Effectiveness evaluation of the synchronization sequence testing in Java concurrent program

Rodolfo Adamshuk Silva[1], **Simone do Rocio Senger de Souza**[2]

[1]Universidade Tecnológica Federal do Paraná
Estrada para Boa Esperança – Dois Vizinhos – PR – Brazil

[2]Instituto de Ciências Matemáticas e de Computação - Universidade de São Paulo
São Carlos – São Paulo – SP – Brazil.

rodolfoa@utfpr.edu.br, srocio@icmc.usp.br

Abstract. *Testing activities for concurrent programs are complex due to the non-determinism in which different synchronization sequences may occur and produce different outputs for a single input. Testing all synchronization sequences is impractical due to the high number of possible synchronization sequences that a concurrent program may perform. This paper presents a case study to evaluate the effectiveness of different synchronization sequences for a given test case. The effectiveness was measured using the number of defects founds by each synchronization sequence in a set of 9 Java multithread programs. The results demonstrate that the execution of different synchronization sequences leads to different effectiveness for some test cases. The evidence found demonstrates the importance of testing different synchronization sequences during testing of concurrent programs.*

Resumo. *As atividades de teste para programas concorrentes são complexas devido ao não-determinismo no qual diferentes sequências de sincronização podem ocorrer e produzir saídas diferentes para uma única entrada. Testar todas as sequências é impraticável devido ao alto número delas que um programa concorrente pode executar. Este artigo apresenta um estudo de caso para avaliar a eficácia de diferentes sequências de sincronização para um dado caso de teste. A eficácia foi medida usando o número de defeitos encontrados por cada sequência de sincronização em um conjunto de 9 programas Java multithread. Os resultados demonstram que diferentes sequências de sincronização têm eficácia diferente para alguns casos de teste. As evidências encontradas demonstram a importância de testar diferentes sequências de sincronização durante o teste de programas concorrentes.*

1. Introduction

Concurrent programming has become an essential paradigm for reductions in computational time in many application domains. A concurrent program is composed of concurrent and/or parallel processes or threads that interact for solving a complex problem. The development of concurrent programs requires the use of primitives to define processes to be executed in parallel, create, destroy and synchronize concurrent processes [Almasi and Gottlieb 1989]. Concurrent programs may compete for the same computing resource, and their interaction may occur in a synchronized way. The communication

and synchronization can be done using shared memory or message-passing paradigms. In shared-memory paradigm, different processes do write and read operations over a shared memory space. On the other hand, in the message-passing paradigm, processes use primitives to send and receive messages. Communication, synchronization, and concurrency among process can produce different and correct outputs when running with a single input depending on the synchronization sequence (non-determinism). Features as non-determinism, synchronization, and inter-process communication difficult the validation and testing and must be regarded in the testing activity, once it is impossible to predict the synchronization sequence that will run.

Software testing is an activity that aims at identifying errors in software and consists of dynamic analysis for the identification and elimination of defects. This activity involves four steps, namely test planning, test case design, implementation and evaluation of results that must be applied along the software development process [Myers et al. 2011]. Test criteria are defined for the selection of test cases aiming at a high probability to find errors and prevent the running of the program with all possible inputs. The testing of concurrent programs is complex when compared with sequential programs due to the non-determinism. A problem that arises while testing concurrent programs is the selection of synchronization sequences. The execution of all feasible synchronization sequences may be impractical, and consequently, it is necessary to select a subset of synchronization sequences to execute. A raised problem in software testing is how to select synchronization sequences and guarantee that the software is well tested. Therefore, it is necessary the definition of an approach to select synchronization sequences to be execute.

Studies have been defined approaches to deal with the selection of synchronization sequences for concurrent programs [Huo Yan Chen et al. 2003, Hong et al. 2012]. However, there is a lack of studies in the efficiency of each synchronization sequence in a given test case. This paper presents a case study to investigate and evaluate the ability to reveal faults (measured by defects found) of different synchronization sequences. It is essential to understand how to identify them to provide support in the selection of synchronization sequences to be executed in the testing. The case study considers a set of nine Java multi-threaded programs, a set of defective programs and a set of test cases taken from Gligoric et al. [Gligoric et al. 2013].

2. Background

The correctness of a concurrent program using shared-memory paradigm requires mutual exclusion, in which statements from the critical section of two or more processes must not be interleaved. A critical section is a segment that must be executed by only one thread at any time. Therefore, synchronization mechanisms are required to ensure the correctness of the results [Tanenbaum 2007]. Synchronization mechanism, such as semaphores, barriers, monitors, and condition variables consist of additional statements that are placed before and after the critical section. Shared-address-space programming paradigms such as threads (e.g., POSIX and Java) and directives (e.g., OpenMP) support synchronization using locks and related mechanisms.

Figure 1 presents an example of two concurrent threads competing for a shared memory resource. Thread 1 (T1) and Thread 2 (T2) compete for the shared variable x. Each thread executes three operations in x: read, increment and write. The execution

order of Thread 1 and Thread 2 may lead to different outputs as shown in Table 1. For instance, T1 executes the read function, increments the variable x locally and executes the write function, the value of x will be 1. After that, T2 executes the read function, increments the variable x and executes the write function, the value of x will be the correct value 2. In another scenario, T1 executes the read function and increments the variable locally. At this moment, a thread preemption occurs, and T2 starts its execution. T2 reads the value of x (0), increments it and executes the write function (x has the value 1). After that, T1 returns its execution and executes the write function. At the end of this execution, x has the incorrect value of 1. If this access to the shared variable is not synchronized, the synchronization access could lead to an error in the programs. The testing activity in this context is responsible for identifying thread scheduling that may lead to errors.

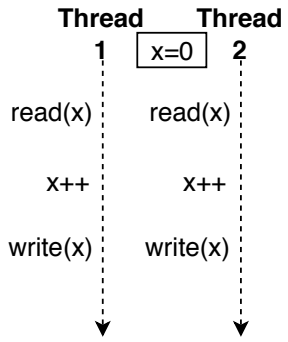

Figure 1. Representation of thread execution

Table 1. Possible synchronization sequences

Sync-seq	x value	Sync-seq	x value
$T1_{read(x)}$	x=0	$T1_{read(x)}$	x=0
$T1_{write(x)}$	x=1	$T2_{read(x)}$	x=0
$T2_{read(x)}$	x=1	$T1_{write(x)}$	x=1
$T2_{write(x)}$	x=2	$T2_{write(x)}$	x=1
correct		incorrect	
$T2_{read(x)}$	x=0	$T2_{read(x)}$	x=0
$T2_{write(x)}$	x=1	$T1_{read(x)}$	x=0
$T1_{read(x)}$	x=1	$T1_{write(x)}$	x=1
$T1_{write(x)}$	x=2	$T2_{write(x)}$	x=1
correct		incorrect	
$T1_{read(x)}$	x=0	$T2_{read(x)}$	x=0
$T2_{read(x)}$	x=0	$T1_{read(x)}$	x=0
$T2_{write(x)}$	x=1	$T2_{write(x)}$	x=1
$T1_{write(x)}$	x=1	$T1_{write(x)}$	x=1
incorrect		incorrect	

Different approaches have been defined to select synchronization sequences during the test of concurrent programs. The Multiple Execution Testing (MET) approach consists of executing a program P with the same test input several times and examining the result of each execution. If one of the executions presents an output not expected, an error was identified in the program. This approach does not ensure that all possible synchronization sequences will be executed. Therefore, a non-executed synchronization sequence can lead the program to an error state. The Deterministic Execution Testing (DET) approach [Tai et al. 1989] executes the program P with a test case t defined by an input x and a synchronization sequence s. The execution of x with the sequence s is forced, and if the result is different from the expected, a fault was found in P. The problem with this approach lies in finding all synchronization sequences that can be exercised by P and determining their executability.

Delamaro [Delamaro 2004] developed an approach for the reproduction of a Java concurrent program using instrumentation. It is based on the technique of *Record and Playback*, in which the synchronization sequence occurred during the run of synchronized methods and objects is recorded. In the playback phase, the synchronization sequence guides the next event to be run when a thread enters a synchronized point. The approach was useful for replicating Java concurrent programs with instrumentation. Lei and Carver [Lei and Carver 2010] proposed the Reachability Test in which all the feasible synchronization sequences are obtained, reducing the number of redundant ones.

Through the identification of "race conditions" between pairs of synchronizations, the approach determines during execution which synchronizations are possible to occur in a new run. The prefix-based testing technique is employed to run the program deterministically until a specific part and, after that point, it allows non-deterministic execution. Synchronization sequences are generated automatically and on-the-fly, without building any static model. This model presents the problem of the explosion of concurrency states since the number of states grows exponentially with the number of processes and the number of synchronizations to be performed.

3. Case study design

Software engineering case studies examine software engineering phenomena in their real-life settings and require a flexible design, in contrast to the fixed designs of classic experiments [Runeson et al. 2012]. The case study protocol defined by Runeson et al. [Runeson et al. 2012] was used as a guidance on the design of this case study. The Rationale for the study emerges from the fact that the execution of all synchronization sequences is impractical due to the time and computational cost necessary to execute them all. The problem is how to select synchronization sequences and guarantee that the set of synchronization sequences is reliable to consider the program as well tested? A metric used in software testing that evaluates a test case is the number of faults caught by its execution (well known as mutation testing criteria). The higher the number of faults caught is, the better is the test case. We undertake this study to observe how different synchronization sequences affect the testing of Java multithread programs. The main research question being addressed by this study is:

- RQ1: What is the effectiveness of different synchronization sequences in the testing of Java concurrent programs?

The case study will investigate the testing of nine Java multithread programs. Each program has a set of test case already defined that will be used in the conduction of the study case. The number of defects revealed by the execution is used as a metric to evaluate the effectiveness of a synchronization sequence; therefore, a set of defective programs is used to evaluate the effectiveness of a synchronization sequence. These defective programs were generated using Mutation operators for Java multithreded programs [Gligoric et al. 2013]. The set of programs, test cases, and defective programs were obtained in the experiment conducted by Gligoric et al. [Gligoric et al. 2013]. Table 2 presents the multithread programs, the number of defective programs and the number of test cases for each program. In the context of this embedded single-case study, each program is a unit of analysis, the context is the Deterministic Execution Testing.

The conduction of the case study follows the Deterministic Execution Testing described as follows. (1) For each program, one test case is selected from the test case set. The program is executed with the test case. If the output of the execution is correct, the synchronization sequence executed is stored. (2) The synchronization sequence stored is executed with all defective programs, and the output is stored. (3) The data collected is analyzed, and the effectiveness (number of defects found) is calculated and stored. This procedure is performed five times for all test cases for all programs. This is a common practice used for seeking the execution of different synchronization sequences. Figure 3 presents an overview of each iteration of the procedure.

Table 2. Objects programs used in the study

Objects	#LOC	#Defective	#Test Case
Account	52	6	22
Accounts	43	6	8
Airline	38	5	18
Allocation	78	5	20
Bubble	37	2	12
Buffer	89	9	8
LinkedList	179	4	32
Shop	113	8	9
Tree	122	38	15

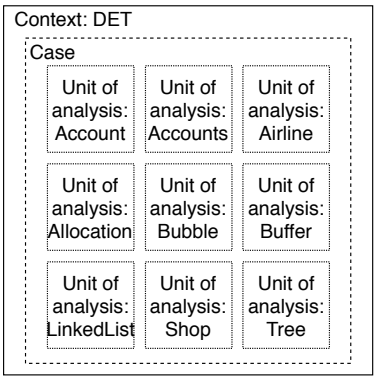

Figure 2. Embedded single-case study

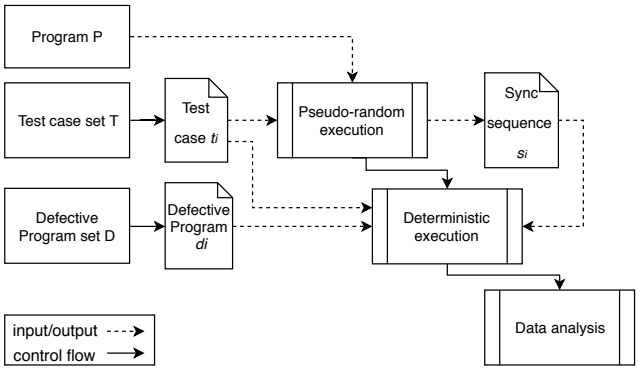

Figure 3. Overview of the case study conduction

JPF-inspector tool[1] was used to support the deterministic execution of the program and defective programs. The deterministic execution of the original program was necessary to generate different synchronization sequences once it is impractical to get it only with freely executions. However, JPF randomly selects threads to be executed, therefore we called this execution as pseudo-random. During the first step of the case study, five synchronization sequences were generated by selecting random threads to execute. A pilot case study shown that the increase in the number of synchronization sequences did not increase the number of defects found.

A set of synchronization sequences was generated for each test case of each program. After that, defective programs were also executed using the JPF-inspector deterministically, i.e., the thread scheduling follows the synchronization sequences obtained in the execution of the original program. The defective programs were executed to verify if the synchronization sequence used was able to identify the defect inserted in the program, proving thus, its effectiveness. The defect was considered found if the defective program presents a result different from the original program or it cannot follow the synchronization sequence. Table 3 presents the total number of executions for each program for the case study. Column "Exec i" corresponds to the initial execution of the program to generate the synchronization sequences and "Exec d" corresponds to the execution of the defective program.

[1]https://jpf.byu.edu/hg/jpf-inspector

Table 3. Number of executions performed in this case study

Subject	#Test	#Sync	#Defective	Exec i	Exec d
Account	22	5	6	110	660
Accounts	8	5	6	40	240
Airline	18	5	5	90	450
Allocation	20	5	5	100	500
Bubble	12	5	2	60	120
Buffer	8	5	9	40	360
LinkedList	23	5	4	115	460
Shop	9	5	8	45	360
Tree	15	5	38	75	2 850
Number of executions				**675**	**6 000**

4. Data Analysis

DeMillo and Offutt [DeMillo and Offutt 1991] defined three conditions that a test case t must satisfy to identify a defect in a code. (1) Reachability: the defect must be executed. (2) Necessity: the state of the defective program after the execution of the defect must be different from the state of the correct program after the execution of the same point. (3) Sufficiency: the difference in the state immediately following the execution of the defect must propagate to the end of execution.

In this case study, defects were considered reveled if the sufficiency condition is achieved, i.e., it presents an error as output, or it cannot follow the synchronization sequence. As a result, three errors were identified while applying deterministic testing for Java multithread programs. Error 1 is related to the output of the program and errors 2 and 3 are related to the synchronization sequence.

1. Error - This error occurs when the inserted defect makes the defective program to present a different output from the expected.
2. Not enough sync - This error occurs when the synchronization sequence finishes, however, the defective program is still running.
3. Error sync - This error occurs when the defective program cannot execute the thread presented in the synchronization sequence.

Table 4 presents the cases in which there is a variation of defects found depending on the synchronization sequence executed. The first column represents the program under testing, the second column corresponds to the test case id, the third column presents the total number of defective programs, and the last column describes the number of defects found for the five synchronization sequences executed. For the test cases out of the table, the number of defects found was the same in all executions.

With the data collected, it is possible to answer the research question 1 that investigates differences in the effectiveness related to the number of defects found by the execution of synchronization sequences. Therefore, it is essential to execute different synchronization sequences during the testing of Java multithread programs. Figure 4 presents the best and worst scenarios for test cases that exhibited synchronization sequences with different effectiveness. For instance, observing the results of test case $T11$ from *Tree* program, the best synchronization sequence was able to identify 15 defects (39%). On the other hand, the worst synchronization identified only 11 defects (29%).

Figure 5 presents the number of defects found for each synchronization sequence of test case $T11$ in the *Tree* program. It is noteworthy that each synchronization sequence,

Table 4. Number of defects found per synchronization sequence

Program	Test	Def	Effectiveness				
			S1	S2	S3	S4	S5
Account	T1	6	2	2	2	0	0
	T12		2	1	2	2	2
Accounts	T1	6	3	3	0	3	3
	T3		3	3	0	3	0
Airline	T4	5	2	2	4	2	2
	T5		2	4	2	4	3
	T11		2	2	4	4	4
	T13		4	4	2	4	4
	T15		2	4	4	4	4
Bubble	T11	2	2	1	0	0	1
Buffer	T4	9	5	7	5	6	6
Shop	T1	8	4	3	4	4	3
	T2		3	3	4	3	3
	T3		3	4	4	4	3
Tree	T6	38	11	12	14	11	12
	T7		10	12	12	12	12
	T8		14	14	13	11	12
	T9		13	14	12	14	14
	T11		15	11	13	13	12
	T12		14	13	15	12	15

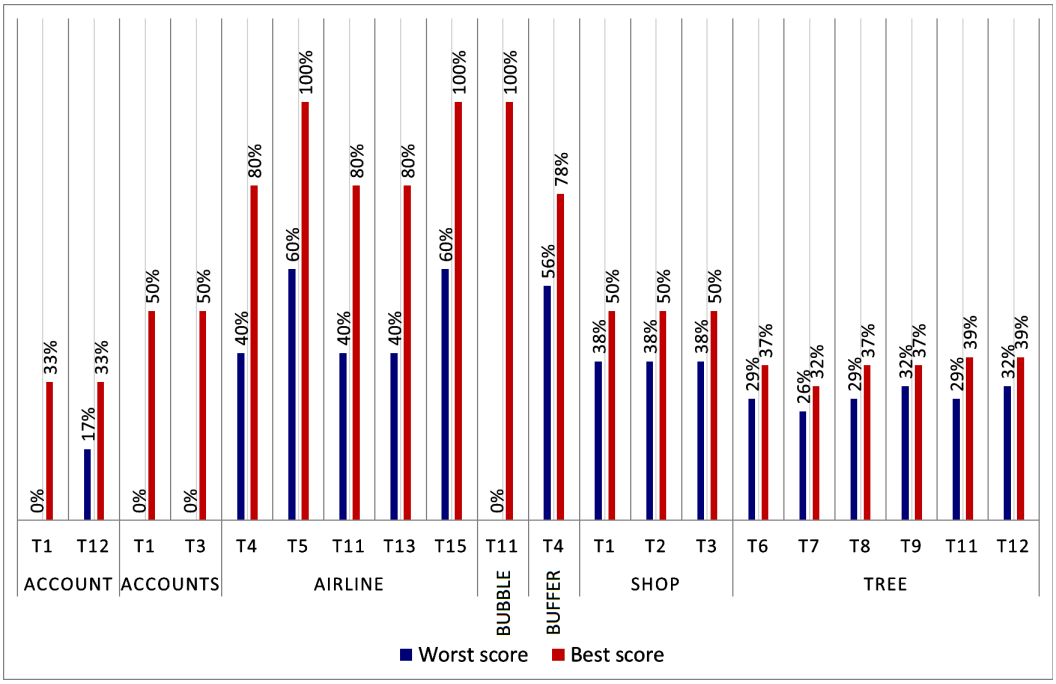

Figure 4. Effectiveness for best and worst synchronization sequences

in this test case, presents different effectiveness. This scenario exemplifies the importance of testing different synchronization sequences during the testing of concurrent programs. Analyzing a scenario in which only one synchronization sequence is executed or a non-deterministic execution is used, if just the synchronization sequence S2 is executed, some defects may not be revealed, decreasing the quality of the test case set. With these results, it is possible to identify strong and weak synchronization sequences considering the score, i.e., there are synchronization sequences that reveal more defects than others.

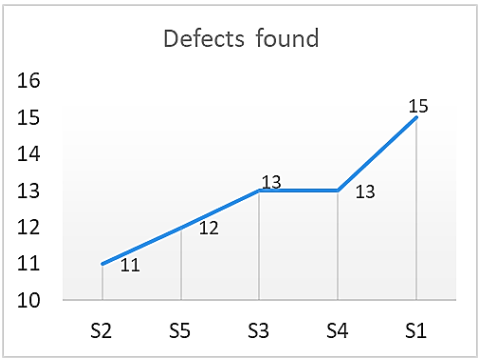

Figure 5. Defects found for each synchronization sequence of test case $T11$

5. Summary and discussion

The objective of this case study was to apply the Deterministic Execution Testing in a set of Java concurrent programs and investigate the effectiveness of synchronization sequences. Therefore, the number of defects found was used to evaluate different synchronization sequences. As a result, it was possible to affirm that different synchronization sequences can achieve different effectiveness, being crucial to the efficiency of the test case. This result is essential and motivates the investigation about to how to select the best synchronization sequences to be used in the testing of concurrent programs.

In this case study, we identified three different error situations that can happen in the execution of defective programs. Moreover, another situation was observed in which the defective program finished its execution without error, however, it does not follow the complete synchronization sequence. This situation was called “Early termination”, once the defective program finishes its execution before the end of the synchronization sequence. In the literature, there is no definition about how to deal with this error, so that the defect was considered not revealed.

Some of the error situations can be considered better, regarding the process of revealing defects. The primary objective of the testing activity is to obtain a different output from the defective program in comparison with the expected output. In this context, the error situations found in this study can be classified due to its significance.

The more important is Situation 1, once the output presented by the defective program is different from the expected output and it means that reachability, necessity, and sufficiency conditions were reached. Situation 2 is not considered an error, once the defective program finishes its execution without showing an error even presenting a synchronization sequence shorter than the one presented by an error-free program. Situation 3 is the opposite, in which the synchronization sequence of the defective program is larger than the one presented by the error-free program. This situation is significant because the defective program did not finish its execution. Situation 4 is more significant than the other two, once the defective program cannot follow the synchronization sequence due to a difference in the possible thread scheduling. This new thread scheduling may represent a difference in the concurrent characteristic of the program. Table 5 presents the error situations found in testing of Java multithread programs.

In the testing of concurrent programs, the tester has a predefined set of test cases. While testing concurrent programs, it is necessary to execute some (or all) synchroniza-

Table 5. Error situations found in testing Java multithread programs

Situation	Error	Description	Significance
1	Error	The program presents a different output from the expected	1
2	Early termination	The synchronization of the program is shorter than the original	4
3	Not enough sync	The synchronization sequence finishes, and the program is still running	3
4	Error sync	The program cannot execute the synchronization sequence	2

tion sequences to evaluate the behavior of the programs. As the execution of all sequences is expensive and time-consuming, a subset of sequences is selected and executed. In this study, multiple execution testing approach was used to generate and execute different synchronization sequences, though this approach does not guarantee the execution of "interesting" sequences. When an interesting synchronization sequence is executed, defects in the code are more likely to be found.

In this case study, the defective programs were executed five times with each test case, and as a result, it was demonstrated that different synchronization sequences could provide different effectiveness. During the execution of synchronization sequences, five sequences were randomly generated to achieve different scores. The generation and execution of those sequences were time-consuming, once no heuristic supported this activity. So, a challenge in this field is how to find interesting sequences.

A threat to validity in this case study is the representativeness of the programs. Nine programs of different size and complexity were taken from Gligoric et al. [Gligoric et al. 2013], and that was already used in different studies in the testing of concurrent programs to mitigate this threat. These programs present different concurrent functions that represent the context of Java multithread programs [Gligoric et al. 2010, Jagannath et al. 2011, Gligoric et al. 2013].

6. Conclusions

Concurrent programs have characteristics that set them apart from sequential programs such as communication, synchronization, parallelism, and concurrency. These characteristics are present in most concurrent programs and must be considered during the testing activity. The concurrency in Java is achieved by using threads, which are independent paths of execution through the application code. In the context of concurrent programs, software testing activity presents itself as challenging, since characteristics such as nondeterminism of concurrent programs must be evaluated using testing techniques.

In this paper, the Deterministic Execution Testing approach was used to evaluate the effectiveness of a set of synchronization sequences for a set of nine Java multithread programs. The first step consisted in the deterministic execution of each test case five times to generate different synchronization sequences. Those sequences were executed in the set of defective programs. A set of three errors were defined to decide if a defect was revealed or not. Synchronization sequences were classified based on the effectiveness as good (high number of defects found) and bad (low number of defects found).

As a result, it was possible to affirm that different synchronization sequences may reach different effectiveness considering one test case. This conclusion is essential for future works that aim to identify and select "interesting" synchronization sequences, i.e., sequences that may find more faults in the program during the testing activity. It is important to mention that the choice of synchronization sequence is an issue present in most

testing techniques for concurrent programs. This study is an initial contribution to studies that propose selection and generation of synchronization sequences during the testing activity of concurrent programs.

References

Almasi, G. S. and Gottlieb, A. (1989). *Highly parallel computing*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA.

Delamaro, M. E. (2004). Using instrumentation to reproduce the execution of Java concurrent programs. In *Simpósio Brasileiro de Qualidade de Software*.

DeMillo, R. A. and Offutt, A. J. (1991). Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910.

Gligoric, M., Jagannath, V., and Marinov, D. (2010). Mutmut: Efficient exploration for mutation testing of multithreaded code. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 55–64.

Gligoric, M., Zhang, L., Pereira, C., and Pokam, G. (2013). Selective mutation testing for concurrent code. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 224–234, Lugano, Switzerland.

Hong, S., Ahn, J., Park, S., Kim, M., and Harrold, M. J. (2012). Testing concurrent programs to achieve high synchronization coverage. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 210–220, New York, NY, USA.

Huo Yan Chen, Yu Xia Sun, and TH Tse (2003). A strategy for selecting synchronization sequences to test concurrent object-oriented software. In *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*, pages 198–203.

Jagannath, V., Gligoric, M., Jin, D., Luo, Q., Rosu, G., and Marinov, D. (2011). Improved multithreaded unit testing. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 223–233. ACM.

Lei, J. and Carver, R. H. (2010). A stateful approach to testing monitors in multithreaded programs. *9th IEEE International Symposium on High-Assurance Systems Engineering*, 00:54–63.

Myers, G. J., Sandler, C., and Badgett, T. (2011). *The Art of Software Testing*. Wiley Publishing, 3rd edition.

Runeson, P., Host, M., Rainer, A., and Regnell, B. (2012). *Case Study Research in Software Engineering: Guidelines and Examples*. Wiley Publishing, 1st edition.

Tai, K., Carver, R., and Obaid, E. (1989). Deterministic execution debugging of concurrent Ada programs. In *Proceedings of the Computer Software and Applications Conference*, pages 102–109.

Tanenbaum, A. S. (2007). *Organização Estruturada de Computadores*. Prentice Hall, Sao Paulo, Brazil, 5. edition.