understanding of the code. The reduction of invariants is due to the intra-procedural dataflow. Here the author contributes a prototype implementation of a new invariant filtration technique. This technique results in the integration of various analysis tools. Moreover, the invariant reduction is dependent on the data flow in between methods, which never happens in our approach to suppress the invariants within a function.

According to **Nimmer et al** [9], static checking is not able to verify the errors with required specifications so it is difficult to determine a specification in a particular program. In this paper to improve the effectiveness, two techniques inference through static analysis and inference through dynamic invariant detection are added. Two different annotation assistant tools like Daikon for dynamic case and Houdini for static case is used. The static checking can be improved using the assistance of both the tools. Comparing with our approach, this approach fails in providing the required number of effective testcases for generating the invariants.

3. Background Study

In this paper, some basic concepts about bug detection in concurrent program using invariant analysis are discussed. Some of the concepts are discussed broadly below.

3.1. The Concurrent Bugs

A bug in a program code is an error or flaw which causes it to behave in an unexpected way and produce undesirable results. It is of two types:

i. Bohrbugs
ii. Heisenbugs

A bohrbug can be easily detected and corrected by using standard debugging techniques i.e. these are deterministic in nature while a heisenbug is completely non-deterministic and transient in nature. We cannot use standard debugging techniques to correct them. It is otherwise known as concurrency bug [10]. Given below are a few common types of concurrency bugs:

i. *Atomicity Violation:* An atomic block of code or method of a thread gets interrupted due to interleaving of some other thread during its execution.
ii. *Order Violation:* When the shared variables are accessed by multiple threads in a wrong order, or read/write operation is done in wrong order, then order violation occurs.
iii. *Deadlock:* This is the most common problem in concurrent programs. Multiple threads are competing for getting access to a common shared resource but each has to wait for an infinitely long time as each thread is holding some resource and requesting for another.
iv. *Bad Composition:* Sometimes methods aren't well written for extension but it is accessed by multiple threads. It may result in incorrect behaviour for some specific interleaving.

Example:

Thread 1:	Thread 2:
pthread_mutex_lock(L1);	pthread_mutex_lock(L2);
pthread_mutex_lock(L2);	pthread_mutex_lock(L1);

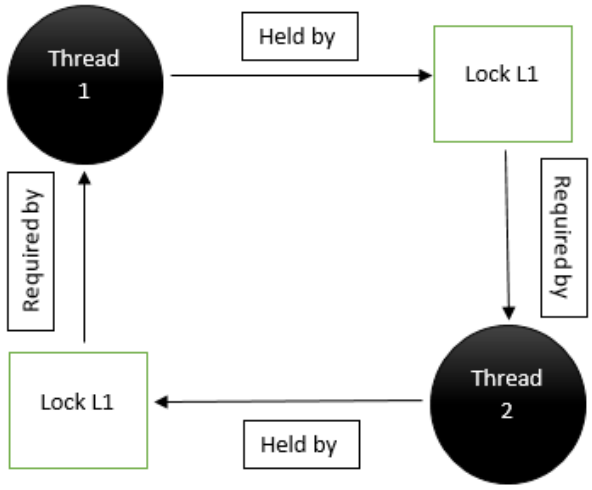

Fig 1. Pictorial description of a common deadlock

The example shown in Fig 1, shows a code snippet which may not always result in a deadlock however, for some specific interleaving, it may result in a deadlocked situation.

3.2. Program Invariants

A program invariant is a character that remains constant throughout the execution of the program [11]. It is considered as a logical assertion that is always held true during any phase of execution. It is usually used in assert statements, documentations and formal specifications. In the above example, there are various types of program invariants like being non-zero ($a \neq 0$), being in a range ($0 < a < 9999$), following an ordering ($b \geq a$), etc. The various types of invariants [2] are:

i. Loop invariants: It is an invariant that holds true at the entry as well as the exit of every iteration of a loop.
ii. Class invariants: It is used to restrict the objects of a class, i.e. all the member functions must preserve the invariant.

Example:

```
int f1(int a, int b)
{
int s=a;
for (int i = 1; i < b; i++)
{
sum=2*a+i;                    // loop invariant
}
return sum;
}
```

3.3. Static Call Graph

It is used for representing the caller-callee relationships between subroutines in a computer program. Each node represents a method and each edge (*f*, *g*) represents the method f calling g. A circular path in the call graph indicates recursion. An example of a static call graph is shown in Fig 2.

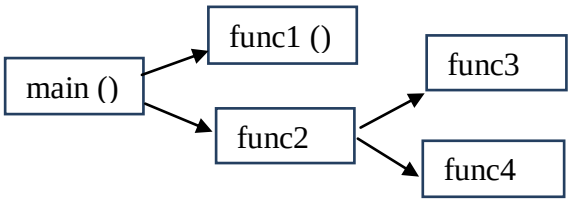

Fig 2. Example of a Static Call Graph

Call graphs are very easy to understand and can be used for maintenance and further extension of an application. It reduces the tedious job of going through multiple lines of code to find a particular program feature (module, sub-routine, etc). It is the graphical representation of the work flow. It can be used to trace the function calls in runtime. It can be used for reverse engineering and debugging. It can also show the modules that aren't called at all.

4. Invariant Analysis for Bug Detection

This section discusses the set of steps conducted to obtain various concurrent bugs from the invariants which are generated from the source code. The workflow of our approach to obtain the required set of concurrent bugs is described in Fig. 3. It is performed through 3 steps.

Step 1: First, we select the various components of the software which means the threads or processes of that software. Let us take the small processes as A, B, .. We find out the invariants from the individual components.
Step 2: We find out the invariants of individual components and as a whole the total invariants of the concurrent software.

Step 3: Combining the static call graph with a reduction technique which is used for reducing the redundant invariants for both set of invariants. Through this we are able to find out the concurrent bugs present in the software.

4.1. Workflow of the Proposed Approach

The input to this schematic model is the source code. Using the Daikon tool, we got two types of invariants through unit test cases and integrated test cases. The concurrent bugs can be located in components as well as in the oracle while integration. The threads used during the function call graph maybe shared during different function usage. The bug appears when thread 1 and thread 2 (say) share a common variable and compete for occupying it. The invariant method shows the various invariants present in a particular component. After integration of the components, we may get different invariants which are common between the component and the oracle.

The technique which we propose is able to detect the fault [13] in the data accessing among the different variable through monitoring the shared single variables. The patterns which are faulty can be calculated by $W_1 - R_2$ where W_1 represents the writing operation of thread 1 and R_2 represents the reading operation of thread 2. The concurrent bugs which are related to different variables may not work for all the techniques followed. In our approach the bug detection can be done using the static call graph and the failure invariants. The static call graph is used for finding the function relations between two different functions. The failure invariants [2] are the invariants which are ultimately not going to be true for all the cases.

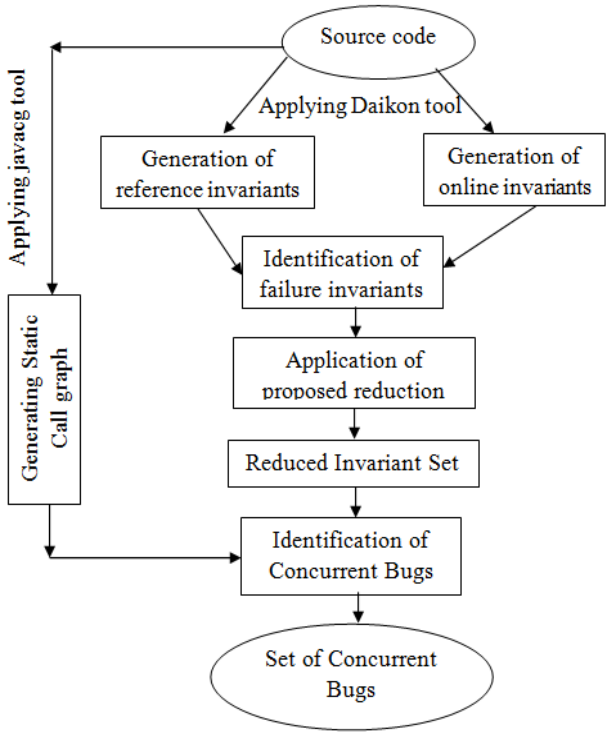

Fig 3. The schematic diagram of our approach

4.2. Sample Program

We have considered the Dining Philosopher problem as our sample problem. It is basically a synchronization problem where threads compete with each other to get some resources while previously holding some resources. The code of the dining philosopher problem can be found at [12]. A snapshot of the dining Philosopher problem is shown in Fig 4.

```
package diningphilosopher;

import java.util.ArrayList;

public class Dining
{
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Chopstick chopstick1 = new Chopstick("chopstick 1 ");
        Chopstick chopstick2 = new Chopstick("chopstick 2 ");
        Chopstick chopstick3 = new Chopstick("chopstick 3 ");
        Chopstick chopstick4 = new Chopstick("chopstick 4 ");
        Chopstick chopstick5 = new Chopstick("chopstick 5 ");

        Philosopherer philosopher1 = new Philosopherer("philosopher 1 ", chopstick5,chopstick1);
        Philosopherer philosopher2 = new Philosopherer("philosopher 2 ", chopstick1,chopstick2);
        Philosopherer philosopher3 = new Philosopherer("philosopher 3 ", chopstick2,chopstick3);
        Philosopherer philosopher4 = new Philosopherer("philosopher 4 ", chopstick3,chopstick4);
        Philosopherer philosopher5 = new Philosopherer("philosopher 5 ", chopstick4,chopstick5);
        long startTime = System.currentTimeMillis();
        ArrayList<Thread> threads = new ArrayList<Thread>();
        threads.add(philosopher1);
        threads.add(philosopher2);
        threads.add(philosopher3);
        threads.add(philosopher4);
        threads.add(philosopher5);
        //--Start the order of the meal
        philosopher2.start();
        philosopher4.start();
        philosopher1.start();
        philosopher3.start();
        philosopher5.start();
```

Fig 4. Dining Philosopher problem code.

4.3. Daikon Tool for Generating Invariants

This tool is used for detecting the likely program invariants dynamically. The likely invariants can be found out from the program execution through checking the values, verifying the patterns and finding the relationship between the various variables. The potential invariants are generally generated by instantiating at each method entry and exit. Although there are many potential invariants [16], testing is efficient because most potential invariants are falsified quickly and don't need to be tested. Fig 5 provides a snapshot of the invariants generated by Daikon for the Dining Philosopher problem.

```
----------------------------------------------------------------
==========
Chopstick:::OBJECT
this.name != null
================================================================
==========
Chopstick.Chopstick(boolean, java.lang.String):::ENTER
enable == true
================================================================
==========
Chopstick.Chopstick(boolean, java.lang.String):::EXIT
this.enable == orig(enable)
this.name == orig(name)
name.toString == orig(name.toString)
this.enable == true
================================================================
==========
Chopstick.Chopstick(java.lang.String):::ENTER
================================================================
==========
Chopstick.Chopstick(java.lang.String):::EXIT
this.name == orig(name)
name.toString == orig(name.toString)
this.enable == true
================================================================
==========
Chopstick.pickdown():::ENTER
this.enable == false
================================================================
==========
Chopstick.pickdown():::EXIT
```

Fig 5. Snapshot of invariant list for Dining Philosopher generated by Daikon tool.

Table 1. Invariant List for Dining Philosopher problem

Functions	Invariants
Chopstick.pickdown():::ENTER	this.enable = false
Chopstick.pickdown():::EXIT	this.name = orig(this.name)
	this.name.toString = orig(this.name.toString)
	this.enable = true
Chopstick.pickup():::ENTER	
Chopstick.pickup():::EXIT	this.name = orig(this.name)
	this.name.toString = orig(this.name.toString)
	this.enable = false
Philosopherer.run():::ENTER	
Philosopherer.run():::EXIT	this.name = orig(this.name)
	this.name.toString = orig(this.name.toString)
	this.leftChopstick = orig(this.leftChopstick)
	this.leftChopstick.name = orig(this.leftChopstick.name)
	this.leftChopstick.name.toString = orig(this.leftChopstick.name.toString)
	this.rightChopstick = orig(this.rightChopstick)
	this.rightChopstick.name = orig(this.rightChopstick.name)
	this.rightChopstick.name.toString = orig(this.rightChopstick.name.toString)

4.4. Failure Detection Using Program Invariants

4.4.1. Finding Failure Invariants

Failure invariants are basically the invariants which are not in the correct invariant set. From Table 1, we find out some invariants which are present in unit testing are not present in integration testing [14]. The invariants maybe called as false positive invariants which means these invariants exists but not visible or detected during unit testing.

The false negative invariants which can be considered as the positive invariants are discussed later in the next section. In Table 1,

this.leftChopstick.name.toString == orig (this.leftChopstick.name.toString) invariant is present in unit testing but not present in integration testing and hence present in the failure invariant. So, these types of invariants are called failure invariants.

4.4.2. Filtering the failure Invariants

Using Daikon tool we are able to find out invariants from unit testing and integration testing. Now, for finding the difference of these two types of invariants we usually take six different types of notations like UI, U!I, NI, N!I, Bin and Ter. Where U represents unary, Bin represents binary, Ter represents ternary and I represent interesting. We filter out the unjustified or uninteresting invariants by removing the U related terms.

4.5. Modified Program Invariant Reduction

For reducing the redundant invariants [15] of each function, we use the following steps:
i. The various failure invariants which are present at Exit points of the function and its parent function, we reduce/deduct the invariant from the parent function.
ii. We can check the invariants present in parent and child functions, whether they occur for the same reason or not.
iii. When the failure invariants occur at both Entry and Exit points of a function, the reduction is done for all the failure invariants occurring in parent function.
iv. We should check the false negative invariants which are actually positive and shouldn't be considered as a negative invariant.
Previously only i and iii steps are followed. But we have introduced ii and iv for better reduction mechanism. Here, Enter and Exit are the two parts of the function which are controlled through ENT_i and EX_i. The techniques for reduction consist of the operations like:

$$\text{i. } ENT_i = ENT_i - EX_i \quad (1)$$

$$\text{ii. } EX_i = EX_i - ENT_i \quad (2)$$

$$\text{iii. } EX_i = EX_i - U\ \{EX_i : f_j \rightarrow f_i\} \quad (3)$$

Where, $f_j \rightarrow f_i$ means function f_j calls f_i, which is determined through the function call graph. We use (1) and (2) for checking the failure invariants at both Enter and Exit level. These failure invariants may not occur because of the Enter and Exit portions of f_i. (3) is used for checking the repeated failure invariants at f_j. We also check the negative failure invariants here which are actually positive and discard the reductions.

4.6. Javacg Tool for Generating Call Graph

The javacg is a collection of java programs which is used for building the function call graph of a java program. The javacg package has two jar files namely:

i. javacg-0.1-SNAPSHOT-static.jar: it is the .jar file that has the static call graph builder.

ii. javacg-0.1-SNAPSHOT-dycg-agent.jar: it is the .jar file that has the dynamic call graph builder.

In our implementation we have used the static call graph builder to find out the relationship between the caller and callee in our sample program i.e. the dining philosopher problem. As input, it takes a .jar file which consists of all the classes of the program for which we want the function call graph. Then it gives the caller-callee relationship in a tabular format. The snapshot of the static call graph generated through the javacg tool is provided in Fig 6. The caller-callee graph of the sample program is represented in a pictorial manner in Fig 7 for better understanding.

```
C:Chopstick java.lang.Exception
M:Chopstick:<init>(boolean,java.lang.String) (O)java.lang.Object:<init>()
M:Chopstick:<init>(java.lang.String) (O)Chopstick:<init>(boolean,java.lang.String)
M:Chopstick:pickup() (M)java.lang.Object:wait()
M:Chopstick:pickup() (M)java.lang.Exception:printStackTrace()
M:Chopstick:pickdown() (M)java.lang.Object:notifyAll()
C:Dining Chopstick
C:Dining Philosopherer
C:Dining java.util.ArrayList
C:Dining java.lang.Thread
C:Dining java.lang.InterruptedException
C:Dining Dining
C:Dining java.lang.Object
C:Dining [Ljava.lang.String;
C:Dining Chopstick
C:Dining Philosopherer
C:Dining java.util.ArrayList
C:Dining java.util.Iterator
C:Dining java.lang.Thread
C:Dining java.lang.InterruptedException
C:Dining java.lang.System
C:Dining java.util.Iterator
C:Dining java.io.PrintStream
M:Dining:<init>() (O)java.lang.Object:<init>()
M:Dining:main(java.lang.String[]) (O)Chopstick:<init>(java.lang.String)
M:Dining:main(java.lang.String[]) (O)Chopstick:<init>(java.lang.String)
M:Dining:main(java.lang.String[]) (O)Chopstick:<init>(java.lang.String)
M:Dining:main(java.lang.String[]) (O)Chopstick:<init>(java.lang.String)
M:Dining:main(java.lang.String[]) (O)Chopstick:<init>(java.lang.String)
M:Dining:main(java.lang.String[]) (O)Philosopherer:<init>(java.lang.String,Chopstick,Chopstick)
M:Dining:main(java.lang.String[]) (O)Philosopherer:<init>(java.lang.String,Chopstick,Chopstick)
M:Dining:main(java.lang.String[]) (O)Philosopherer:<init>(java.lang.String,Chopstick,Chopstick)
M:Dining:main(java.lang.String[]) (O)Philosopherer:<init>(java.lang.String,Chopstick,Chopstick)
M:Dining:main(java.lang.String[]) (O)Philosopherer:<init>(java.lang.String,Chopstick,Chopstick)
M:Dining:main(java.lang.String[]) (S)java.lang.System:currentTimeMillis()
M:Dining:main(java.lang.String[]) (O)java.util.ArrayList:<init>()
M:Dining:main(java.lang.String[]) (M)java.util.ArrayList:add(java.lang.Object)
M:Dining:main(java.lang.String[]) (M)java.util.ArrayList:add(java.lang.Object)
M:Dining:main(java.lang.String[]) (M)java.util.ArrayList:add(java.lang.Object)
M:Dining:main(java.lang.String[]) (M)java.util.ArrayList:add(java.lang.Object)
M:Dining:main(java.lang.String[]) (M)java.util.ArrayList:add(java.lang.Object)
M:Dining:main(java.lang.String[]) (M)Philosopherer:start()
M:Dining:main(java.lang.String[]) (M)Philosopherer:start()
M:Dining:main(java.lang.String[]) (M)Philosopherer:start()
M:Dining:main(java.lang.String[]) (M)Philosopherer:start()
M:Dining:main(java.lang.String[]) (M)Philosopherer:start()
M:Dining:main(java.lang.String[]) (M)java.util.ArrayList:iterator()
M:Dining:main(java.lang.String[]) (I)java.util.Iterator:hasNext()
M:Dining:main(java.lang.String[]) (I)java.util.Iterator:next()
M:Dining:main(java.lang.String[]) (M)java.lang.Thread:join()
```

Fig 6. Snapshot of the static call graph of Dining Philosopher problem obtained from javacg tool

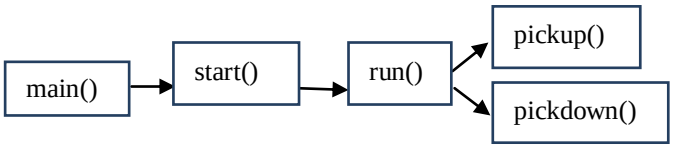

Fig 7. Static call graph of the Dining Philosopher problem

4.7. Case Studies for Bug Detection

Using our approach we found out the test cases of integration testing from Daikon along with the static call graph from javacg tool. Combining these above things with the reduced invariants we are able to find the concurrent bugs along with the number of threads used and the LOC, which are discussed in Table 2. Four different case studies are considered for validating the approach. The test cases may be executed randomly for many times. The bugs can be of three types: deadlock, atomic violation and bad composition. The case studies are:

Dining Philosopher:

In the dining philosopher problem, there are total 5 numbers of philosophers with 5 chopsticks to eat the noodles. One chopstick will be on the left side and another on the right side of the philosopher. As the number of chopsticks is less than the required number of chopsticks, the deadlock may occur when two adjacent philosophers request for the same chopstick to eat. When the program encounters deadlock the failure invariants [17] can be captured there comparing with the present invariants.

Bounded Buffer:

In the bounded buffer problem, the producer provides the elements to the stack and the consumer consumes from the stack. The problem occurs when the producer provides more elements to the buffer but the consumer is unable to consume all the elements. In another case, the producer is lacking in providing the element but the consumer wants to consume the elements. In both the cases, bad

composition concurrent bug [18] will be encountered. This problem can be checked through generating a set of reference invariants from the program and some component wise invariants are checked for bad composition bug.

Rand:

The Problem is based on random generation of two variables "a" and "b". The third variable "c" is used for generating the randomized tree. Here the tree is made based on the random choosing of the variables and the outputs obtained by the variable c.

Old Classic:

This example shows a deadlock that occurs as a result of a missed signal, i.e. a wait() that happens after the corresponding notify(). The defect is caused by a violated monitor encapsulation, i.e. directly accessing monitor internal data ('Event.count') from concurrent clients ('FirstTask', 'SecondTask'), without synchronization with the corresponding monitor operations ('wait_for-Event()' and 'signalEvent()'). The resulting race [19] is typical for unsafe optimizations that try to avoid expensive blocking calls by means of local caches.

4.8. Experimental Results and Discussion

From Table 2, we got the data about the lines of code, number of testcases and the number of threads with different type of concurrent bugs obtained from different concurrent programs [20]. The concurrent programs that are discussed above mainly faces 3 types of concurrent bugs like Deadlock, Atomic violation and Bad composition. These bugs are invisible which occurs at runtime. The snapshots of the outputs through JPF tool of some of the concurrent programs are given in Fig 8 and Fig 9.

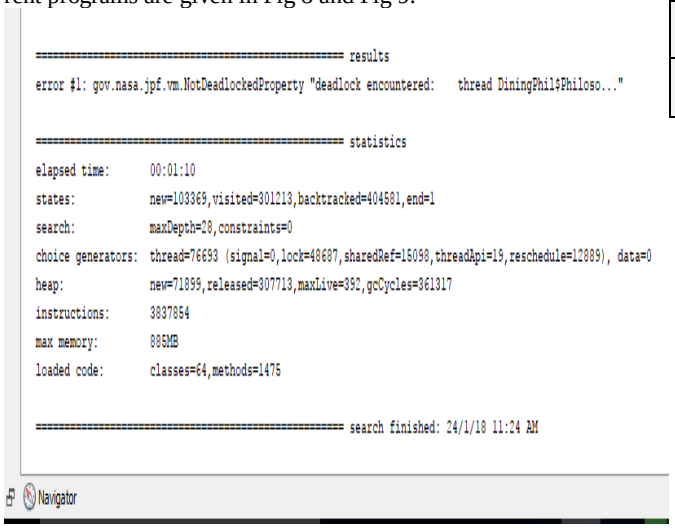

Fig 8. Snapshot of Dining Philosopher program

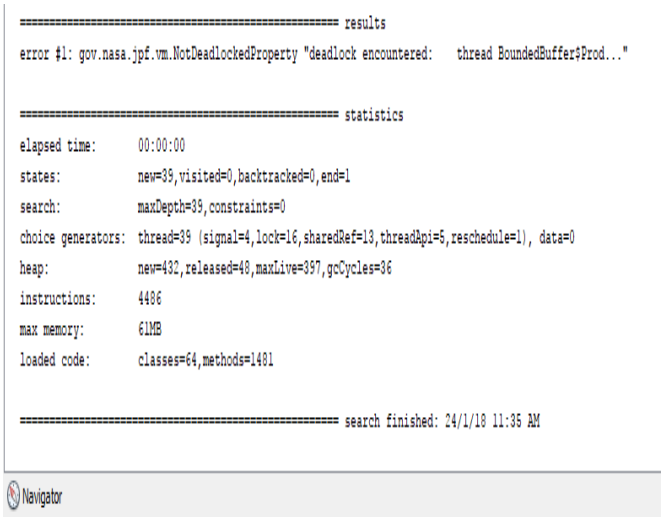

Fig 9. Snapshot of BoundedBuffer program

Table 2: Information about Concurrent bugs

Program	LOC	Bug Description	No. of Threads	Test cases
Dining Philosopher	45	Deadlock	76693	1475
Bounded Buffer	110	Deadlock, bad composition	39	1480
Rand	19	Atomicity violation	1	1362
OldClassic	96	Deadlock	32	1477

Through our approach, we got deadlock in bounded buffer problem which was not detected by earlier approaches. We also found out the concurrent bugs like atomic violation in Rand program which was not detected earlier. The numbers of test cases are significantly different in comparison with the earlier approaches.

5. Conclusion

This paper presents an invariant based approach for detection of various concurrent bugs. The concurrent bug occurs due to bad interleavings between objects. Through our approach, it can be able to reduce the no. of redundant invariants and able to find out the different types of concurrent bugs using the invariant based approach. Reduction in no. of invariants also results in reduction of no. of unwanted test cases. The proposed approach provides a correct range of invariants. The future work can be to trace different other causes of concurrent bugs using invariants. The failure invariants can be checked which are dependent on other variables.

References

[1] V. Kahlon and C. Wang, "Universal Causality Graphs: A precise happens-before model for detecting bugs in concurrent programs", (2010) International Conference on Computer Aided Verification, pp. 434–449.

[2] Wang, R., Ding, Z., Gui, N., & Liu, Y., "Detecting Bugs of Concurrent Programs with Program Invariants", (2017) IEEE Transactions on Reliability, 66(2), pp. 425-439.

[3] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., & Xiao, C., "The Daikon system for dynamic detection of likely invariants", (2007) Science of Computer Programming, 69(1-3), pp. 35-45.

[4] Gonzalez, A., "Automatic error detection techniques based on dynamic invariants", (2007) (Doctoral dissertation, MS thesis, Delft University of Technology, The Netherlands).

[5] Isaratham, W., "Equivalence Partitioning as a Basis for Dynamic Conditional Invariant Detection", (2015) (Doctoral dissertation, National University of Ireland Maynooth).

[6] Kusano, M., Chattopadhyay, A., & Wang, C., "Dynamic generation of likely invariants for multithreaded programs", (2015) In Software Engineering (ICSE), IEEE/ACM 37th IEEE International Conference, Vol. 1, pp. 835-846.

[7] Zaharieva-Stojanovski, M., & Huisman, M., "Verifying class invariants in concurrent programs", (2014) In International Conference on Fundamental Approaches to Software Engineering, Springer, pp. 230-245.

[8] Ellsworth, D., "Improving Dynamic Invariant Saliency with Static Dataflow Analysis", (2013).

[9] Nimmer, J. W., & Ernst, M. D., "Invariant inference for static checking: An empirical evaluation", (2002) ACM SIGSOFT Software Engineering Notes, Vol. 27, Issue 6, pp. 11-20.

[10] Bianchi, F., Margara, A., & Pezze, M., "A Survey of Recent Trends in Testing Concurrent Software Systems", (2017) IEEE Transactions on Software Engineering.

[11] Betts, A., Chong, N., Donaldson, A., Deligiannis, P., & Ketema, J., "Implementing and evaluating candidate-based invariant generation", (2017) IEEE Transactions on Software Engineering.

[12] http://my.oschina.net/sharkbobo/blog/270238.

[13] Wong, W. E., Gao, R., Li, Y., Abreu, R., & Wotawa, F., "A survey on software fault localization", (2016) IEEE Transactions on Software Engineering, Vol. 42, Issue 8, pp. 707-740.

[14] Hong, S., Staats, M., Ahn, J., Kim, M., & Rothermel, G., "The impact of concurrent coverage metrics on testing effectiveness", (2013) In Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference, pp. 232-241.

[15] Chattopadhyay, Arijit., "Dynamic Invariant Generation for Concurrent Programs", (2014) PhD diss., Virginia Tech.

[16] Park, Sangmin, Richard W. Vuduc, and Mary Jean Harrold. "Falcon: fault localization in concurrent programs", (2010) In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, Volume 1, pp. 245-254.

[17] Auer, A., Dingel, J., & Rudie, K., "Concurrency control generation for dynamic threads using discrete-event systems", (2014) Science of Computer Programming, Vol. 82, pp. 22-43.

[18] Lu, S., Tucek, J., Qin, F., & Zhou, Y., "AVIO: detecting atomicity violations via access interleaving invariants", (2006) In ACM SIGOPS Operating Systems Review, Vol. 40, No. 5, pp. 37-48.

[19] Zhang, W., Lim, J., Olichandran, R., Scherpelz, J., Jin, G., Lu, S., & Reps, T., "ConSeq: detecting concurrency bugs through sequential errors", (2011) In ACM Sigplan Notices, Vol. 46, No. 3, pp. 251-264.

[20] Farzan, A., Holzer, A., Razavi, N., & Veith, H., "Con2colic testing", (2013) In Proceedings of the 9th Joint Meeting on Foundations of Software Engineering, pp. 37-47.

[21] https://github.com/gousiosg/java-callgraph