Comparison of Concurrent Program Behavior Using Java Interactive Visualization Environment

M. Shobitha, R. Prakash Sidharth, P. K. Sreesruthi, P. Varun Raj, and Jayaraman Swaminathan

Abstract It's important for software practitioners to understand mutual exclusion in different systems because most of the problems in concurrent systems boil down to achieve mutual exclusion. Mutual exclusion for different types of concurrent scenarios-multithreaded, parallel, distributed can be achieved in different ways by the constructs provided by the programming language. For each of the mentioned types, the performance (or behavior) varies in different ways. The performance of mutual exclusion algorithms is measured by mainly six metrics. This paper shows the comparison between the performance of a chosen synchronization algorithm by analyzing the sequence diagrams obtained from Java Interactive Visualization Environment (JIVE), a dynamic analysis framework for Java program visualization. This paper also presents the results and observations obtained after comparing dining philosophers problem in the above mentioned scenarios. The results are based on the metrics - Message complexity, Synchronization delay and Response Time. The analysis is done on low load performance.

Keywords Distributed mutual exclusion · Visualization · Comparison · Parallel · Analysis · Multithreading

1 Introduction

With technology rising on a never before seen scale, concurrent systems have been playing an increasingly prominent role in software development. Three forms of concurrent systems have been widespread over the years, namely multithreaded, parallel and distributed. While the multithreaded and parallel programs pertain to cooperating processes/threads on a single system, distributed programs span multiple systems, possibly spread over a large geographical area. It is a complex field that consists of devices that communicate and organize tasks to act as a single-coherent system. It

M. Shobitha · R. P. Sidharth · P. K. Sreesruthi (✉) · P. Varun Raj · J. Swaminathan
Department of Computer Science Engineering, Amrita Vishwa Vidyapeetham, Amritapuri, India
e-mail: swaminathanj@am.amrita.edu

S. Smys et al. (eds.), *Inventive Computation and Information Technologies*, Lecture Notes in Networks and Systems 336, https://doi.org/10.1007/978-981-16-6723-7_29

combines the power of multiple machines to improve the performance of a program with unmatched scalability. The only reason that distributed programming is yet to come to the forefront can be attributed to its increased complexity in deployment and maintenance.

Concurrent systems are characterized by some challenges which are entirely different and non-existent in the sequential systems. Although each of these systems throws its unique challenges, the fundamental theme underlying these challenges revolve around safety, liveness and fairness. Dealing with these challenges by techniques such as mutual exclusion and other forms of synchronization introduces second level challenges such as message complexity, response time and other overhead. However, despite the challenges and the overhead involved in concurrent systems, they continue to play a dominant part in software development, due to the performance and scalability benefits they provide.

In this paper, we have analyzed the performance of mutual exclusion in the case of three different concurrent programming paradigms, namely multithreaded, parallel and distributed programming. There are predominantly six metrics used for measuring the performance of mutual exclusion algorithms. These include message complexity, synchronisation delay, response time, system throughput, low and high load performance and best and worst case performance. Since our observations deal with low load performance, more emphasis has been given to message complexity, synchronization delay and response time.

The goal of this project is to directly contrast the performance of distributed programs against their parallel and multithreaded equivalent using sequence diagrams obtained from Java Interactive Visualization Environment (JIVE) and further compare them using the aforementioned three metrics. This analysis has been carried out using the dining philosophers problem in all three programming implementations. The further observations and results based on this analysis are specified in the upcoming sections with an array of scope for future work.

The rest of the paper is structured as follows. Section 2 discusses some of the closely related work. Section 3 describes the implementation of multithreaded, parallel and distributed versions of the dining philosophers problem. Section 4 provides the analysis of the different concurrent versions providing key insights into their implementations. Section 6 summarizes the work and provides directions for future work.

2 Related Work

In this section, some closely related works of previous papers and their contribution to the concurrent program analysis are summarized. Concurrency around the scenarios-multithreaded, parallel and distributed systems are discussed. This is then, followed by the significance of JIVE-an interactive visualization tool used for the analysis.

Dining Philosophers problem [1] is addressed to acknowledge the working of concurrent programs. Java Threads [2] is used for the implementation of multi-threaded programming. MapReduce and Fork/Join are popular java based frameworks for achieving parallelism. In the dining philosophers problem, there are only five philosophers. Hence, the Fork/Join framework is a better choice for parallel program implementation [3, 4]. RMI framework [5] is used to build remote communication between Java programs that help to build distributed applications. But, all these frameworks have a drawback that they can't access shared objects at the same time, which makes synchronization [6] a vital part in the implementation of the concurrent program.

Sequence diagrams are interaction diagrams that record the relationships between objects when the programs work collaboratively. Object-to-object interaction of a program using sequence diagrams makes it easy to understand the workflow and the complexity of the program [7–9]. Sharp and Rountev [10] explain sequence diagrams and their working principles. Various limitations of the UML Sequence diagrams are pointed out and a set of techniques to overcome this limitation is highlighted. UML diagrams are extremely large and clustered making them hard to interpret properly. So the earlier unreadable UML sequence diagram is expanded upon to interactively explore different aspects of the diagram, to focus on subsets of the expressed behavior. Thus, the earlier sequence diagram can be elaborated for better understandability and proper interpretation by the programmer.

The sequence diagram of Java programs can be obtained through the Eclipse plugin tool Java Interactive Visualization Environment [11]. JIVE visually portrays both the call history and the runtime state of a program. The call history can be seen in the sequence diagram, where each execution thread is depicted in a different color, simplifying the object interactions. JIVE also generates runtime models, verifies and validates those models against design time models [12]. Ajaz et al. [13] already presented the performance study of a parallel program using JIVE on multicore systems. Kishor et al. [12] emphasize a methodology to interpret the sequence diagram in the form of a finite state diagram. The key state variables are annotated by the user. This is later combined with the execution trace to obtain the sequence diagram. Since state diagrams are smaller in magnitude compared to sequence diagrams, they provide more insight into program behavior and detect subtle errors.

3 Program Implementation

To shed light and raise discussion on concurrency, the dining philosophers algorithm has been implemented in this paper [14]. The dining philosophers problem is a classical problem of synchronization. Five philosophers sit around a circular table. Each of them has two states - thinking and eating. Five forks are placed on the table. In order to eat, the philosopher must have two forks in hand. No two adjacent philosophers can eat simultaneously.

3.1 Multithreaded Programming

To start the thread, the start() method must be invoked. Java provides Thread class to implement multithreaded programming. Thread class provides various methods to create and perform operations on a thread. One of the methods to create a thread process in Java, using Threads, is by extending the Thread class and overriding its run() method.

```
public void run(){
    while(true){
        thinking();
        //Philosopher gets hungry
        fork.take();
        eating();
        fork.release();
    }
}
```

Code 1: run() method of the Philosopher class

Since in the dining philosophers problem, no two adjacent philosophers can get hold of both the forks simultaneously, it's important to have synchronization in the availability of the forks. When several threads are trying to access a common resource, it is necessary to have control over who has access. And this is what synchronization does. In the multithreaded program, two synchronized methods, namely take() and release(), are used. The synchronized methods lock an object for any shared resource; in this scenario - the forks. The message complexity, the number of messages required for the execution of the critical section, is 2(N-1), where N represents the number of philosophers.

```
synchronized void release(){
    Philosopher philosopher =  (Philosopher)
       Thread.currentThread();
    int number = Philospher.Number;
    fork[number] = false;
    fork[(number+1)%5] = false;
    notifyAll();
}
```

Code 2: synchronized method - release()

```
synchronized void take(){
    Philosopher philosopher =  (Philosopher)
       Thread.currentThread();
    int number = Philospher.Number;
    while(fork[number] || fork[(number+1)%5]){
        try{
            wait();
        }
        catch(InterruptedException e){}
    }
    fork[number] = true;
    fork[(number+1)%5] = true;
}
```

Code 3: synchronized method - take()

The program begins by initializing the threads of five philosophers. The threads start executing the run() method, where the philosopher has to go to the thinking state for a random period and then go to the eating state. The mutual exclusion part will be handled by the two synchronized methods, while taking and releasing the forks.

3.2 Parallel Programming

Fork/Join framework is a framework in Java that sets up and executes parallel programs by taking advantage of multiple processors, which is accomplished by identifying the availability of processor cores and allocating the tasks accordingly. It uses a divide-and-conquer strategy: divide a very large problem into smaller parts. These smaller parts can be further divided into even smaller ones, recursively until a part can be solved directly. In the parallel program, ForkJoinTask is used. Through this mechanism, a small number of actual threads in ForkJoinPool controls a large number of tasks to be executed. ForkJoinTask.invokeAll() method combines fork() and join() in a single call and starts the instances of all the philosophers.

```
for (int i = 0; i < philosophers.length; i++) {
     Object leftFork = forks[i];
     Object rightFork = forks[(i + 1) % forks.length];
     philosophers[i] = new Philosopher(leftFork,
        rightFork);
     subtasks.add(philosophers[i]);
}
ForkJoinTask.invokeAll(subtasks);
```

Code 4: using invokeAll() method

To attain mutual exclusion, the concept of the synchronized block is used. For any shared resource, the synchronized block is used to lock an object. Nested synchronized blocks help to get hold of both the forks the philosopher needs. So, the message complexity becomes 4(N-1). The instances of all the five philosophers are invoked by ForkJoinTask.invokeAll() method. The instances then run the compute() method, where the synchronized block is defined. Thus, mutual exclusion and concurrency are achieved.

```
synchronized (leftFork) {
  synchronized (rightFork) {
    //eating
    gotForks(leftFork,rightFork);
    try {
        TimeUnit.MILLISECONDS.sleep((int)(Math.random()*50));
    } catch (InterruptedException e) {}
  }
}
```

Code 5: synchronized block

3.3 Distributed Programming

Remote communication is an inevitable part of distributed programming. In Java, Remote Method Invocation (RMI) helps to achieve this communication between the systems. RMI allows an object running in one Java virtual machine to invoke methods on an object running in another Java virtual machine. In an RMI application, there will be a client program and a server program. The server program will generate a remote object and a reference of that remote object is made available for the client (Code 6). The remote objects on the server can be accessed by client requests, and thus, the client can invoke the server's methods (Code 7).

```
public static void main(String[] args) {
        try {
                String name = "ForkServer";
                ChopstickInterface server = new
                   ChopstickServer();
                Registry registry =
                   LocateRegistry.createRegistry(8000);
                registry.rebind(name, server);
        } catch (Exception e) {
                e.printStackTrace();
        }
}
```

Code 6: main() method of the server program

Unlike multithreaded and parallel programs, synchronized keyword can't be used in distributed systems because they can only be used to control access over a shared object within the same JVM. Hence, five semaphores have been used to keep track of the availability of the five forks. A semaphore uses a counter to keep track of the status of a shared resource and controls its access. The message complexity is 2(N-1) here.

```
public static void main(String[] args) {
        try {
                String name = "ForkServer";
                Registry registry =
                   LocateRegistry.getRegistry();
                frk= (ChopstickInterface)
                   Naming.lookup("//localhost:8000/"+name);
                for (int i = 0; i <= 4; i++)
                        new Philosophers(i, registry);
        }
        catch(Exception e) {
                System.err.println(e);
        }
}
```

Code 7: main() method of the client program

To execute the program, the server program has to run and then the client program. By running the server, it creates an object. Then, using reBind() method, it registers this object with the RMIregistry. To make use of the server methods, the client needs a reference of the object that the server created. With the help of the lookup() method, the client fetches the object from the registry using its bind name. Remote communication is thus established. The server program includes getForks() and returnForks(), methods to handle the forks, which were included in the remote interface. In the client program, five philosophers are initialized using Java Threads and they run simultaneously.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ChopstickInterface  extends Remote{
        int getForks(int philNum) throws
            RemoteException;
        int returnForks(int philNum) throws
            RemoteException;
}
```

Code 8: Remote interface of the program

4 Analysing Program Behavior

4.1 Multithreaded Program Sequence Diagram

There are five philosophers present in Fig. 1 namely Philosopher 1, Philosopher 2 and so on till Philosopher 5. As in Fig. 1, we can see that Philosopher 5 goes to the thinking state for a while and then starts eating. After a certain amount of time Philosopher 5 releases the fork which allows its adjacent Philosopher 1 to gain access on one of the forks. The time gap between one process leaving the critical section and the next process accessing it is known as the Synchronization delay. In this case, the time interval between one philosopher exiting the eating state(releasing left and right forks) and the next philosopher entering the eating state(gaining control over left and right forks) is the synchronization delay. In Fig. 1, there is a time gap between take1 and eating1. This time gap between sending the request and then getting control over both the forks is known as the Response time. When comparing Synchronization delay and Response time, the delay is relatively lower and response time is very high.

4.2 Parallel Program Sequence Diagram

Figure 2 shows the sequence diagram on parallel implementation of the Dining philosophers diagram. Philosopher 1, initially being in the thinking state, goes to the hungry state (requests for forks) after a while. This time gap between sending a request for forks(hungry 1) and getting access to it(getForks1) is known as Response Time. Philosopher 1 then puts back the fork, i.e., returnForks(), making it available

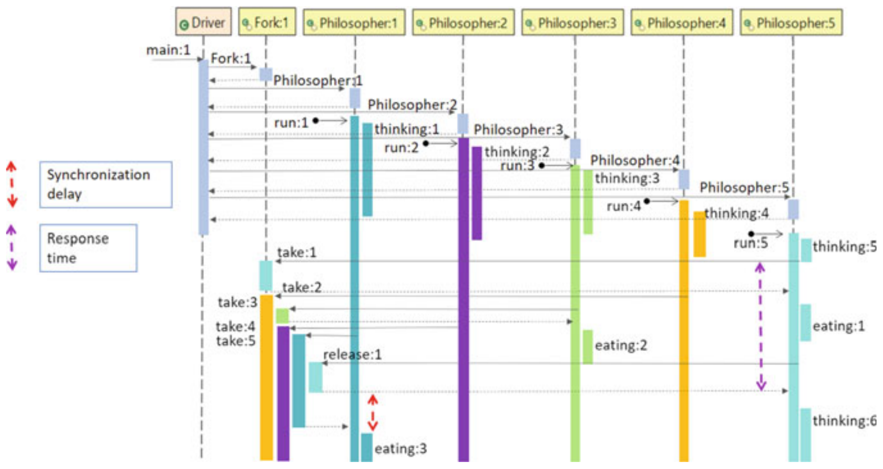

Fig. 1 Sequence diagram of multithreaded implementation

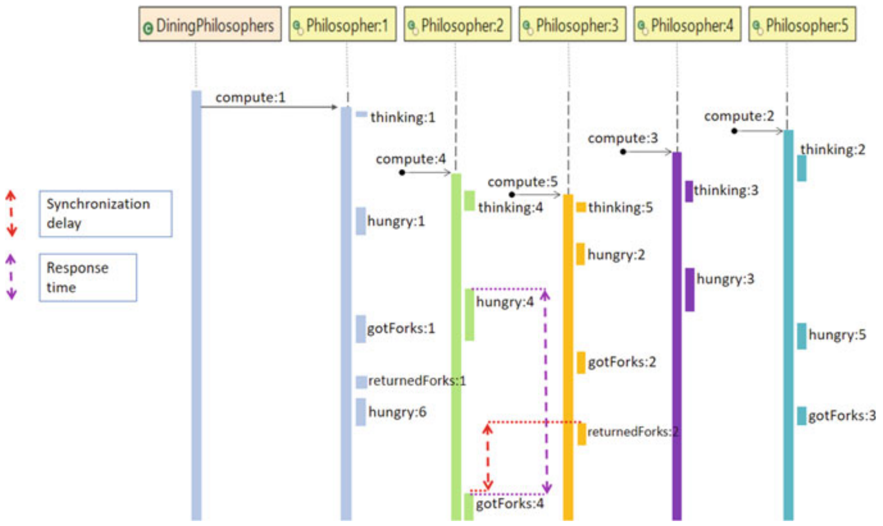

Fig. 2 Sequence diagram of parallel implementation

for adjacent philosophers. The time gap between returnedForks1 and GotForks4 is the Synchronization Delay. When looking at the variations of Synchronization delay, in most of the cases it is low. There are a few exceptions where the delay is drastically high, but on an average it is low. When comparing the Response time, for most of the cases it is relatively high. In this observation, there has been no low response time.

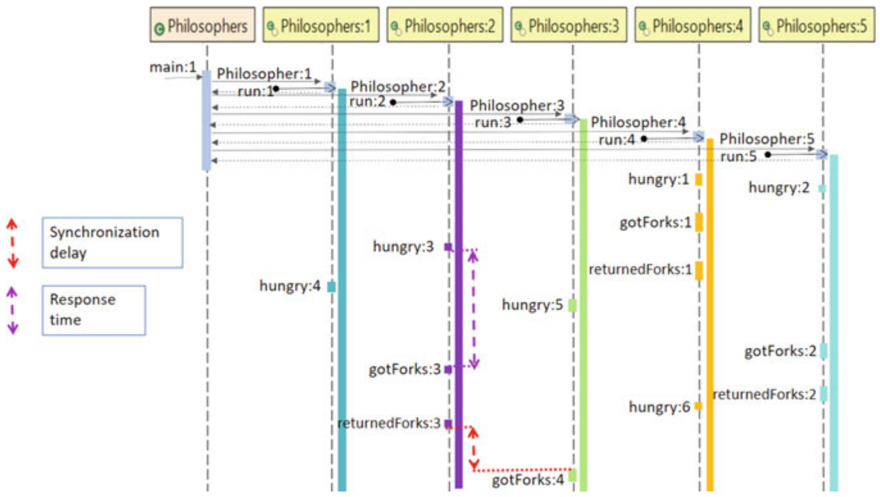

Fig. 3 Sequence diagram of distributed implementation

Table 1 Comparison based on three metrics

Metrics	Multithreaded	Parallel	Distributed
Message complexity	$2(N-1)$	$4(N-1)$	$2(N-1)$
Synchronization delay	Low	Low	very low
Response time	High	High	Low

4.3 Distributed Program Sequence Diagram

For distributed programs, the communication processes are distributed across different hosts. So, different sequence diagrams will be obtained for the server program and client program. But, here our main motive is to check upon the states of the philosopher, only the client program is taken into consideration (Fig. 3). Initially, all the philosophers will be in the thinking state and after some random amount of time they will move to the hungry state. Mutual exclusion takes place at this point. Philosophers check whether the forks are available for them. If both the forks are free, it's denoted by gotForks. By checking the diagram, it can be observed that the time gap between hungry and gotForks is most of the time high. So, it can be concluded that the response time of distributed programs is high. Comparing two adjacent philosophers, the time gap between returnedForks of one philosopher and gotForks of the other philosopher is very low, making the synchronization delay of the distributed program too low (Table 1).

5 Results

The following are observed from the sequence diagrams obtained from the concurrent programs:

- In the multithreaded program, the synchronization delay is low and the response time is high.
- In the parallel program, the synchronization delay is mostly low, but in some cases it is drastically high. Also, the response time is high.
- In the distributed programs, the synchronization delay and response time are low.

- When comparing synchronization delay of the three executions, Multithreaded execution and parallel execution take relatively lower time, while distributed execution takes the lowest time.
- When comparing the response time of the three executions, multithreaded execution and parallel execution took higher time while distributed execution took relatively lower time.

6 Conclusion

When the concept of mutual exclusion is taken into consideration, distributed programming rarely springs to mind owing to its increasingly complex nature. So we have directly compared the efficiency of distributed programming to its more frequently used counterparts such as parallel and multithreaded programming. The main observations were based on the comparison of performance metrics of Dining philosophers problem, on a small load, run in three different ways-multithreaded, parallel and distributed systems.

Java Threads are used for multithreaded programming and synchronization is achieved by using the synchronized methods. The message complexity of the multithreaded program is $2(N-1)$, N is the count of the threads/philosophers. To attain parallelism, Fork/Join framework is used and the synchronized blocks help in synchronization. The message complexity of the parallel program is $4(N-1)$, N is the count of philosophers. And, distributed programs are built using RMI framework and semaphores are used to achieve synchronization. Its message complexity is $2(N-1)$, N is the number of philosophers [14].

The performance analysis of the concurrent programs is done using the metrics message complexity, synchronization delay and response time. The message complexity is the number of messages required per execution of a critical section. It was found that in terms of message complexity the program run in parallel was the most complex while the programs run in distributed and multithreaded manner shared the same complexity. Synchronization delay is the time required for a process to enter the critical section after another process exits the critical section. It was also observed that Synchronization delay for parallel and multithreading is almost similar but Synchronization delay for distributed systems is less compared to the other two. Response time is the time interval a request waits for its critical section execution to be over after its request messages have been sent out. When comparing the response time of parallel and multithreaded programming, they have almost same response time but, while comparing response time of distributed and multithreading or parallel, response time of multithreading is more than response time of distributed.

In conclusion, for a mutual exclusion program of low load, if RT is the response time and SD is the synchronization delay then,

$$\text{RT (Multithreading)} \sim \text{RT (Parallel)} > \text{RT (Distributed)} \tag{1}$$

$$\text{SD (Multithreading)} \sim \text{SD (Parallel)} > \text{SD (Distributed)} \tag{2}$$

As part of future work, the paper can be extended by experimenting on high load and comparing results.

References

1. E.W. Dijkstra, *Hierarchical Ordering of Sequential Processes* (Academic Press, 1972)
2. S. Oaks, H. Wong, *Java: Threads* (O'Reilly & Associates, Inc., 1999)
3. R. Stewart, J. Singer, *Comparing Fork/Join and MapReduce*
4. J. Ponge, *Fork and Join: Java Can Excel at Painless Parallel Programming Too!* (2011). https://www.oracle.com/technical-resources/articles/java/fork-join.html
5. S.P.A.R. Quintao, *Performance Evaluation of Java RMI: A Distributed Object Architecture for Internet Based Applications*
6. D. Lea, *Concurrent Programming in Java: Design Principles and Patterns* (1997)
7. N.S. Nair, A. Mohan, S. Jayaraman, Interactive exploration of compact sequence diagrams—JIVE based approaches, in *2020 Third International Conference on Smart Systems and Inventive Technology (ICSSIT)*. https://doi.org/10.1109/ICSSIT48917.2020.9214261
8. S. Jayaraman, B. Jayaraman, D. Lessa, Compact visualization of Java program execution. Softw. Pract. Exper. **47**, 163–191. https://doi.org/10.1002/spe.2411
9. K. Jevitha, S. Jayaraman, M. Bharat Jayaraman, Sethumadhavan, *Finite-State Model Extraction and Visualization from Java Program Execution* (Practice and Experience, Software, 2020). https://doi.org/10.1002/spe.2910
10. R. Sharp, A. Rountev, Interactive exploration of UML sequence diagrams, in *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*. https://doi.org/10.1109/VISSOF.2005.1684295
11. P. Gestwicki, B. Jayaraman, Methodology and architecture of JIVE, in *Proceedings of the 2005 ACM Symposium on Software Visualization, SoftVis'05* (ACM, New York, NY, USA, 2005), pp. 95–104
12. S. Jayaraman, D. Kishor Kamath, B. Jayaraman, Towards program execution summarization: deriving state diagrams from sequence diagrams, in *2014 Seventh International Conference on Contemporary Computing (IC3)*. https://doi.org/10.1109/IC3.2014.6897190
13. A.A. Aziz, M. Unny, S. Niranjana, M. Sanjana, S. Jayaraman, decoding parallel program execution by using Java interactive visualization environment (JIVE): behavioral and performance analysis, in *2019 3rd International Conference on Computing Methodologies and Communication (ICCMC)*. https://doi.org/10.1109/ICCMC.2019.8819754

14. Distributed Mutex. https://github.com/01shobitha/Distributed-Mutex/tree/master/Dining-Philosopher. Last accessed 30 June 2021
15. S.K. Gandhi, P.K. Thakur, *Analysis of Mutual Exclusion Algorithms with the significance and Need of Election Algorithm to solve the coordinator problem for Distributed System* (2013)
16. K. Karippara, A.D. Nayar, V. Illikkal, N. Vasudevan, S. Jayaraman, Synthesis, analysis and visualization of networked executions, in *2020 5th International Conference on Communication and Electronics Systems (ICCES)*. https://doi.org/10.1109/ICCES48766.2020.9138091
17. J. Maassen, R. Van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, R. Hofman, Efficient Java RMI for parallel programming. ACM Trans. Programm. Lang. Syst. (2001)
18. J.M. Kannimoola, B. Jayaraman, K. Achuthan, Run-time analysis of temporal constrained objects, in eds. by D. Seipel et al. (DECLARE 2017, LNAI 10997, Springer Nature, Switzerland AG, 2018), pp. 20–36. https://doi.org/10.1007/978-3-030-00801-7_2
19. M. Abraham, K.P. Jevitha, *Runtime Verification and Vulnerability Testing of Smart Contracts*, eds. by M. Singh et al. (ICACDS 2019, CCIS 1046, Springer Nature Singapore Pte Ltd. 2019), pp. 333–342. https://doi.org/10.1007/978-981-13-9942-8_32
20. C. Artho, A. Biere, Applying static analysis to large-scale, multi-threaded Java programs, in *Proceedings 2001 Australian Software Engineering Conference*. https://doi.org/10.1109/ASWEC.2001.948499