Int J Syst Assur Eng Manag (December 2023) 14(6):2345–2353
https://doi.org/10.1007/s13198-023-02081-1

ORIGINAL ARTICLE

Concurrent fault localization using ANN

Debolina Ghosh[1] · **Jay Prakash Singh**[2] · **Jagannath Singh**[3]

Received: 4 November 2022 / Revised: 3 July 2023 / Accepted: 25 July 2023 / Published online: 19 September 2023

Abstract The software is becoming more capable of providing better solutions to our day-to-day activities. In order to increase performance, concurrent programs are always preferred. But the concurrency produces obstacles to different phases of software development, including testing and debugging. Finding faults in a concurrent program is always a challenging task due to the presence of many threads overlapping each other, problems in sharing memories, etc. In this paper, we have proposed a Back propagation neural network (BPNN) to generate ranks for each class of a given program. These ranks indicate the probability of a fault being present in each class. The model is trained using test case coverage data, and it is tested using virtual test cases. In all three case studies, the fault localization technique assigned the highest rank to the classes where the actual faults were implanted. By using BPNN and training the model with test case coverage data, the technique shows promising results in identifying the classes where faults are likely to occur. This can greatly aid in the testing and debugging processes of concurrent programs, improving their overall performance.

Keywords Fault localization · Artificial neural network · Back propagation · Concurrent programs · Concurrent fault localization · Benchmark programs

✉ Debolina Ghosh
debolina442@gmail.com

Jay Prakash Singh
jaykiit.research@gmail.com

Jagannath Singh
jagannath.singhfcs@kiit.ac.in

[1] Department of Information Technology, Manipal University Jaipur, Jaipur, Rajasthan, India

[2] Department of Computer Science and Engineering, Manipal University Jaipur, Jaipur, Rajasthan, India

[3] School of Computer Engineering, KIIT Deemed to be University, Bhubaneswar, Odisha, India

1 Introduction

Software fault localization is the process of identifying the specific location or locations in the source code of a software program where a fault or error is likely to have occurred. This process is an important step in software debugging, as it helps developers narrow down the root cause of a problem and fix it more efficiently. Localizing faults is a more tedious task than testing (Chauhan 2010). Traditional fault localization techniques, as found in the literature, perform well for sequential programs. Traditional fault localization techniques are commonly used in software development and debugging to identify the location of faults in software programs. These techniques do not rely on advanced statistical analysis or machine learning algorithms but rather use heuristics or manual methods to identify suspicious code locations (Zakari et al. 2020; Shaikh et al. 2023). A few modern techniques have been introduced to improve the performance of testing techniques (Shaikh et al. 2021, 2022, 2021; Almomani et al. 2015).

Spectrum-based fault localization (SBFL) is a widely used technique for identifying the location of faults in software programs (Sarhan and Beszédes 2022; Jafarzadeh et al. 2022). It is based on the idea that the behavior of a faulty program during execution, as reflected in the outcomes of test cases, is different from the behavior of a correct program. Spectrum-based fault localization techniques use information about the outcomes of program executions, such as passing or failing test cases, to identify suspicious code locations that are likely to contain the fault. The performance

of existing fault localization techniques is not satisfactory for concurrent programs (Shaikh et al. 2020).

In concurrent programs, identifying the location of bugs or faults is more difficult as the faults occur during rum time (Bianchi et al. 2017). Fault detection in concurrent programs can be a challenging task due to the inherent complexity of concurrent execution. Concurrent programs are designed to run multiple threads or processes concurrently, often sharing resources and interacting with each other, which can lead to various types of faults or errors. Though the Spectrum-based Fault Localization (SBFL) technique can work for finding concurrent faults, the results are not satisfying (Ghosh and Singh 2020). Hence, there is a need for a new fault localization technique that can locate the faults or bugs present in concurrent programs.

In this paper, we have focused on identifying the faults present in concurrent program. Due to the uncertainty present in the concurrent program such as atomicity violations, thread interleaving, memory access patterns, etc., identifying the faults at the statement level of the concurrent program is very challenging. Therefore, we have tried to consider the function or branch level in our proposed approach to locating the fault in the program. In our approach, We have considered one test case at a time for the given concurrent program, then find the result of the test case i.e., pass or fail, and the function or branch coverage information against that particular test case. These information are considered as inputs for the back-propagation neural network. After tuning, the model generates a suspiciousness score for each branch or function. The branch or function with a higher suspiciousness score is more likely to be faulty. Hence, the proposed technique helps the programmer to find the faulty branch so that only the statements of the faulty branch need to be examined.

The other sections of this paper are organised as follows: In Sect. 2, we discuss some basic concepts that are required to understand the proposed technique. Closely related works are described in Sect. 3. Section 4, represents our proposed technique, and implementation details are presented in Sect. 5. Finally, Sect. 7 concludes the paper.

2 Basic concepts

In this section, we explain the basics of Concurrency, faults in concurrent programs, and a brief idea of an artificial neural network and its advantages in fault localization.

2.1 Concurrency in programming

Concurrency in programming refers to the ability of a program to execute multiple tasks simultaneously, or concurrently, rather than sequentially (Roscoe 1998). It allows a program to make progress on multiple tasks at the same time, even on a single processor system, by interleaving the execution of tasks. Concurrency is an important concept in modern programming as it enables efficient utilization of system resources, improves performance, and enhances responsiveness in concurrent or parallel environments. Concurrency introduces challenges such as race conditions, deadlocks, and synchronization issues that need to be carefully managed to ensure the correct behavior of concurrent programs. Techniques like locks, semaphores, and atomic operations can be used to synchronize access to shared resources and prevent data races. Properly designed concurrent programs can significantly improve performance and responsiveness, leading to more efficient and scalable software systems (Singh and Mohapatra 2018).

2.1.1 Advantages of concurrency

1. Now a days, CPU's are multi-core. Any single-threaded program utilises only one core. So it doesn't affect CPU performance. But multi-threaded program use multiple cores of the CPU, so the performance of CPU utilization increases.
2. Multi-threaded programs allow a series of program to run concurrently. Hence, it increases the response speed of the program.

2.1.2 Disadvantages of concurrency

1. Concurrent program increase context switching, which has adverse effects on CPU execution.
2. Multi-threaded program may cause deadlock and starvation.
3. Concurrent programs may limit resources after execution.

2.2 Concurrency issues in software testing

Concurrency can introduce several challenges in software testing, as the concurrent execution of multiple tasks can impact the behavior and correctness of a software system. Faults or bugs in concurrent programs can occur during the thread interleaving in run-time (Angus et al. 1990). Faults in concurrent programs not only occur during normal scenarios of computation but also due to thread interleaving, concurrent data access, and shared memory access. Hence, finding the faults present in a concurrent program is a more challenging task compared to sequential programs (Tu et al. 2019).

2.3 Artificial neural network

An artificial neural network (ANN) is a type of computational model that is inspired by the structure and function of the human brain (Yegnanarayana 2009). It is a type of machine learning algorithm that is used for tasks such as pattern recognition, classification, regression, and optimization. ANNs are a key component of deep learning, which is a subset of machine learning that focuses on neural networks with multiple hidden layers, enabling them to learn complex representations of data. ANNs continue to be an active area of research, with ongoing efforts to improve their performance, interpretability, and robustness for various real-world applications. ANNs consist of interconnected nodes or neurons organized in layers that receive input data, process it through activation functions, and produce output predictions. An ANN specifically uses a learning algorithm that trains the dataset and modifies the weight of each neuron based on the error rate between actual and target output. After the model is trained, it can predict the new set of data (Guillod et al. 2020).

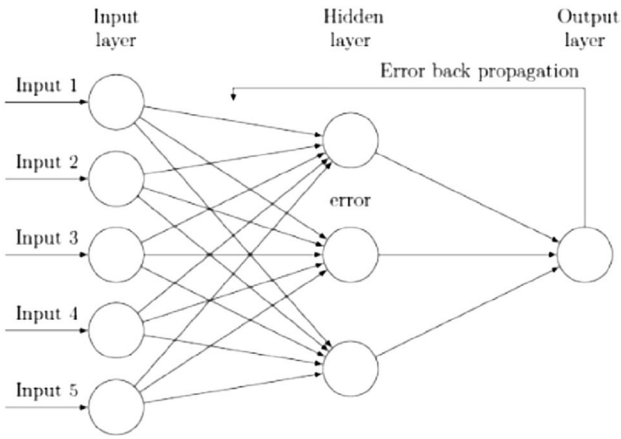

Fig. 1 Back propagation

2.3.1 Back propagation neural network

Backpropagation, short for “backward propagation of errors," is a widely used supervised learning algorithm used in artificial neural networks (ANNs) for training the network to make accurate predictions (Singh and Sahoo 2011). It is a method used to adjust the weights of connections between neurons in an ANN during the training process. Proper tuning of the networks decreases the rate of error and, hence, increases the reliability of the model. Figure 1 shows the working principle of the back propagation technique.

3 Literature review

In this section, we review some of the fault localization techniques present in the literature. We have reviewed some works related to the application of Machine learning for software fault localization and some works related to fault localization techniques in concurrent programs. A few related research works are presented in Table 1.

You et al. (2013) focused on the importance of the weights of failing and passing test cases to the similarity

Table 1 Literature survey

Author	Methodology	Results
You et al. (2013)	Modify the weights of failing and passing test cases	Modified weight values of failing test cases to are able to locate the faults more accurately
Chakraborty et al. (2018)	EnSpec: a new code entropy to spectrum based technique	makes the fault localization more robust and effective
Wong et al. (2011)	Radial basis function (RBF) based fault localization technique	The output of the network shows the suspiciousness of the each statement
Zheng et al. (2016)	Deep learning based fault localization technique	Results shows that only 10% statements need to be examine to find the buggy statement
Al Qasem and Akour (2019)	Fault prediction technique on the basis of Multi-layer perceptrons (MLPs) and Convolutional Neural Network (CNN)	Result shows that CNN outperforms the MLP
Park et al. (2010)	New dynamic fault localization technique Falcon	The tool generates the suspicious score of each data-access pattern
Park et al. (2015)	An unified fault localization tool UNICORN	Generates ranks of most important non-deadlock bugs
Koca et al. (2013)	Tool SCRUF which uses Spectrum-Based fault localization technique to find concurrency bugs	Calculate the suspicious score of each pattern in concurrent program
Alves et al. (2017)	Technique that finds the concurrent faults by using bounded model checking and code transformation process	The result shows that the tool can able to locate 84% faults present in the benchmark programs

coefficients in fault localization. They have used modified similarity coefficients and analyzed how failing test cases can have a greater contribution than passing test cases. Their study shows that modifying the weight values of failing test cases to similarity coefficients is able to locate the faults more accurately. Wong et al. (2016) provided a brief study on different fault localization techniques. According to their analysis, static and dynamic slicing techniques were popular from 2004 to 2007. But, since 2008, spectrum-based techniques have become more popular, as they have made a big contribution to automated fault localization. Their study also shows that different evaluation metrics such as EXAM score, T-Score, P-Score, N-Score, etc can be applied to identify the accurate faulty statement.

Chakraborty et al. (2018) proposed a new fault localization approach called EnSpec i.e., they have introduced a new code entropy to the spectrum-based technique. The code entropy is related to the faulty lines executed by the failing test cases. Hence, it makes fault localization more robust and effective. Kim et al. (2016) applied 32 different spectrum-based formulas, and based on the results, they clustered the similar categories into three different groups. To avoid the limitations of each group, a test case optimization technique is implemented. This technique increases the performance of the fault localization technique.

Wong et al. (2011) proposed a Radial Basis Function (RBF) based fault localization technique. The network is trained with coverage information and the results of the test cases. The output of the network shows the suspiciousness of each statement. They have also implemented different benchmark programs to evaluate the accuracy of their proposed technique. Zheng et al. (2016) proposed a deep learning based fault localization technique. The statement coverage information and the results of the test cases are used as input for the network. The output shows the suspiciousness of each statement. They have implemented different benchmark programs, and the results show that only 10% statements need to be examined to find the buggy statement.

Al Qasem and Akour (2019) proposed a software fault prediction technique on the basis of Multi-layer perceptrons (MLPs) and Convolutional Neural Network (CNN). They have compared the MLP and CNN, and the result shows that CNN outperforms the MLP. They have experimented with NASA datasets to check the accuracy of the proposed technique. Asadollah et al. (2016) provided a brief study based on concurrency and non-concurrency bugs. Their study shows that the concurrent bugs are different in terms of severity and fixing time. The study helps to develop debugging and testing related tools for concurrent software.

Park et al. (2010) developed a new dynamic fault localization technique, Falcon. The tool focuses on the data-access pattern among thread interleaving in concurrent programs. The tool generates a suspicious score for each data access pattern. Park et al. (2015) proposed a unified fault localization tool, UNICORN. It is an automated pattern detection based tool for finding non-deadlock concurrency bugs. It monitors the memory access patterns and generates a suspicious score for each pattern. Based on the score, it generates ranks of the most important non-deadlock bugs.

Koca et al. (2013) developed the tool SCRUF which uses Spectrum-Based fault localization techniques to find concurrency bugs. The tool instruments the different versions of the program that run on a particular thread interleaving pattern. Then it applies the SBFL technique to calculate the suspicious score of each pattern. Alves et al. (2017) proposed a technique that finds concurrent faults by using bounded model checking and code transformation process. The technique is implemented in Concurrent C programs or POSIX programs. The result shows that the tool can locate 84% of the faults present in the benchmark programs. Yu et al. (2016) presented testability testing for concurrent programs. A testability model was prepared by using test suite metrics and static metrics to predict the testability of concurrent programs.

Movassagh et al. (2021) presented an article to improve the precision of the perceptron neural network as well as train the neural network utilising meta-heuristic methods. The primary objective of this presentation is to extract as much useful information as possible from the UCI data in order to improve accuracy. In this research, the authors used meta-heuristic methods, to identify better NN coefficients. Alzubi et al. (2022) presented a novel strategy for detecting Android malware using machine learning. The Support Vector Machine (SVM) classifier and the Harris Hawks Optimisation (HHO) algorithm are the two components that make up the technique that has been suggested. To be more explicit, the function of the HHO method is to optimise the hyperparameters of the SVM classifier, while the SVM is responsible for doing the classification of malware based on the model that was determined to be the best fit, as well as producing the best possible result for how the features should be weighted.

4 Proposed approach

Locating faults in concurrent programs is more challenging than finding faults in sequential program. In concurrent programs, examining the source code at the statement level is not possible, as the execution sequence in concurrent program depends on thread interleaving during runtime. In this section, we discuss our proposed approach for identifying the faults present at branch or class level in concurrent program based on back propagation neural networks.

```
class ThreadClass extends Thread {
   private int number;
   //class constructor
   public ThreadClass(int number) {
      this.number = number;
   }
        //run method => execution code
        //for thread
   public void run() {
      int counter = 0;
      int numInt = 0;
      //prints the number till specified
        //number is reached, starting from 10
      do {
         numInt = (int) (counter + 10);
         System.out.println(this.getName()
 + " prints " + numInt);
         counter++;
      } while(numInt == number);
// Bug while(numInt != number);
      System.out.println("** Correct! " + this.getName() +
        "printed " + counter + " times.**");
   }
}
```

Fig. 2 Example program

```
public class MainThread {
   public static void main(String [] args) {
      System.out.println("Starting thread_1...");
      //create a thread class instance
      Thread thread_1 = new ThreadClass(16);
      //start the thread thread_1
      thread_1.start();
      try {
          //wait for thread_1 to die
         thread_1.join();
      } catch (InterruptedException e) {
         System.out.println("Thread interrupted.");
      }
   System.out.println("Starting thread_2...");
      Thread thread_2 = new ThreadClass(6);
      //start thread_2
      thread_2.start();
      System.out.println("main() is ending...");
   }
}
```

Fig. 3 Example program

4.1 Sample experiment

In this section we have taken one concurrent program as an example program for evaluating the accuracy of the proposed approach. For concurrent programs, we consider branch coverage of each class due to thread interleaving during runtime. We have considered a multithreaded program as a sample program. The program consists of two classes, presented in Figs. 2 and 3. In Fig. 2 ThreadClass executes the code for the thread and prints the number till the specified number is reached, starting from 10. It increments the counter, whereas MainThread creates a thread class instance and starts the thread as shown in Fig. 3.

Table 2 Branch coverage of each test cases for sample program

Test cases	MainThread	ThreadClass	Test-case result
T1	1	1	F
T2	1	2	P
T3	1	2	P
T4	1	2	P
T5	1	2	P
T6	1	1	F

Table 3 Virtual Testcases

v_1	1	0	.	0
v_2	0	1	.	0
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
v_n	0	.	0	1

Table 2 presents the coverage matrix for an example program, where the number of rows equals the number of test cases and the number of columns equals the number of classes or branches. The last column denotes the results of each test case, i.e., either pass or fail. The value of each row indicates the frequency of execution for a specific class or branch in the corresponding test case. Now, each class is covered by the test cases, either pass or fail. As example, MainThread is covered by two failed test cases, i.e., T1 and T6. A single row presents the coverage information for a particular test case, as shown in Table 2.

Now, we consider a set of virtual test cases $v_1....v_n$ as shown in Table 3. The speciality of a virtual test case is that it executes only one statement, which in reality is not possible, so it is called a virtual test case. Each row of this matrix is fed to the backpropagation network, and the model gives the suspiciousness score of each corresponding class or method as output.

4.2 Framework overview

Figure 4 represents the block diagram of the proposed fault localization technique.

Firstly, we generate the test cases for the concurrent program under test. We have considered a test suit to 6 test cases. Next, we find function or branch coverage and the result of the test cases, i.e., pass or fail after executing all the test cases.

Next, a backpropagation network is created with two input neurons, one output layer, and one hidden layer. Each class or branch works as input for the network. As our example

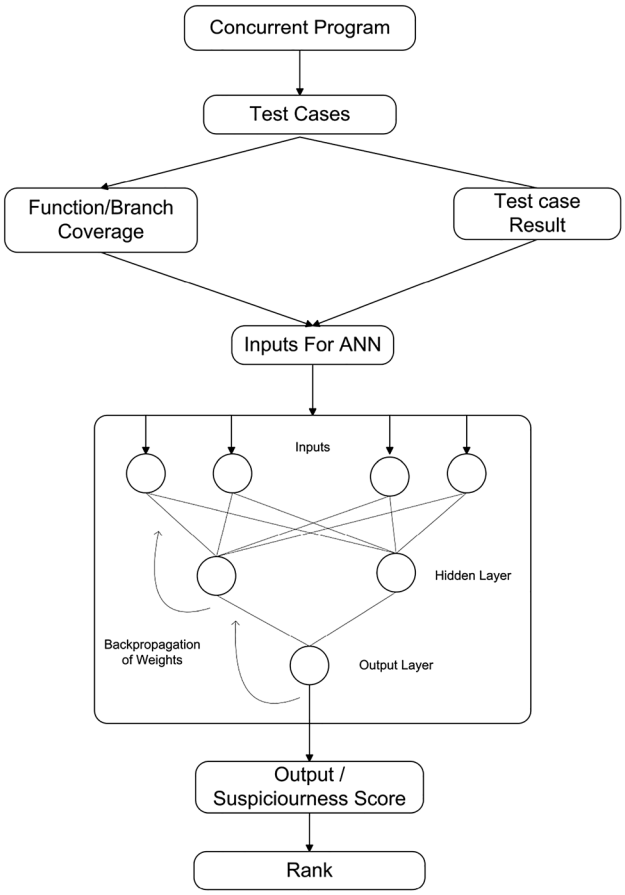

Fig. 4 Block diagram of proposed approach

program consists of two classes, we have two input layers. The collected coverage data and test case results are collectively used as input for the back-propagation network. The network is trained with collected coverage data. The first input for our example program is 1, 1 with 0 as the expected output. Fail test cases are represented as 0, and pass test cases are represented as 1. If the actual output is not matched with the expected output, then the weight of the network is modified, and the updated weight is considered for the next input. Hence, to train the network, the weight is continuously updated until the error is minimum, i.e., approx 0.001.

After training the Back-propagation network, the test data or virtual test cases are fed to the network. The generated output gives a suspiciousness score for each branch or function. Based on the suspiciousness score, the rank of each branch or function is generated. The branch with a higher suspicious score and a lower rank is more faulty.

In the example program, we found the suspiciousness score of two classes, i.e., MainThread and TreadClass as presented in Table 4. The suspiciousness score of ThreadClass is higher than that of MainTread. Hence, we can say that ThreadClass contains a faulty statement.

Table 4 Suspiciousness score

MainThread	ThreadClass
0.19	0.53

5 Implementation and result analysis

In this section, we present the implementation details of the proposed framework for finding the faults present in concurrent program. We discuss the detailed case studies considered for the experiment. Finally, we compare our fault localization approach with other closely related fault localization techniques.

5.1 Experimental setup

To execute the proposed approach, we have used a Linux based system with 32 GB of RAM and a 3 GHz Intel (R) i5 processor. An Eclipse IDE with JDK version 12 is used to develop an automated fault localization technique for concurrent programs.

5.2 Case study

In this section, we have used three Java open source benchmark programs to evaluate the accuracy of the proposed technique. The benchmark programs are downloaded from the SIR repository (Do et al. 2005). The programs are Account, Piper and Producer_Consumer having 66, 74 and 99 LOCs, respectively. The detailed information on the benchmark programs is presented in Table 5. We have introduced faults manually in each benchmark programs to check the accuracy of the proposed approach. Firstly, we have seeded a fault in Account class in our first case study, i.e., Account program. Likewise, we have introduced a fault manually to the Piper class and the Buffer class in the second and third case studies, i.e., the Piper program and Producer_Consumer program respectively.

5.3 Result analysis

In this section, we present the experimental results of the proposed approach. The suspiciousness scores of each class for each benchmark program are presented in Tables 6, 7 and 8.

Table 5 Detail information of Benchmark Programs

Programs	LOCs	Classes	Types of concurrency
Account	66	3	Deadlock, race
Piper	74	4	Deadlock
Producer_Consumer	99	8	Race

Table 6 Suspiciousness score of each class of account program

Class	Score	Rank
Main	0.22	3
ManageAccount	0.34	2
Account	0.66	1

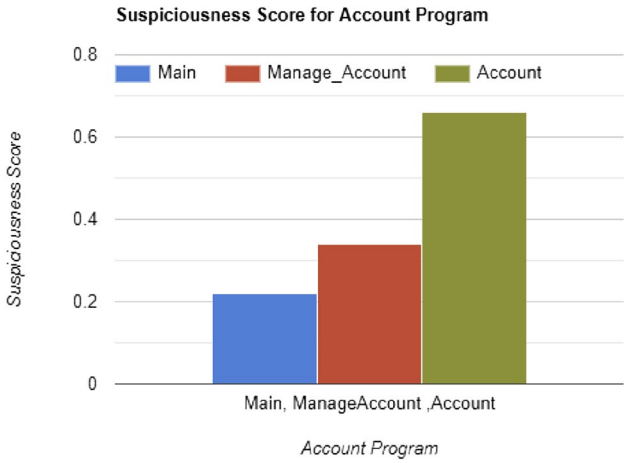

Fig. 5 Score of account programs

From Table 6 and Fig. 5, we find that the suspiciousness score of the account class is the highest. As mentioned in Sect. 5.2, we have seeded a fault in the Account class, and from the result, we can verify that the Account class contains the fault. Likewise, for the Piper program, from Table 7 and Fig. 6, we can confirm that the class Piper has the highest probability of containing faults, as faults were seeded manually into the Piper class initially. From Table 8 and Fig. 7, we can verify that the buffer class has the most suspicious score of being a faulty statement, and the HaltException class stands in second.

From the result analysis, we can say that the proposed model gives satisfactory results in fault finding. To verify the accuracy of our model, we have initially introduced faults in a class of each benchmark program, and after training, the model generates the highest suspiciousness score for that particular class. Hence, the programmer needs to examine only the suspected class to locate the fault that occurred during concurrency. So, the proposed approach makes the debugging process easier and saves time in finding faults in concurrent programs.

5.4 Comparison with related work

We have compared our proposed technique with the widely used spectrum-based metric, D-Star. We have executed our benchmark programs with D-Star. As a result, we find D-Star generates a suspiciousness score of 4.8 for all three

Table 7 Suspiciousness score of each class of piper program

Class	Score	Rank
IBM_AirlinesProducer	0.27	3
Piper	0.57	1
IBM_Airlines.Consumer	0.31	2
IBM_Airlines	0.24	4

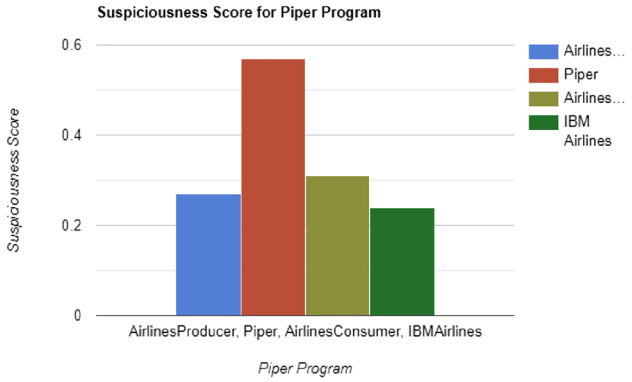

Fig. 6 Score of piper programs

Table 8 Suspiciousness score of each class of Producer_Consumer program

Class	Score	Rank
Producer	0.17	7
Attribute	0.23	6
HaltException	0.42	2
Buffer	0.59	1
ProducerConsumer	0.35	3
AttrData	0.24	5
Consumer	0.28	4

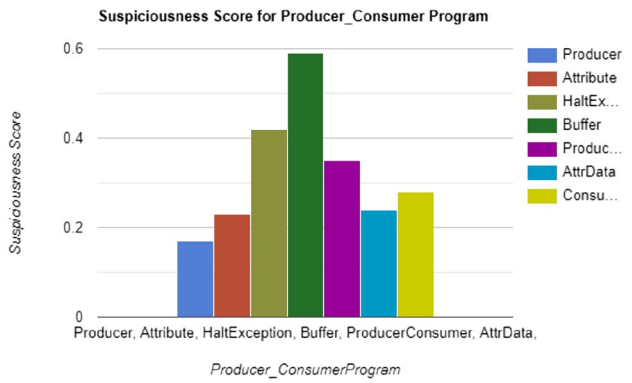

Fig. 7 Score of Producer_Consumer programs

Table 9 Account program

Class	D-Star score	Proposed approach score
Account	4.8	0.22
Main	4.8	0.34
ManageAccount	4.8	0.66

Table 10 Piper program

Class	D-Star score	Proposed approach score
IBM_Airlines	5.2	0.27
Piper	5.2	0.57
IBM_Airlines.Consumer	5.2	0.31
IBM_Airlines	5.2	0.24

classes for our first case study, i.e., the Account program, as shown in Table 9. Hence, it is very difficult to find the exact faulty class, whereas our proposed approach assigns rank 1 to the Account class, which contains the actual fault.

In the same way, D-star generates a suspicious score of 5.2 for all four classes for the second case study, i.e., Piper program, as presented in Table 10. But, the proposed technique assigns rank 1 to the faulty class Piper. Similarly, for our third case study, D-Star generates an 8.3 suspiciousness score for the Buffer class, Consumer class, HaltException class and Produce_consumer class and 0 for the rest of the classes. But, the proposed technique is able to identify the faulty class, i.e., Buffer and assign rank as 1. Hence, we can say that our proposed technique gives a more accurate result compared to the widely used D-Star technique.

6 Threats to validity

Here, we present some of the limitations of our tool-

1. In this work, we have considered Back-propagation technique to find concurrent faults that occur during program execution. The efficiency of the proposed technique needs to be evaluated for advanced machine learning models.
2. We have experimented with each concurrent program having 3 to 7 classes. For large-scale concurrent programs, the efficiency of the proposed technique needs to be evaluated.

7 Conclusion

In this paper, a fault localization technique is specifically designed for concurrent programs. Locating faults at the statement level in concurrent programs is challenging, therefore, we focus on identifying faulty classes using a back-propagation technique. The technique involves training a model using coverage data obtained from executing test cases. The model is then fine-tuned, and it generates suspiciousness scores for each class as its output. Classes with higher suspiciousness scores are more likely to contain faults. This approach narrows down the search space for the programmer, allowing them to focus their examination on the suspicious classes to identify the faulty class. The results from the case studies conducted in the paper demonstrate the effectiveness of the proposed technique. In the first case study, the programmer only needs to examine 33% of the code to find the fault. Similarly, for the second and third case studies, the examination effort is significantly reduced to 0.25% and 0.12% of the code, respectively. In the future, we plan to apply the proposed technique to large real-world programs, which will further validate its effectiveness. Additionally, we intend to explore advanced machine learning techniques or deep learning models to enhance the accuracy of the fault localization tool.

Funding There was no outside funding or grants received that assisted in this study.

Declarations

Conflict of interest The authors affirm that they have no conflicts of interest and have addressed all ethical concerns, including those involving human or animal engagement. Such consent is not relevant.

References

Al Qasem O, Akour M (2019) Software fault prediction using deep learning algorithms. Int J Open Source Softw Process (IJOSSP), IGI Global 10(4):1–19

Almomani O, Al-Shugran M, Alzubi JA, Alzubi OA (2015) Performance evaluation of position-based routing protocols using different mobility models in manet. Int J Comput Appl 119(3)

Alves EHdS, Cordeiro LC, Eddie Filho BdL (2017) A method to localize faults in concurrent c programs. In: J Syst Softw Elsevier vol 132, pp 336–352

Alzubi OA, Alzubi JA, Al-Zoubi AM, Hassonah MA, Kose U (2022) An efficient malware detection approach with feature weighting based on harris hawks optimization. Cluster Comput 1–19

Angus IG, Fox GC, Kim JS, Walker DW (1990) Solving problems on concurrent processors, vol 2. Prentice-Hall, Inc., New York

Asadollah SA, Sundmark D, Eldh S, Hansson H, Enoiu EP (2016) A study of concurrency bugs in an open source software. In: IFIP

International conference on open source systems, pp 16–31. Springer
Bianchi FA, Margara A, Pezzè M (2017) A survey of recent trends in testing concurrent software systems. IEEE Trans Softw Eng IEEE 44(8):747–783
Chakraborty S, Li Y, Irvine M, Saha R, Ray B (2018) Entropy guided spectrum based bug localization using statistical language model. arXiv preprint arXiv:1802.06947
Chauhan N (2010) Software testing: principles and practices. Oxford University Press, Oxford
Do H, Elbaum SG, Rothermel G (2005) Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empirical Softw Eng Int J 10(4):405–435
Ghosh D, Singh J (2020) Spectrum-based fault localization for concurrent programs. In: 2020 international conference on computer science, engineering and applications (ICCSEA), pp 1–5. IEEE
Guillod T, Papamanolis P, Kolar JW (2020) Artificial neural network (ann) based fast and accurate inductor modeling and design. IEEE Open J Power Electron IEEE 1:284–299
Jafarzadeh N, Jalili A, Alzubi JA, Rezaee K, Liu Y, Gheisari M, Sadeghi Bigham B, Javadpour A (2022) A novel buffering fault-tolerance approach for network on chip (noc). IET Circ Dev Syst
Kim J, Park J, Lee E (2016) A new spectrum-based fault localization with the technique of test case optimization. J Inf Sci Eng Citeseer 32(1):177–196
Koca F, Sözer H, Abreu R (2013) Spectrum-based fault localization for diagnosing concurrency faults. In: IFIP international conference on testing software and systems, pp 239–254. Springer
Movassagh AA, Alzubi JA, Gheisari M, Rahimi M, Mohan S, Abbasi AA, Nabipour N (2021) Artificial neural networks training algorithm integrating invasive weed optimization with differential evolutionary model. J Ambient Intell Hum Comput 1–9
Park S, Vuduc R, Harrold MJ (2015) Unicorn: a unified approach for localizing non-deadlock concurrency bugs. Softw Test Verif Reliab Wiley Online Library 25(3):167–190
Park S, Vuduc RW, Harrold MJ (2010) Falcon: fault localization in concurrent programs. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering, Volume 1, pp 245–254. ACM/IEEE international conference on software engineering
Roscoe B (1998) The theory and practice of concurrency
Sarhan QI, Beszédes Á (2022) A survey of challenges in spectrum-based software fault localization. IEEE Access, IEEE 10:10618–10639
Shaikh MS, Ansari MM, Jatoi MA, Arain ZA, Qader AA (2020) Analysis of underground cable fault techniques using matlab simulation. Sukkur IBA J Comput Math Sci 4(1):1–10
Shaikh MS, Hua C, Jatoi MA, Ansari MM, Qader AA (2021) Parameter estimation of ac transmission line considering different bundle conductors using flux linkage technique. IEEE Can J Electr Comput Eng 44(3):313–320
Shaikh MS, Hua C, Jatoi MA, Ansari MM, Qader AA (2021) Application of grey wolf optimisation algorithm in parameter calculation of overhead transmission line system. IET Sci Meas Technol 15(2):218–231
Shaikh MS, Hua C, Hassan M, Raj S, Jatoi MA, Ansari MM (2022) Optimal parameter estimation of overhead transmission line considering different bundle conductors with the uncertainty of load modeling. Opt Control Appl Methods 43(3):652–666
Shaikh MS, Raj S, Babu R, Kumar S, Sagrolikar K (2023) A hybrid moth-flame algorithm with particle swarm optimization with application in power transmission and distribution. Decision Anal J 6:100182
Singh J, Mohapatra DP (2018) Dynamic slicing of concurrent aspectj programs: an explicit context-sensitive approach. Softw Practice Exp Wiley Online Library 48(1):233–260
Singh J, Sahoo B (2011) Software effort estimation with different artificial neural network. Foundation of Computer Science, USA
Tu T, Liu X, Song L, Zhang Y (2019) Understanding real-world concurrency bugs in go. In: Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems, pp 865–878. ACM
Wong WE, Debroy V, Golden R, Xu X, Thuraisingham B (2011) Effective software fault localization using an rbf neural network. IEEE Trans Reliab 61(1):149–169
Wong WE, Gao R, Li Y, Abreu R, Wotawa F (2016) A survey on software fault localization. IEEE Trans Softw Eng IEEE 42(8):707–740
Yegnanarayana B (2009) Artificial neural networks. PHI Learning Pvt. Ltd, New Delhi
You Y-S, Huang C-Y, Peng K-L, Hsu C-J (2013) Evaluation and analysis of spectrum-based fault localization with modified similarity coefficients for software debugging. In: 2013 IEEE 37th annual computer software and applications conference, pp 180–189. IEEE
Yu T, Wen W, Han X, Hayes JH (2016) Predicting testability of concurrent programs. In: 2016 IEEE international conference on software testing, verification and validation (ICST), pp 168–179. IEEE
Zakari A, Lee SP, Abreu R, Ahmed BH, Rasheed RA (2020) Multiple fault localization of software programs: a systematic literature review. Inf Softw Technol Elsevier 124:106312
Zheng W, Hu D, Wang J (2016) Fault localization analysis based on deep neural network. In: Mathematical problems in engineering, Hindawi 2016