International Journal of Engineering & Technology, 7 (2.6) (2018) 275-282

International Journal of Engineering & Technology

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

Research Paper

Concurrency testing using symbolic path finder

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*
**Email: bidush.sahoo@gmail.com*

Abstract

Concurrent programs have specific features such as italic communication, synchronization and nondeterministic behavior that make the testing activity complex. The objective is to find various types of concurrent defects. In this paper, we have used a model checking tool called Symbolic Path Finder (SPF) which is the upgradation of Java Path Finder (JPF) for concurrent testing. SPF is used for generating the test cases to check concurrent defects such as deadlock, race condition etc. SPF generates symbolic execution tree of the given code which is used as an input for test case generation. The execution is done for finding the test cases in concurrent program where number of threads is operating together with the concurrent defects. The test cases show the type of concurrent defects in the respective line number of the source code.

Keywords: *Coverage Criteria; Java Path Finder; Model Checker; Symbolic Execution; Symbolic Path Finder.*

1. Introduction

A large number of operational components are most likely to happen in a concurrent program simultaneously. Each part of the operation thus expected in a sequential program. Each part of the operation is communicating with one another. Each such sequence of small operations is called as thread. Because of this activity, the sequential programs are known as single-threaded programs [1]. So, in a multi-threaded program execution, number of threads operates randomly with a condition imposed by a synchronization behavior.

As the interleaving of operations is unpredictable, it depends on the roles of the execution of a particular program. The interleaving operations among various threads make the concurrent program extremely difficult to test. It implies that the nature of concurrent program execution is non-deterministic. Moreover, the analysis of concurrent program is difficult because of the complexity caused by multiple thread interactions [2]. So, for prevention of complexity in accessing the operations, the shared variables are made as atomic. Atomic operation is executed as a single machine instruction as it can pause other thread's execution. So, other threads cannot interfere in the updated values of atomic execution. The concurrent program testing checks various types of concurrent defects like race condition, deadlock, atomic violation, etc.

Non-determinism can be simulated by JPF which normal testing is not able to do. The execution environment of JPF helps the scheduling sequence which is restricted by the test driver. However, the systematic generation of all non-deterministic choices is required by the JPF tool. The two processes that is able to solve this simulation problem is:

i. Back-tracking
ii. State matching

In back-tracking mechanism, JPF restores the previous unexplored execution states, if it is present. JPF can walk backward in order to find different possible scheduling sequences which are not yet executed. Back-tracking is an efficient mechanism when the state storage is minimized.

The second approach for avoiding unnecessary work is state matching. Heap and thread stack snapshot are the major part of the execution state of the program. JPF checks every new state which is similar to other states. It can back-track to the nearest unexplored and non-deterministic choice.

There are two inputs to JPF:

i. The class file of source code.
ii. The configuration file that specify the execution mode and properties to be verified.

The verification report contains the concurrent error and its positions. JPF forms some extensive features like implementation of new execution mode, checking of program properties, formatting the reports and creating user interfaces. As JPF is a model checker, so it supports back-tracking, state matching and non-determinism in data and scheduling decision. Through JPF, the state space execution tree is generated. In the state space sequence the byte code instructions is termed as transitions. The first instruction in the sequence is basically non-deterministic choice of a thread in context switching format.

In each transition, JPF saves the current state for back-tracking and state matching purpose. The state changes which are performed within JVM are also included as the job of JPF. JPF is a combination of various components which are configured in runtime.

The sections are arranged in the following sequence. Section 2 provides the literature survey of the various works done. Third section contains the framework for implementation of testing using SPF. The detailed implementation with results is discussed in Section 4. Section 5 concludes the result and provides the future guidelines.

2. Related Work

A lot of work is done in the field of concurrent program testing and but the work on SPF are few. The concurrent testing is done using various coverage criteria such as synchronization coverage, interleaving coverage etc. The existing work are basically on providing different coverage criteria in the verification tool and tried to get maximum coverage over different perspectives.

As per Pasareanu et. al [3], the JCBMC tool is used for checking the models provided by boundary condition. The source code is transferred into some symbolic formula in the disjunctive form. Through this model the verification can be done with the base thread not using any condition for getting sub formula in the symbolic execution. The secondary thread is present for checking the decision methods and provides the verification conformity.

As per Enoiu et.al [4], the functional testing is used for validating the implemented code with its relevant properties. Model checking is generally used for structural test case generation. As now a day, model checking is less used so the state space exclusion problem creates different problems in creating the test cases. Hence for validation of these types of approaches in industrial applications unique approach is needed. The functional block diagram language is used for solving these types of validation problems.

Symbolic execution can be treated as a program analysis technique in the view of Kersten et. al [5]. They have used symbolic execution as a method for test case generation. Symbolic execution can generate test sets for gaining for path coverage in a loop free program. SPF is the extended version of JPF model checker for reducing the state space explosion problem. In this approach a bounded loop is taken for consideration to check the user control in the symbolic execution method. When symbolic execution is used for test case generation the paths are checked but the branch coverage may be losing its value.

As per Vissar et. al [6], the model checking and symbolic execution are used for the structural coverage of the source code are used in the complex data structure. Branch coverage of the source code during the symbolic execution is the main idea of this approach. This approach basically provides focus on white box testing in complex data set.

JPF tool is not only used for model checking in Java environment, it is also used for reducing state space explosion problem in testing. SPF is the advancement of JPF which checks the problem of test case generation of concurrent program through symbolic execution tree.

3. Proposed Approach for Concurrent Execution

The workflow is provided in Fig-1.

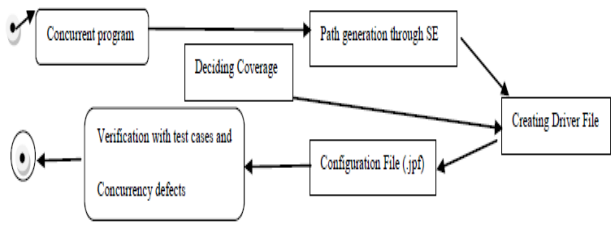


Fig. 1: Schema diagram for concurrent testing using SPF

The above described schema diagram (Fig-1) represents the work flow of the approach. A concurrent program is considered for testing. The coverage criterion is decided to cover the test case. The path generation file is created from the concurrent program for symbolic execution tree. Then, the driver file is created from coverage and path builder. Then, the configuration file is created for the class and driver file. The output is generated by verifying the configuration (.jpf) file. The generated output are- (i) a set of automatically generated test cases and (ii) the line numbers marked with concurrency defects.

3.1. The Concurrent Program

The concurrent program execution of a program is provided below, where the threads interchange [7] their values at different point of time. The condition change and the value change in a given time.

A class file with the constraints is provided below.

```
public class SPF_Class {
  public int foo(int x, int y) {
    int z = x + y;
    if (z > 0) {
      z = 1;
    } else {

      z = z - x;
    }
    if (x < 0) {
      z = z * 2;
    } else if (x < 10) {
      z = z + 2;
    } else {
      z = -z;
    }
    if (y < 5) {
      z = z - 10;
    } else {
      z = z - 20;
    }
    return z;
  }
```

Program. 1: An example of concurrent program

Program-1 is a concurrent program, considered for model testing through SPF where we can check the concurrent defects in it and the infeasible paths also.

3.2. Java Path Finder

JPF is a combination of various components which is executed through some configured run time environment. It is configured in java environment. It provides the output as the checking the concurrent errors in the source code while running.

Virtual machine is basically helps the core program to run. For example JPF core which is meant for java byte code helps in executing java program for finding the concurrent errors. JPF core is also takes the configuration/properties as an input for verification. The java path finder tool provides the report of verification which checks the types of concurrent errors [8] and the different test cases of source code for further analysis. As the virtual machine run little bit slow in comparison with the programming language it helps, so JPF runs slow in comparison with core java.

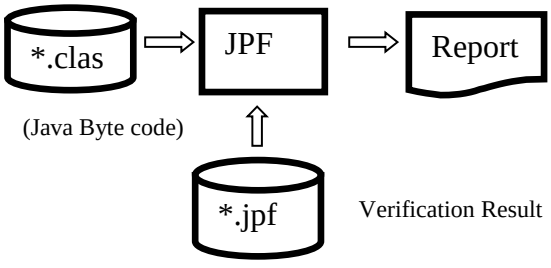


Fig. 2: JPF work structure

The structure in Fig-2 describes the work structure of the JPF model checker which provides the model checking facility along with its different features. Execution choice is a feature of JPF which is used through instruction set. The execution diversion can be identified through JPF tool in the source code and checks all of them in orderly format. So, the execution mechanism in JPF tool is different from another virtual machine as it executes the complete path of the source code. The uniqueness of JPF tool is allowing the user to provide its own choice of inputs.

In a state diagram of the source code of concurrent programs the path length increases at will. This is basically called state explosion problem in software testing. JPF uses back tracking technology where it checks the states of the program through state matching [9]. Once it finds a similar program state it will back track to the previous choice point/state which is unexplored. It will start from that unexplored choice/state and proceed till it finds any concurrent error.

3.3. Symbolic Execution

A technique in software engineering testing for generating the test data in improvising the program quality is symbolic execution. There are different states of using symbolic execution. These are:

1. The path selection is the prime job of the symbolic execution. The actual data provides series of results when the symbolic execution is executed.
2. Symbolic values replace the set of expression in the symbolic execution tree with an output variable represented through an expression.
3. A flow graph is generated through the symbolic execution which is obtained from the source code.
4. Each flow in the flow graph provides the identification of decision point and the job attached with it. The flow graph from top to bottom the job assignments and branching are provided along with the path condition.
5. The path condition is based on the input symbols to check the infeasible solution.

The basic idea of using symbolic execution is replacing the actual data and inputs by the symbolic values. It also helps for displaying the symbolic expression in graphical format. So the symbolic values are used for getting the output value using the program function.

```
Algorithm Symbolic_Execution_Creater (S)
// given the source code, S.
// the output will be the symbolic execution tree, T, of S.
01      If (S is Uninitialized) {
02      If (S is reference field of type T) {
03      Nondeterministically initialize
04       1. T to Null
05       2. A new object of class T {with uninitialized field values}
06       3. An object created during a prior initialization of a field
of type T
07      If (method precondition is violated) {
08      backtrack () ;}
09      If (S is primitive (or string) field)
10      Initialize S to a new symbolic value of appropriate type}}
```

Algorithm. 1: Algorithm for creating Symbolic Execution

Algorithm-1 is the generalized lazy initialization algorithm to generate the symbolic execution tree. In this algorithm, uninitialized fields are the method inputs. It uses lazy initialization for initializing the values, which means the field initialization is done when it access the methods symbolic execution. The input objects in the method are not provided in a boundary at the beginning. The program which is executed symbolically has three parts.

1. Symbolic values of variables.
2. Path condition
3. Program counter

Among this the path condition is the condition used by the symbolic inputs. It checks the condition which the input follows for relevant path association. The program counter is the counter for checking the execution of the next execution. The symbolic execution tree [11] shows the paths followed during the symbolic execution process. The tree nodes represent the states and the edges represent the activity between the nodes.

For an example in a program which is used for interchanging the integer variables x and y, where x>y. the symbolic execution tree is made in the fig. 5. In the starting the program counter will be considered as true. x and y has the corresponding symbolic values as X and Y. During updating the program counter, it selects between two different paths mention in fig. 5[12]. In this figure it shows the conditional statements are executed after the execution of its previous statement. The program counter is changing its value according to the proceedings. When the path condition gives negative value then it shows that there is no input which can satisfy the condition. That means unreachable condition arises and the symbolic execution can't proceed in that path.

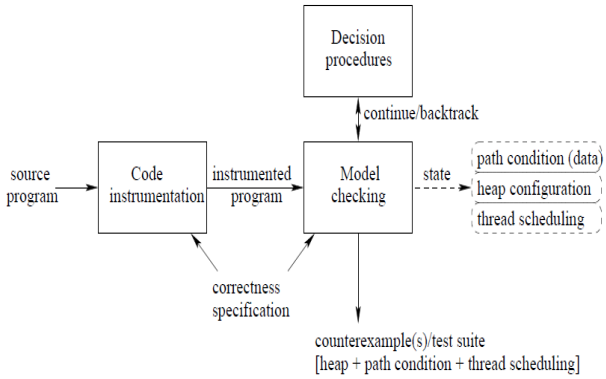


Fig. 3: Methodology for generating symbolic execution tree

The above methodology described graphically in Fig-3 signifies the model checking procedure along with the outputs as path conditions and thread scheduling. In this figure, the code instrumentation is done for rectifying the specifications. The instrumented program is taken as the input for model checking. Then the output will be generated as the path conditions and the thread scheduling.

```
int x, y;
1: if (x > y) {
2: x = x + y;
3: y = x - y;
4: x = x - y;
5: if (x - y > 0)
6: assert (false);
}
```

Program. 2: A sample code

The above sample code is used for generating the symbolic execution tree from the above code using the algorithm-1.

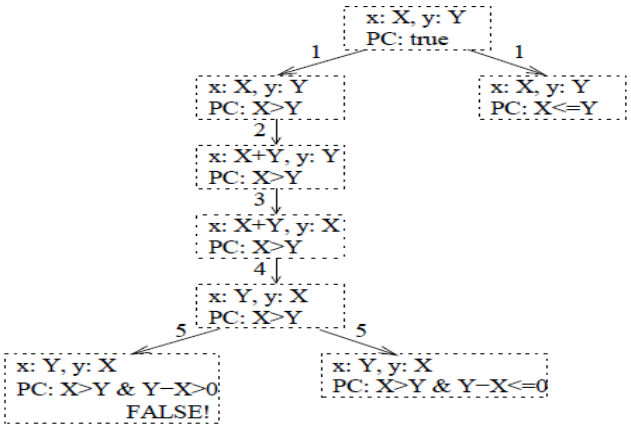


Fig. 4: converted symbolic execution tree of program-2

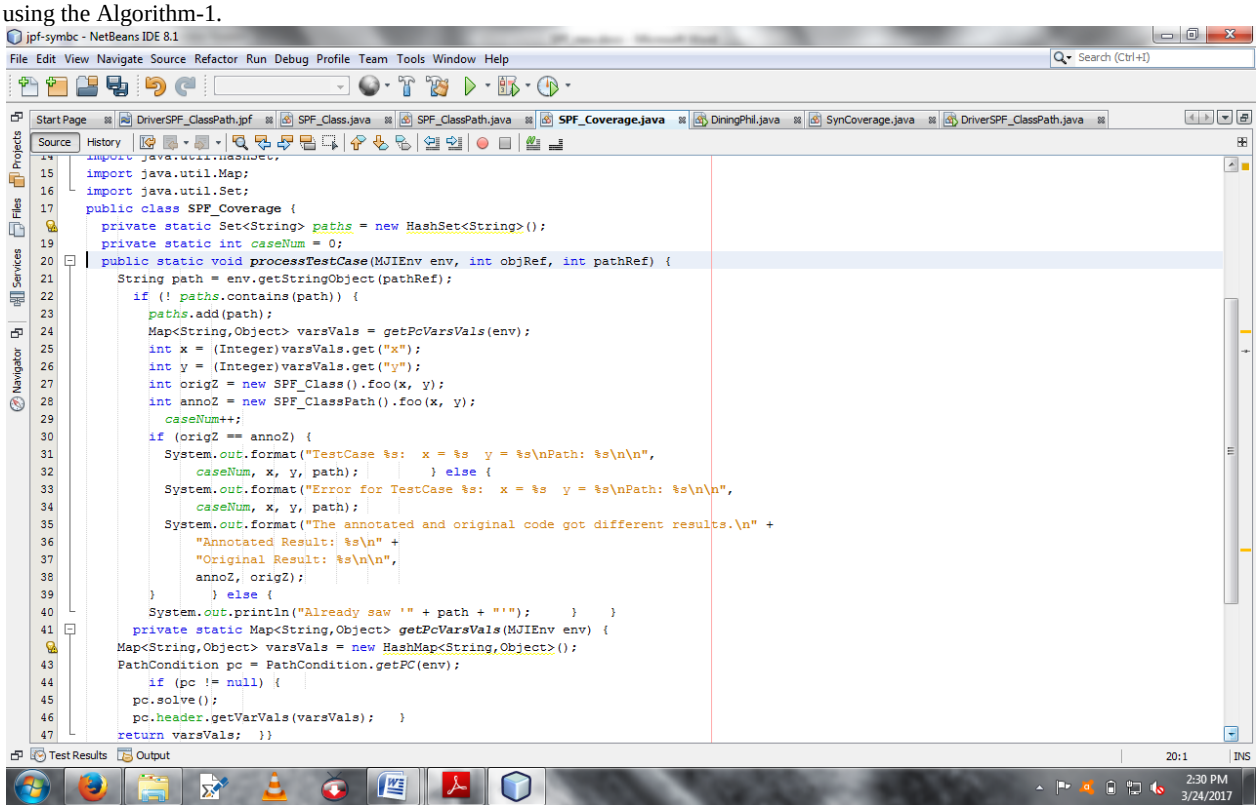
The above figure (Fig-4) is the converted symbolic tree of the program-2 given above. The symbolic execution tree is generated using the Algorithm-1.

3.4. Symbolic Path Finder

Symbolic Path Finder (SPF) is the enhanced project of JPF which is available in the project jpf-symbc [13, 14]. It provides extended facility of jpf core from standard to non-standard byte code for symbolic interpretation. The information required during symbolic execution are basically stored in the attributes related with the program variable.

SPF improvised jpf in executing the symbolic execution tree by handling the multithreading concept and simplification in analysis. It uses some user defined methods and condition solvers for checking the condition generated by the source code program symbolic execution tree.

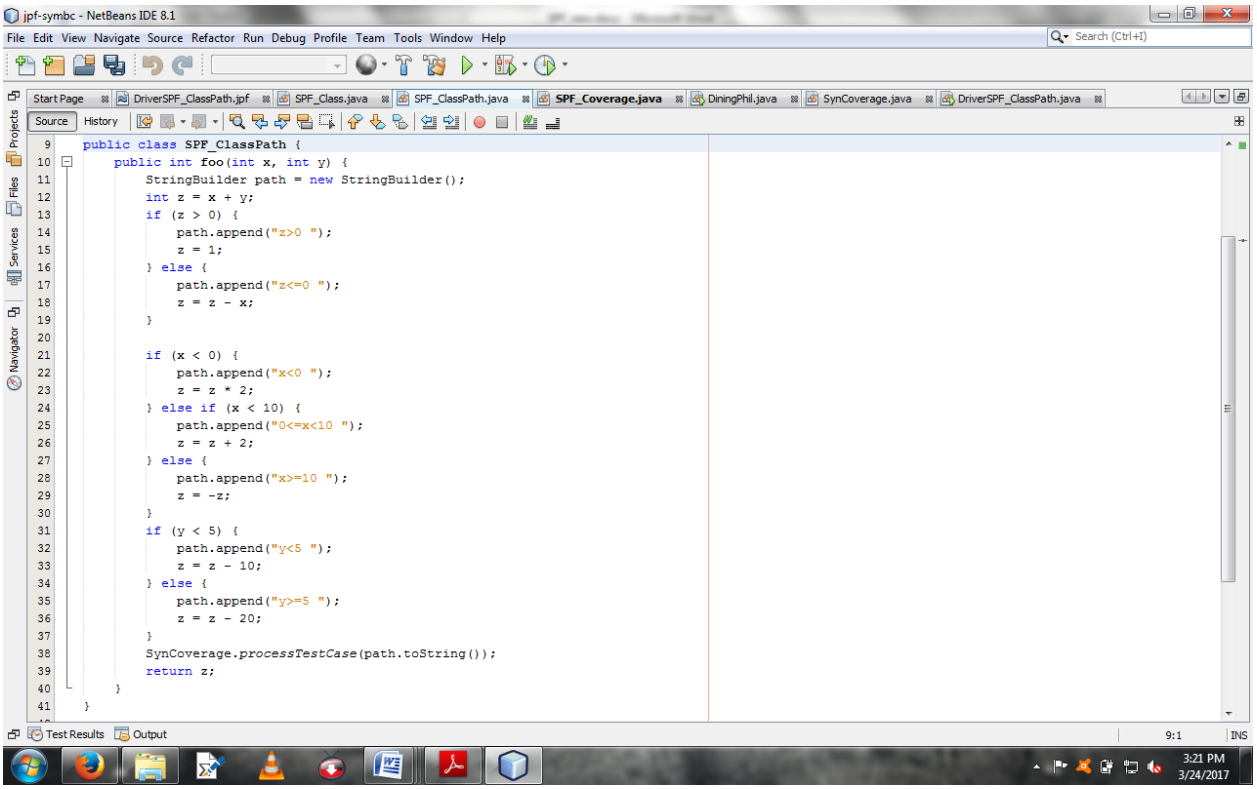
Basically, in SPF [15] the job of choice generator is for implementing the non-deterministic choices. The listener's job is to print the results of the analysis made by the symbolic execution. SPF thus uses some unique peers for modeling issues.



Program. 3: The source code for coverage criteria (Synchronization coverage) of Program-1

The coverage criterion is the criteria to cover the synchronization among the threads in a concurrent program. Basically, there are different types of coverage criteria are present in case of concurrent program. In program-3, it provides the covering of synchronization in program-1 is given in the above snapshot.

The path creation through Symbolic Execution is required for building the path. So, the Class file with path builder technique is given below in Program-4.



Program. 4: The source code for generating Path Builder for Program-1

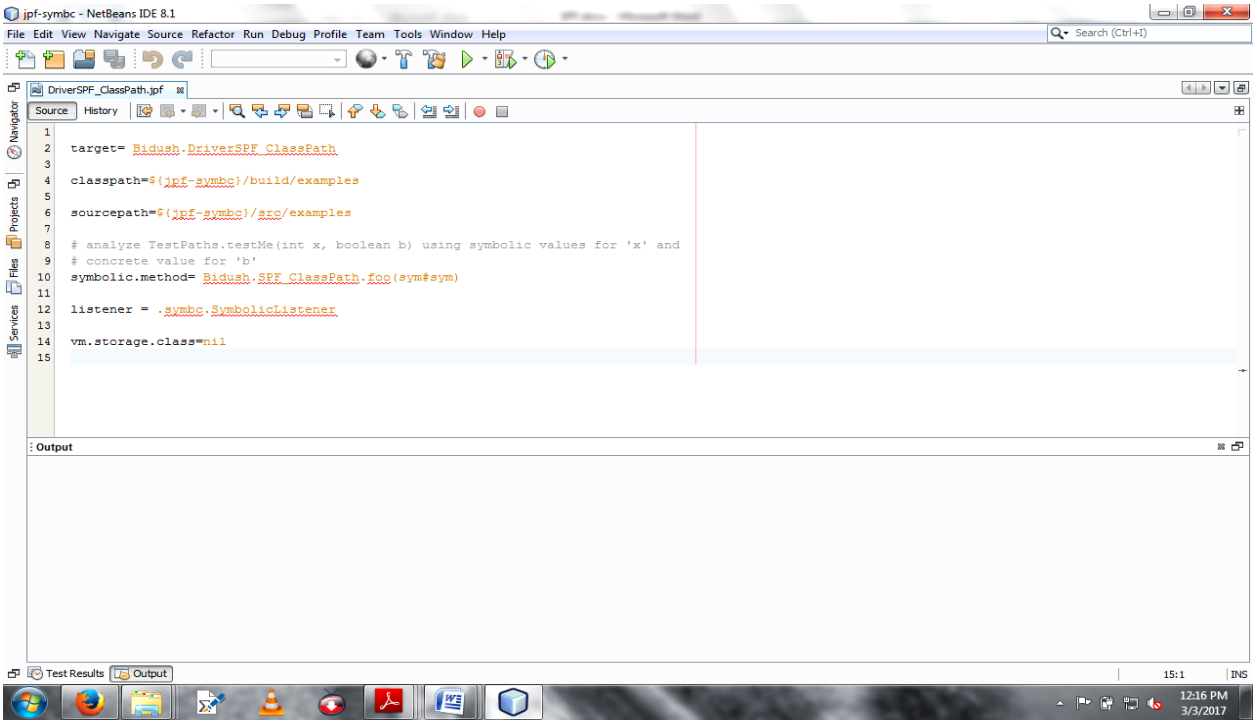
The driver file required for the path builder is given below in Program-5.

```
public class  DriverSPF_ClassPath {// The test driver
   public static void main(String[] args)
            {
     SPF_ClassPath x = new SPF_ClassPath();
     x.foo(2, 5);}}
```

Program-5: The driver file of the path builder program

3.4. Experimental Result

Symbolic path finder (SPF) The configuration file (.jpf) is the file which contains all the configurations required to verify the concurrent defects present in the program. It is given below in program-6.



Program. 6: The source code for Configuration file

The output after verification of the configuration file in SPF is shown in Fig-5 and Fig-6. As this is a large figure, it is shown in two parts.

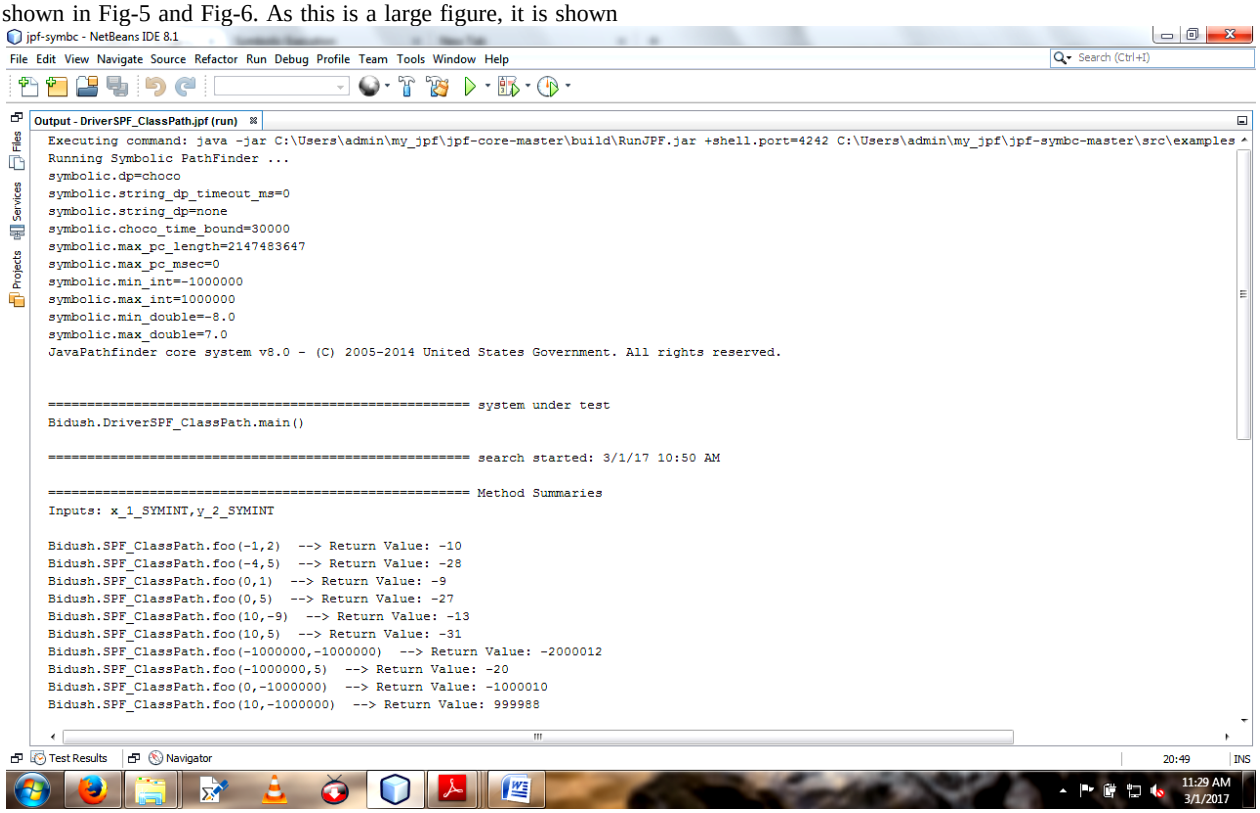


Fig. 5: Output showing the verification result of Program-1(part-1)

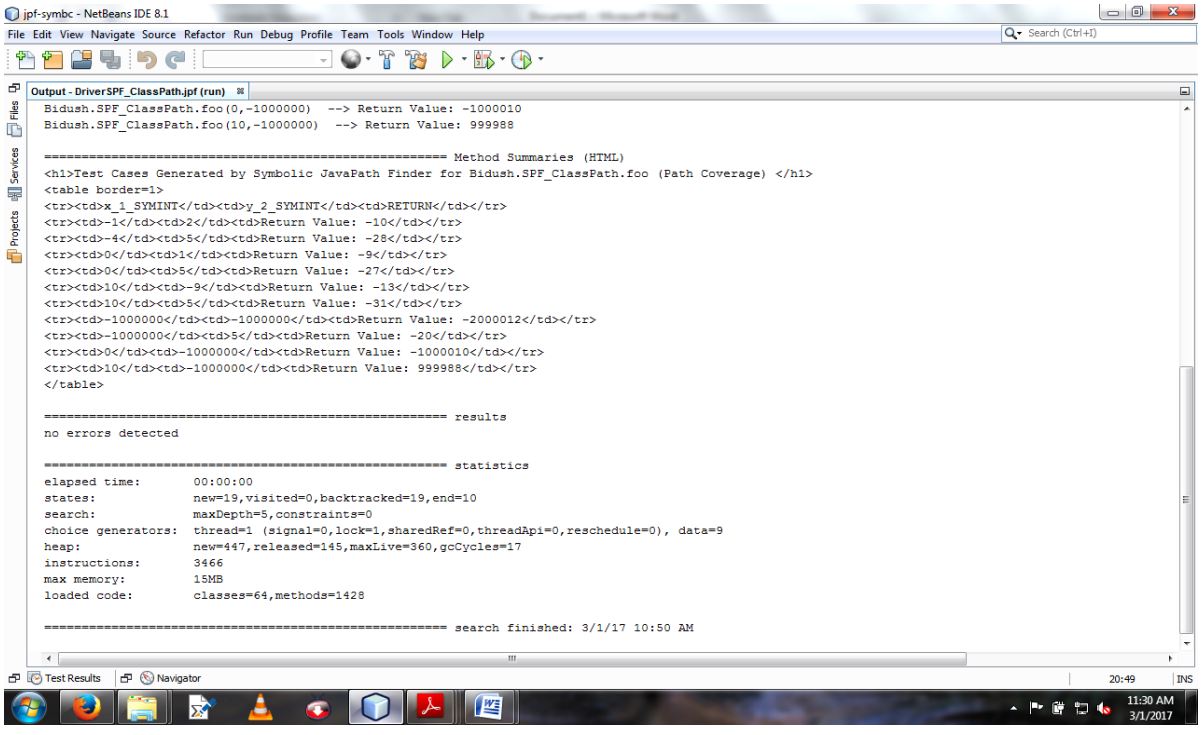


Fig. 6: Output showing the verification result of Program-1(part-2)

It provides the symbolic inputs as X_1 and Y_1, which is used in creating the symbolic execution tree. Fig-5 shows the generated test inputs along with the outputs. It provides some predefined limitations in the initial lines by Max_pc_length, Max_int etc. Next, it generates the test inputs like (-1, 2), (-4, 5) etc. which gives the return values. As there are no concurrent errors in Program-1, no error is detected in Fig-6. In the below displayed figure Fig-7, a concurrent program is given. The concur-

rent defects specified in Fig-8 shows its presence and which type of defect is it. Here we can identify the error occurrences and the reason behind it.

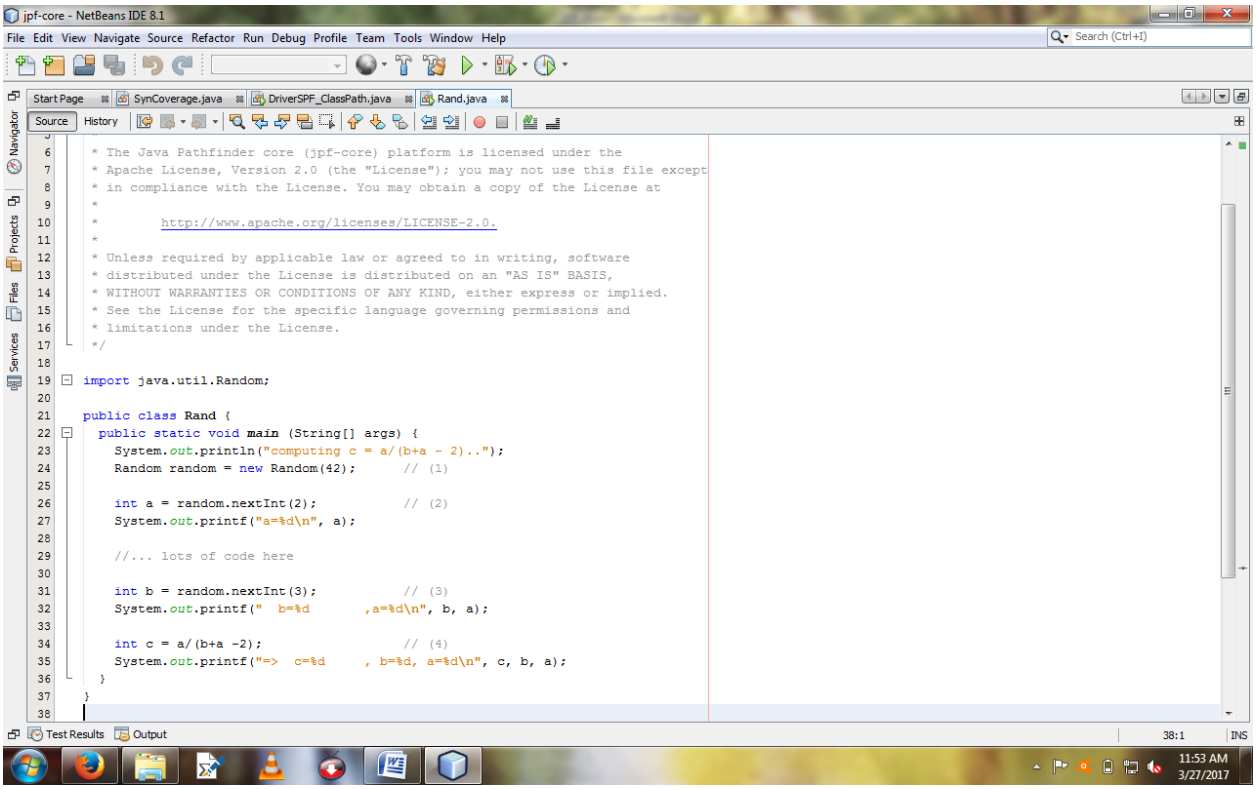


Fig. 7: The source code of a concurrent program with concurrent defects

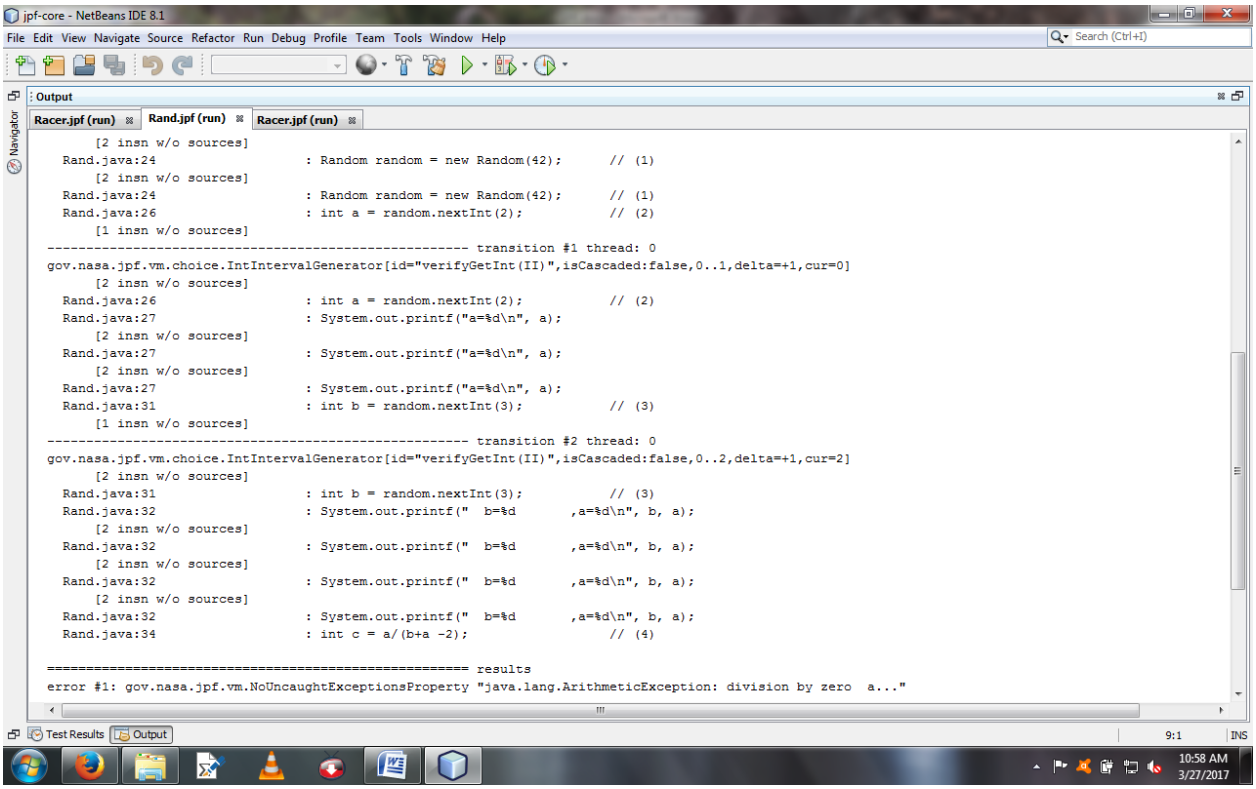


Fig. 8: Verification result of program in Fig-7

4. Experimental Result Discussion

In this implementation we got 10 numbers of test cases. The test inputs can be changed and the outputs can be achieved differently. The synchronization coverage criterion ([16], [17]) holds the interleaving mechanism for transferring the control to different conditions. There is total 3466 no. of instructions in the program. It covers 90 percent in synchronization coverage where as in interleaving coverage it covers 70 percent. It covers 100 percent path coverage. It checks for all infeasible paths and the range of the test inputs.

5. Conclusion

The SPF is a better tool in comparison with other model checking tools in case of synchronization coverage. In comparison with different coverage criteria, synchronization coverage provides more coverage in SPF. As it provides Symbolic execution tree, so the path conditions are easily diagnosed. The SPF in concurrent program will help in checking the different errors like race condition, deadlock or synchronization problem etc. In Future, we will try to modify the SPF in some extent so as to provide different types of coverage criteria. It can be also modified to detect other errors which cannot be handled by other tools.

References

[1] Asadollah, Sara Abbaspour, Hans Hansson, Daniel Sundmark, and Sigrid Eldh., "Towards classification of concurrency bugs based on observable properties", (2015) *In Complex Faults and Failures in Large Software Systems (COUFLESS), IEEE/ACM 1st International Workshop on, pp. 41-47, IEEE.*

[2] Melo, Silvana M., Paulo SL Souza, and Simone RS Souza., "Towards an empirical study design for concurrent software testing", (2016) *In Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE), Fourth International Workshop on, pp. 49-49. IEEE.*

[3] Choudhary, Ankit, Shan Lu, and Michael Pradel., "Efficient detection of thread safety violations via coverage-guided generation of concurrent tests", (2017) *In Proceedings of the 39th International Conference on Software Engineering, pp. 266-277. IEEE Press.*

[4] Yu, Tingting, Wei Wen, Xue Han, and Jane Huffman Hayes., Predicting Testability of Concurrent Programs, (2016) *In Software Testing, Verification and Validation (ICST), IEEE International Conference on, pp. 168-179. IEEE.*

[5] Chen, F., Rosu, G., "Parametric and sliced causality", (2007) *In: Computer Aided Verification, Springer, pp. 240–253, LNCS 4590.*

[6] Serbanuta, T.F., Chen, F., Rosu, G., "Maximal causal models for multithreaded systems", (2008) *Technical Report UIUCDCS-R-2008-3017, University of Illinois at Urbana-Champaign.*

[7] Bianchi, Francesco, Alessandro Margara, and Mauro Pezze., "A Survey of Recent Trends in Testing Concurrent Software Systems", (2017) *IEEE Transactions on Software Engineering.*

[8] Melo, Silvana M., Paulo SL Souza, and Simone RS Souza., "Towards an empirical study design for concurrent software testing", (2016) *In Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE), Fourth International Workshop on, pp. 49-49. IEEE.*

[9] Guo, Shengjian, Markus Kusano, and Chao Wang., "Conc-iSE: Incremental symbolic execution of concurrent software", (2016) *In Automated Software Engineering (ASE), 31st IEEE/ACM International Conference on, pp. 531-542. IEEE.*

[10] Lahiri, S., Qadeer, S., "Back to the future: revisiting precise program verification using SMT solvers", (2008) *In: Principles of Programming Languages, ACM, pp. 171–182.*

[11] Dutertre, B., de Moura, L., "A fast linear-arithmetic solver for dpll(t)", (2006) *In: Computer Aided Verification, Springer, pp. 81–94, LNCS 4144.*

[12] Metzler, Patrick, Habib Saissi, Péter Bokor, and Neeraj Suri., "Quick verification of concurrent programs by iteratively relaxed scheduling", (2017) *In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pp. 776-781. IEEE Press.*

[13] Wang, C., Yang, Z., Kahlon, V., Gupta, A., "Peephole partial order reduction", (2008) *In: Tools and Algorithms for Construction and Analysis of Systems, Springer, pp. 382–396 LNCS 4963.*

[14] Terragni, Valerio, and Shing-Chi Cheung., "Coverage-driven test code generation for concurrent classes", (2016) *In Software Engineering (ICSE), IEEE/ACM 38th International Conference on, pp. 1121-1132. IEEE.*

[15] Musuvathi, M., Qadeer, S., "CHESS: Systematic stress testing of concurrent software", (2006) *In: Logic-Based Program Synthesis and Transformation, Springer, pp. 15–16 LNCS 4407.*

[16] Lal, A., Reps, T.W., "Reducing concurrent analysis under a context bound to sequential analysis", (2008) *In: Computer Aided Verification, Springer, pp. 37–53, LNCS 5123.*

[17] Wang, Jie, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei., "A comprehensive study on real world concurrency bugs in Node. Js", (2017) *In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, pp. 520-531. IEEE Press.*