UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Search based software testing for the generation of synchronization sequences for mutation testing of concurrent programs

Rodolfo Adamshuk Silva

Tese de Doutorado do Programa de Pós-Graduação em Ciências de
Computação e Matemática Computacional (PPG-CCMC)

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Rodolfo Adamshuk Silva

Search based software testing for the generation of synchronization sequences for mutation testing of concurrent programs

Doctoral dissertation submitted to the Institute of
Mathematics and Computer Sciences – ICMC-USP, in
partial fulfillment of the requirements for the degree of
the Doctorate Program in Computer Science and
Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and
Computational Mathematics

Advisor: Profa. Dra. Simone do Rocio Senger
de Souza

USP – São Carlos
July 2018

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

S586s Silva, Rodolfo Adamshuk
Search based software testing for the generation
of synchronization sequences for mutation testing
of concurrent programs / Rodolfo Adamshuk Silva;
orientador Simone do Rocio Senger de Souza. -- São
Carlos, 2018.
131 p.

Tese (Doutorado - Programa de Pós-Graduação em
Ciências de Computação e Matemática Computacional) --
Instituto de Ciências Matemáticas e de Computação,
Universidade de São Paulo, 2018.

1. Concurrent programming. 2. Mutation testing.
3. Search based software testing. I. Souza, Simone
do Rocio Senger de, orient. II. Título.

Rodolfo Adamshuk Silva

**Teste baseado em busca para geração de sequências de
sincronização para o teste de mutação de programas
concorrentes**

Tese apresentada ao Instituto de Ciências
Matemáticas e de Computação – ICMC-USP,
como parte dos requisitos para obtenção do título
de Doutor em Ciências – Ciências de Computação e
Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e
Matemática Computacional

Orientadora: Profa. Dra. Simone do Rocio Senger
de Souza

USP – São Carlos
Julho de 2018

ACKNOWLEDGEMENTS

Firstly, I would like to thank God for the possibility to live every day with happiness and motivation.

I would like to thank my family: my parents and to my brother and sister for supporting me throughout this PH.D. and all my life in general. Besides that, thank you for believing in my potential more than myself sometimes.

I would like to express my sincere gratitude to my advisor Prof. Dr. Simone do Rocio Senger de Souza for the continuous support of my Ph.D. study, for her attention, availability, help and, mainly, for her patience. Her guidance helped me during this research and writing of this thesis. Also, I would like Prof. Dr. Paulo Sergio Lopes de Souza for the comments, encouragement, support and contribution to this research.

My sincere thanks also go to Prof. Dr. Gordon Fraser who provided me an opportunity to join their team as an intern at the University of Sheffield, and who gave me access to the laboratory and research facilities. Thank you for your support and advice. His precious support during the internship was crucial for the results present in this thesis. I want to thank all the friends I made in Sheffield that helped me to feel more comfortable there and showed me how tasty is the meat pie at The Old House.

Last but not the least, I would like to thank my labmates and a special thank you for those that trespassed the walls of the laboratory and became partners in life: José, Silvana, Joice, Stevão, Ricardo, Maria, Leo, Pedro, Danillo, Lina, Brauner, Daniel, João, Jaqueline, Valéria, Jadson, Vânia, Juliana, and Draylson.

ABSTRACT

SILVA, R. R. **Search based software testing for the generation of synchronization sequences for mutation testing of concurrent programs**. 2018. 131 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2018.

Concurrent programming has become an essential paradigm for reductions in the computational time in many application domains. However, the validation and testing activity is more complex than the testing for sequential programs due to the non-determinism, synchronization and inter-process communication. Mutation testing is based on mistakes produced by software developers and presents a high effectiveness to reveal faults. However, high computational costs limit its applicability even for sequential code, becoming higher for concurrent programs in which each test has to be executed with different (ideally all) thread schedules. To date, only selective mutation have been applied to reduce the number of mutants in concurrent programs, however, the problem of state explosion of thread schedules still remains. This Ph.D. thesis presents the SBBMuT approach that applies deterministic execution and genetic algorithm for the generation and execution of a set of synchronization sequences during the mutation testing of Java multithreaded programs. An experimental study was conducted, and the results showed that the set of synchronization sequences generated by SBBMuT achieved a higher mutation score in comparison with the use of the Java PathFinder model checking tool.

Keywords: Concurrent programming, Mutation testing, Search based software testing.

RESUMO

SILVA, R. R. **Teste baseado em busca para geração de sequências de sincronização para o teste de mutação de programas concorrentes**. 2018. 131 p. Tese (Doutorado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2018.

A programação concorrente tornou-se um paradigma essencial para a redução no tempo computacional em muitos domínios de aplicação. No entanto, as atividades de verificação, validação e teste são mais complexas do que o teste para programas sequenciais devido ao não determinismo, sincronização e comunicação entre processos ou threads. O teste de mutação é baseado em enganos cometidos por desenvolvedores de software e apresenta uma alta eficácia para revelar defeitos. No entanto, o alto custo computacional limita a sua aplicação mesmo para programas sequenciais, e tornando-se maior para programas concorrentes no qual cada teste deve ser executado com diferentes (idealmente todas) sequências de sincronizações. Na literatura, apenas mutação seletiva foi aplicada para reduzir o número de mutantes em programas concorrentes, no entanto, o problema de explosão no número de sequências de sincronização ainda permanece. Esta tese de doutorado apresenta a abordagem SBBMuT que aplica execução determinística e algoritmo genético para a geração e execução de um conjunto de sequências de sincronização durante o teste de mutação para programas Java multithread. Um estudo experimental foi conduzido e os resultados mostram que o conjunto de sequências de sincronização gerada pela SBBMuT conseguiu alcançar um escore de mutação maior em comparação com a utilização da ferramenta de validação de modelos Java PathFinder.

Palavras-chave: Programação concorrente, Teste de mutação, Teste de software baseado em busca.

LIST OF FIGURES

Figure 1 – Methodology for the development of the thesis.	28
Figure 2 – Communication using message-passing.	32
Figure 3 – Message-passing communication example	32
Figure 4 – Uniform-memory-access shared-address-space computer (GRAMA <i>et al.</i> , 2003).	33
Figure 5 – Representation of thread execution	35
Figure 6 – Parallel control-flow graph (SOUZA <i>et al.</i> , 2008).	42
Figure 7 – Mutation Testing procedure for concurrent programs.	43
Figure 8 – Overview of JPF-core	49
Figure 9 – Execution flow of the Genetic Algorithm	52
Figure 10 – Representation of a chromosome, Greedy crossover and Swapping mutation operators	53
Figure 11 – Hill-Climbing search process (FREITAS <i>et al.</i> , 2010).	54
Figure 12 – Simulated Annealing technique (FREITAS <i>et al.</i> , 2010).	54
Figure 13 – Overview of the context, case, and units of analysis	66
Figure 14 – Overview of the protocol of the case study	66
Figure 15 – JPF-inspector selecting thread scheduling	67
Figure 16 – Killed mutants for each synchronization sequence of test case <i>T11</i>	72
Figure 17 – Mutation score for best and worst synchronization sequences	72
Figure 18 – Important characteristics in SBMT (SILVA <i>et al.</i> , 2017)	78
Figure 19 – Overview of the SBBMuT approach	80
Figure 20 – First step of the SBBMuT approach	81
Figure 21 – Second step of the SBBMuT approach	82
Figure 22 – Third step of the SBBMuT approach	83
Figure 23 – Fourth step of the SBBMuT approach	84
Figure 24 – JPF-inspector execution listing thread scheduling	85
Figure 25 – Chromosome with five genes	88
Figure 26 – Fitness calculation for two chromosomes	89
Figure 27 – Crossover operator example	90
Figure 28 – Architecture of BeMutation tool	91
Figure 29 – BeMutation Class Diagram	92
Figure 30 – Genetic Algorithm procedure	101
Figure 31 – Mutation score for SBBMuT and JPF	103

Figure 32 – 3D scatter plot for synchronization sequence grouping 108

LIST OF ALGORITHMS

Algoritmo 1 – Original program execution	81
Algoritmo 2 – Testing execution	84
Algoritmo 3 – Genetic Algorithm	87

LIST OF SOURCE CODES

Source code 1 – Airline class in Java.	39
Source code 2 – Runner1 in Java.	105

LIST OF TABLES

Table 1 – Possible synchronization sequences	35
Table 2 – Errors in concurrent programs	37
Table 3 – Mutation operators for Java multithreaded programs	45
Table 4 – Studies on the application of meta-heuristic techniques for the testing of concurrent programs	58
Table 5 – Objects programs used in the study	65
Table 6 – Mutation operators for Java multithread programs	65
Table 7 – Number of executions performed in this experiment	68
Table 8 – Mutation score for different test cases for <i>buffer</i> program	69
Table 9 – Number of killed mutants per synchronization sequence	70
Table 10 – Results of the Buffer program execution	71
Table 11 – Mutation score for best and worst synchronization sequences	71
Table 12 – Error situations found in mutation testing of concurrent programs	74
Table 13 – Input and output of Step 1	80
Table 14 – Input and output of Step 2	82
Table 15 – Input and output of Step 3	83
Table 16 – Input and output of Step 4	84
Table 17 – Mutation operators for Java multithreaded programs	86
Table 18 – Object programs used in the study	99
Table 19 – Experiment design	99
Table 20 – Total of execution in the experiment for each evaluation	101
Table 21 – Mutation score for the SBBMuT and JPF approaches	102
Table 22 – Classification of errors in the mutation testing of concurrent programs	104
Table 23 – Influence of number of threads in the mutation killing	106
Table 24 – Multiple executions of JPF versus SBBMuT execution	107
Table 25 – Synchronization sequences grouped by error for test case 6 of Tree program	109

CONTENTS

1	INTRODUCTION	23
1.1	Motivation	25
1.2	Objective	26
1.3	Methodology	27
1.4	Contributions	28
1.5	Thesis outline and overview	28
2	BACKGROUND	29
2.1	Initial considerations	29
2.2	Concurrent Programming	29
2.2.1	<i>Message-passing</i>	30
2.2.2	<i>Shared-Memory</i>	33
2.3	Errors in concurrent programs	35
2.4	Java multithreading model	38
2.5	Software testing for concurrent programs	40
2.5.1	<i>Structural testing</i>	40
2.5.2	<i>Mutation testing</i>	42
2.5.3	<i>Testing strategies</i>	44
2.6	Verification of concurrent programs	47
2.6.1	<i>Model checking</i>	48
2.6.1.1	<i>System modeling</i>	48
2.6.1.2	<i>Specification and verification</i>	48
2.6.2	<i>Java PathFinder tool</i>	49
2.6.2.1	<i>JPF extensions</i>	50
2.7	Search-Based Software Engineering	51
2.7.1	<i>Meta-heuristics search techniques</i>	52
2.7.2	<i>Search Based Software Testing</i>	54
2.7.3	<i>Search Based for testing concurrent programs</i>	58
2.8	Final considerations	61
3	CASE STUDY	63
3.1	Introduction	63
3.2	Case study design	63

3.2.1	<i>Case study protocol</i>	66
3.3	Data Analysis	68
3.3.1	<i>Threats to validity</i>	72
3.4	Summary and discussion	73
3.5	Conclusions	74
4	SEARCH BASED BEHAVIORAL MUTATION TESTING	77
4.1	Initial considerations	77
4.2	The definition of SBBMuT approach	77
4.2.1	<i>Program execution</i>	79
4.2.2	<i>Generation and selection of mutants</i>	81
4.2.3	<i>Meta-heuristic optimization</i>	82
4.2.4	<i>Testing execution</i>	83
4.3	Implementation of the SBBMuT: BeMutation testing tool	84
4.3.1	<i>Program execution</i>	85
4.3.2	<i>Generation and selection of mutants</i>	85
4.3.3	<i>Meta-heuristic optimization</i>	86
4.3.3.1	<i>The Problem</i>	87
4.3.3.2	<i>Encoding</i>	87
4.3.3.3	<i>Parameters</i>	88
4.3.3.4	<i>Initialization</i>	88
4.3.3.5	<i>Evaluation</i>	88
4.3.3.6	<i>Termination Check</i>	89
4.3.3.7	<i>Selection Method and Crossover</i>	89
4.3.3.8	<i>Elitism</i>	90
4.3.3.9	<i>Mutation</i>	90
4.3.4	<i>Testing execution</i>	90
4.3.5	<i>BeMutation Architecture</i>	91
4.4	Discussion	92
4.5	Final considerations	93
5	EXPERIMENTAL STUDY	95
5.1	Initial considerations	95
5.1.1	<i>Objective</i>	95
5.2	Experiment design	96
5.2.1	<i>Definition</i>	96
5.2.2	<i>Planning</i>	96
5.2.2.1	<i>Context selection</i>	96
5.2.2.2	<i>Research question definition</i>	97
5.2.2.3	<i>Hypothesis formulation</i>	97

5.2.2.4	<i>Variable selection</i>	98
5.2.2.5	<i>Object selection</i>	98
5.2.2.6	<i>Design selection</i>	99
5.3	Operation	99
5.4	Analysis and interpretation	101
5.4.1	<i>Descriptive analysis</i>	102
5.4.2	<i>Hypothesis testing</i>	103
5.4.3	<i>Summary and discussions</i>	103
5.4.3.1	<i>Threats to validity</i>	109
5.5	Final considerations	110
6	CONCLUSION	113
6.1	Contributions	114
6.2	Difficulties and Limitations	115
6.3	Future Work	116
6.4	Scientific Production	117
BIBLIOGRAPHY		119

INTRODUCTION

Distributed computing involves concurrent or parallel processes or threads that interact to solve a problem. Their use has become more present in new systems due to technological advances in hardware, e.g., multicore processors. Concurrent programming has been used for the development of software to reduce the computational time in some application domains, for example, a web server that creates processes (or threads) to each request aiming a performance enhancing.

The development of concurrent programs requires the use of primitives for defining processes to be executed in parallel, creating, destroying and synchronizing concurrent processes (ALMASI; GOTTLIEB, 1989). Concurrent programs may compete for the same computing resource, and their interaction may occur in a synchronized way. Communication and synchronization can be performed through shared memory or message-passing paradigms. In the former, different processes do *write* and *read* operations over a shared memory space, whereas, in message passing, *send* and *receive* primitives are used for communication and synchronization among processes in the same or different memory space.

Due to aspects of communication, synchronization, and concurrency among processes or threads, concurrent programs are non-deterministic, once different executions of a concurrent application with a single input can produce different and correct outputs. However, it is impossible to predict the behavior of the program before its execution. This behavior also called synchronization sequence is the order in which the processes or threads executed during the run of the program. As a result of such characteristic, validation and testing activities become more difficult to be performed once one among several synchronization sequences can lead the program to an erroneous state. The objective is to determine (if possible) whether all possible synchronization sequences are performed and evaluate if the outputs are correct.

Software testing aims at identifying errors in software and consists of dynamic analysis for the identification and elimination of defects. It involves four steps, namely test planning, test

case design, implementation and evaluation of results, which must be applied to the software development process (MYERS *et al.*, 2011). It is impractical to test all possible inputs for a given program, so that, the testing criteria are defined for the selection of test cases more likely to find defects. The criteria used rely on the technique used by the tester, e.g., functional testing uses functional characteristics as criteria to be covered, whereas structural testing uses code's structure.

Some testing techniques defined for sequential programs have been adapted to be used for testing concurrent/distributed program, and tools have been developed to support those techniques. Arora *et al.* (2015) classified approaches for testing concurrent programs in 8 categories, namely reachability testing (LEI; CARVER, 2010), structural testing (CARVER, 1993; TAYLOR *et al.*, 1992; YANG *et al.*, 1998; SOUZA *et al.*, 2014), model-based testing (MCMILLAN, 1995; SHOUSHA *et al.*, 2012), mutation testing (SILVA-BARRADAS, 1998; BRADBURY *et al.*, 2006; SILVA *et al.*, 2012b), slicing-based testing (CHENG, 1993; CHEN; XU, 2001), formal methods (WEISS, 1988; VASENIN; KRIVCHIKOV, 2013), random testing (SEN, 2007; SEN, 2008), and search based testing (KŘENA *et al.*, 2010; HRUBÁ *et al.*, 2014).

Besides the definition of techniques, research works have been done in both academy and industry to improve the quality of testing concurrent programs. Melo *et al.* (2015) developed a catalog containing 116 tools for testing concurrent programs. Those tools address seven different testing techniques: structural testing, functional testing, mutation testing, model-based testing, data race and deadlock detection, deterministic testing and symbolic execution.

Approaches, as Multiple Execution Testing (MET), Deterministic Execution Testing (DET) (CARVER, 1993) and Reachability Testing (RT) (LEI; CARVER, 2010) have been defined for testing different synchronization sequences in concurrent programs. In MET, the program is executed several times so that different synchronization sequences can be performed. However, there is no guarantee that other sequences will occur, once the synchronizations are created during the free running of the program and cannot be predicted. On the other hand, DET uses code instrumentation to force the execution of a program through a given synchronization sequence. Finally, RT (LEI; CARVER, 2010) provides mechanisms that find and execute all feasible synchronization sequences through the use of one sequence as a template. However, the testing of all synchronization sequences is costly once a high number of them may occur for even small programs.

Mutation testing is a fault-based testing technique that uses mutation score as a testing criterion (YUE; HARMAN, 2011). It provides a metric that determines the quality - effectiveness - of a test suite through the injection of intentional bugs into a system and counting the number of bugs caught (PARSAI, 2014). The procedure is defined as follows: (1) the original program is run with the initial test case set. If the program presents an output different from the expected it contains a fault and must be corrected otherwise, (2) small syntactic modifications are inserted in the program under test and, for each modification, a modified version of the program -

called mutant - is created. Such modifications are created by mutation operators - a syntactic modification rule that changes the program under test according to a fault taxonomy of the target language or paradigm of development. Therefore, only syntactically correct mutants are created. (3) Each mutant is run with the test case set, and if an error or failure is detected, the mutant is set as killed. Otherwise, the mutant is equivalent and behaves similarly for all inputs as the program under test, or the test case set is weak and needs improvement. (4) New test cases are created to kill the remaining nonequivalent mutants and (5) the mutation score is calculated by the ratio of the number of killed mutants over the total number of mutants minus the nonequivalent mutants. The mutation testing criterion aims at obtaining one as the score, which indicates the test set can kill all non-equivalent mutants.

The mutation testing process changes when it is applied to concurrent programs due to the non-determinism characteristic. If an input is used several times in a concurrent program, different and correct outputs can be obtained, due to the synchronization sequences that may occur. Therefore, both output and synchronization sequence must be analyzed for the killing of a mutant, as reported by [Carver \(1993\)](#) (Deterministic Execution Mutation Testing), [Offutt et al. \(1996\)](#) (Multiple Execution Testing), [Silva-Barradas \(1998\)](#) (Behavior Mutation Analysis), [Delamaro et al. \(2001\)](#) (Record and Playback) and [Silva et al. \(2012a\)](#), [Silva \(2013\)](#) (Multiple Execution Testing Adapted and Deterministic Execution Mutation Testing Adapted). Those techniques reduce the number of synchronization sequences, and however, they cannot select "interesting" sequences, i.e., sequences that kill more mutants.

Currently, the mutation test has been widely accepted in the software testing community, which considers it, although relatively expensive, one of the most effective criteria for revealing defects ([DELAMARO et al., 2016](#)). Despite the benefits of mutation testing regarding effectiveness, the technique poses some problems, as the high number of mutants generated and computational costs necessary for their run and identification of equivalent mutants. When mutation testing is applied to concurrent programs, the cost increases, because different synchronization sequences of the mutant must be run, so that the fault present in the code is revealed ([GLIGORIC et al., 2013](#)). The running of different synchronization sequences is required because one of them can cause the program to produce an incorrect output. The ideal scenario would be the execution of all possible synchronization sequences to evaluate the behavior of the program. As the execution of all sequences is expensive and time-consuming, the approaches select and execute only a subset of sequences obtained through multiple executions ([OFFUTT et al., 1996](#); [SILVA-BARRADAS, 1998](#); [SILVA, 2013](#)).

1.1 Motivation

Mutation testing is a test criterion that has proven to be advantageous as a way to improve the set of test cases for a given program with a high ability to reveal errors. Testers use mutation

testing to measure the quality (regarding effectiveness) of a test suite, and a test suite is considered good if it can distinguish all non-equivalent mutants from the original program. However, in the testing of concurrent programs, the synchronization sequence executed must be considered with the test case input, once some synchronization sequences can reveal the defect whereas other does not. Therefore, the effectiveness of a test case is strictly related to the synchronization sequence (or sequences) executed.

The testing of all synchronization sequences imposes high costs on the testing activity due to the high number of possible sequences for a given program. In the literature, there are several approaches to deal with non-determinism during the testing of concurrent programs (especially the mutation test), and however, any of them guarantee the execution of “interesting” sequences. When an interesting synchronization sequence is executed, defects in the code are more likely to be found. For this reason, the selection of interesting synchronization sequences is necessary to reduce the time and computational costs in the testing activity of concurrent programs.

Meta-heuristic techniques have been effectively employed to reduce computational cost in different software testing contexts. In the context of mutation testing, meta-heuristic search techniques are used for mutation operator’s selection, test data generation, mutant generation and simultaneous mutant and test data generation (SILVA *et al.*, 2017). For concurrent programs, we found in the literature the use of meta-heuristic techniques to search for sequences in model checking, noise injection configuration, contracts and test data, and however, none of them is applied in the context of mutation testing. The explosion of synchronization sequences in the testing of concurrent programs is similar to those found in model checking (GODEFROID; KHURSHID, 2004), in which different paths must be covered, and meta-heuristics are applied to solve it. In noise injection testing technique (FIEDOR *et al.*, 2015) meta-heuristics are used to search the amount of noise should be inserted in a program and the best place to insert it during the execution to make the program follows different synchronization sequences. However, meta-heuristics were not applied to generate synchronization sequences. Therefore, this lack in the search field motivates the definition of an approach that uses meta-heuristic techniques to create a set of synchronization sequences using mutation score as the fitness function.

1.2 Objective

The general purpose of this Ph.D thesis is to present the SBBMuT (Search Based Behavioral Mutation Testing) approach that identifies and generates synchronization sequences of concurrent programs and applies them during software testing. The approach uses a meta-heuristic technique to select interesting synchronization sequence for each test case and prevents the execution of all sequences. An interesting synchronization sequence maximizes the possibility of finding an error and reduces the number of synchronization sequences to be run.

Some specific objectives were defined to achieve the general objective. The first objective is to discover the existence of interesting synchronization sequences regarding mutation score in Java concurrent programs. The second objective is to analyze the SBBMuT approach related to effectiveness in revealing defects in concurrent programs in comparison with another approach.

1.3 Methodology

A methodology with some activities was defined to achieve the objectives, as depicted in Fig. 1. The process started with the definition of the idea. One problem in the mutation testing of concurrent programs is the number of synchronization sequences to be executed for each test case. In some cases, the number of the synchronization sequences is so high that prevents the use of a brute-force approach. In contexts in which a high number of solutions can be found, search based optimization techniques are applied to find good solutions in a short time. Therefore, the idea of using search based techniques for the searching of synchronization sequences was defined.

A systematic review was conducted to identify primary studies in the research area and to guarantee the singularity of the approach (SILVA *et al.*, 2017). The results of this study were used to acquire a theoretical background in the research field. After that, the scope of the work was defined. At this point, we decided how the approach should work, and the approach steps were identified.

An interaction with Prof. Dr. Gordon Fraser was developed in a partnership with Sheffield University in Sheffield - UK for six months. During this period, Prof. Gordon gave some advice and helped in the validation of the approach and helped in the definition of the initial experiment. Prof. Gordon is an expert in the field of Search Based Software Testing, and he has several published studies (MALBURG; FRASER, 2014; ROJAS *et al.*, 2016). He also has some studies in mutation testing (FRASER; ZELLER, 2010; FRASER; ZELLER, 2012; FRASER; ARCURI, 2013; ARCURI; FRASER, 2014), which were included in the aforementioned systematic review, in which genetic algorithms are used to generate test data in the context of mutation testing.

The initial experiment has as objective to execute different synchronization sequences in a set of mutants and observe the mutation score reached and highlight that different synchronization sequences reach different mutation score being some better than others. Each mutant represents a defected inserted in the code, and the objective is to find synchronization sequences that reveal more defects in comparison with others. The results of this experiment prove our hypothesis so that the SBBMuT approach could be defined.

The definition of the approach consists in how the approach should work and the tools and technologies used in it. After the definition, the approach was developed using Java language and was ready to be applied in the experiments. The approach validation consists of an experiment in which the approach is applied, and the results compared with a random approach to verify its

benefits for the testing of concurrent programs.

Figure 1 – Methodology for the development of the thesis.

1.4 Contributions

Considering the objectives, the contributions of this work include:

- To provide an overview of the area, by classifying the main contributions on search based applied to the mutation testing and to provide subsidies to identify research gaps;
- To conduct a test case, demonstrating the difference of effectiveness for distinct synchronization sequences in Java concurrent programs.
- To develop the SBBMuT approach to optimize mutation testing for concurrent programs supporting the selection of interesting synchronization sequences.
- To conduct an experiment to evaluate the efficiency of the SBBMuT approach in comparison with MET approach.

1.5 Thesis outline and overview

Chapter 2 provides an introduction of concurrent programs, software testing for concurrent programs and search based software testing, providing concepts, definitions and main characteristics of them. Chapter 3 addresses the complexity of the problem, an initial case study that aims at identifying the existence of interesting synchronization sequences in a set of Java multithread programs. Chapter 4 shows the definition of the SBBMuT (Search Based Behavioral Mutation Testing) approach, and the instantiation of BeMutation tool to be used in multithreaded Java programs using the genetic algorithm as the meta-heuristic technique. Chapter 5 presents an experiment in which SBBMuT is compared with a random approach, discusses the results and synthesis of findings. The conclusions are summarized in Chapter 6.

BACKGROUND

2.1 Initial considerations

This chapter presents the background concepts about concurrent programs, software testing, and search based software testing. Section 2.2 provides an overview of concurrent programming showing paradigms of development, some functions of Java multi-threading and possible errors found in concurrent programs. Section 2.5 addresses the concepts, techniques, and criteria for conducting the tests for concurrent programs, highlighting the mutation testing criterion, which is the theme of this Ph.D. thesis. Section 2.7 describes search based software engineering, describing some meta-heuristic techniques and its application for testing concurrent programs. Finally, Section 2.8 presents the final considerations of this chapter.

2.2 Concurrent Programming

Concurrent programming adds new concepts, models and primitives to sequential programming, supporting the development of algorithms that, when executed, create processes that can perform concurrently and that interact in problem-solving. In this way, concurrent programming allows the split of a task into smaller portions to increase application performance, improve fault tolerance, or optimize the use of resources such as processors, memory, and I/O devices.

A concurrent program is composed of two or more processes or threads that work together to perform a task (ANDREWS, 2001). This programming style differs from sequential programming in which the Von Neumann's architecture is used and has statements executed sequentially, conditional and unconditional deviations, calls to procedures and repetitive structures. Concurrent programming permits a performance optimization of applications, exploiting the concurrency in different architectures, allowing better use of available resources.

The basic idea of concurrent programming is the partition of an application into concurrent processes, each one responsible for solving a piece of the problem. The partition is done through additional software features not available for sequential programs, such as the activation and termination of concurrent processes. According to [Almasi and Gottlieb \(1989\)](#), concurrent processes are processes that began its execution and, at a specific point in time, have not finalized it yet. These processes compete for resources such as processors, memory, and I/O devices. Parallel processes represent a special type of concurrent processes, as there is a guarantee that they are running on different processors at the same time.

The computation needs to be decomposed into tasks or processes that can execute simultaneously to obtain a speed gain through the use of parallelism ([WILKINSON; ALLEN, 1999](#)). The granularity of the decomposition determines the number of tasks in which a problem is decomposed and the size (workload) of each task. According to [Grama et al. \(2003\)](#), a decomposition into many small tasks is called fine-grained and into a small number of large tasks is called coarse-grained. A fine granularity requires a larger number of processors, once there are more processes to be executed, while a coarser granularity, comparatively, requires a smaller number of processors. One concept that relates to granularity is the degree of concurrency. The maximum number of tasks that can be performed concurrently in a concurrent program at the same time is called maximum concurrency.

A concurrent program may be formed by processes or threads depending on the development paradigm used. When the message passing paradigm is used, the programs are usually composed of processes. Moreover, in a shared memory environment, programs are composed of threads ([TANENBAUM, 2007](#)). In shared memory, different threads do *write* and *read* operations over a shared memory space, whereas, in message passing, processes use primitives to send and receive messages. These paradigms are described in more detail below.

2.2.1 Message-passing

The message-based programming paradigm allows interaction between processes that do not share the same address space. The term message passing refers to a programming model based on SCP (Sequential Communication Processes) ([ROSCOE et al., 1997](#)). In SCP channels statically declared in the code are used to the communication of different processes. A problem arises when it is necessary the declaration of a high number of channels to avoid ambiguity in the communication. Therefore, channels are not flexible enough for communication among a high number of processes several times with different purposes.

In message passing paradigm, two elements are responsible for the communication between processes. The send function sends a data message to a process, and the receive function receives a data message from a process. The basic send and receive operations may be different depending on the development language and follow the model below in which *tag* corresponds to an identifier for the message sent or received.

snd(tag, memlocat, type, procwich, size) rcv(tag, memlocat, type, procwich, size)

memlocat is the beginning of the memory space of the message to be sent (send) or where it will be copied (receive). *size* means the size of the *array* sent or received and *type* represents how many bytes there are per item in *array*. The *procwich* variable indicates the destination process of the message (send) and the transmitter (receive).

The communication and synchronization in the concurrent program must follow the software design to avoid errors. To ensure the semantics of the application, send and receive functions can be implemented as synchronous, asynchronous, blocking, non-blocking and using buffers.

Message-passing platforms usually have additional hardware support, such as DMA (Direct Memory Access) and a network interface for sending and receiving messages. DMA allows copying data from one place in memory to another (for example, communication via buffers) without the CPU support. Network interfaces support the transfer of messages stored in memory buffer without CPU intervention. [Grama et al. \(2003\)](#) defined six forms of communication:

1. Blocking non-buffered (Fig. 2 a) - The *send* operation does not return until the *receive* operation begins its execution. It can be time-wasting since one function must wait another starts its execution to ensure data security.
2. Blocking buffered (Fig. 2 b) - In a *send* operation, the process copies the data into a buffer and returns as soon as the copy is complete. The process that sent the message can continue its execution without compromising the program's semantics.
3. Non-blocking non-buffered - A process that wants to send message posts a pending message and returns so that program can continue the computation. At a point in the future, when the corresponding *receive* is posted, the communication operation is initialized.
4. Non-blocking buffered (Fig. 2 c) - The process that wants to send the message returns after the initialization of the transfer to buffer. However, when returning, the operation may not have been finished.
5. Synchronous - A pair of send/receive operations are considered synchronous when both are executing at a given time and can synchronize at the beginning or end of the transmission. If the communication is done using a buffer, there is a possibility that the *receive* operation does not synchronize, since after the transmission of the data to the buffer, the *send* returns and the process continues its execution.
6. Asynchronous - An asynchronous communication is when its execution occurs at the same time as the execution of the program ([CHAUSSUMIER et al., 1999](#)). A non-blocking communication is not necessarily asynchronous since the data can be copied to a buffer and the communication itself may be delayed.

Figure 2 – Communication using message-passing.

(GRAMA *et al.*, 2003)

Fig. 3 presents how communication occurs using the message-passing paradigm. Four processes (P0, P1, P2, and P3) cooperate to perform a particular task. The vertical lines represent the execution sequence of the process, and the circles represent the points where messages are sent or received. The continuous arrows represent the flow of the message between the processes, and the dashed arrows represent a possible message flow. Due to the non-determinism of the application, it is impossible to determine which flow will occur during program execution.

Figure 3 – Message-passing communication example

Programmers have some options such as RPC (Remote Procedure Call) and RMI (Remote Method Invocation) that are abstractions built on the *send* and *receive* primitives. RPC is a high-level communication mechanism for distributed applications. In this mechanism, the calling procedure does not exist in the same address space as happens in a procedure call. Therefore,

both processes may be on the same system or different systems connected by a network interface (LI, 2000). RMI provides a simple and straightforward model for distributed computing with Java objects. These objects can be Java objects, or they can be simple wrappers around an existing Java API.

2.2.2 Shared-Memory

The term shared-memory computer is used for architectures in which the memory is physically shared among various processors, i.e., each processor has equal access to any memory segment as shown in Fig. 4 (GRAMA *et al.*, 2003). The shared-memory development paradigm allows uniform access to variables so that many agents access this shared resource (SCOTT *et al.*, 2005). All read-only interactions are invisible to the programmer, however, read/write interactions require mutual exclusion for concurrent accesses.

Figure 4 – Uniform-memory-access shared-address-space computer (GRAMA *et al.*, 2003).

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; GRAMA *et al.*, 2003). 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 (POSIX, NT) and directives (OpenMP) support synchronization using locks and related mechanisms.

The semaphore is an abstract data type composed of an integer variable usually started with the value 1, and an empty list of processes. The variable is controlled by the Operating System and manipulated by two atomic operations, namely *lock* (*var_sem*) or *p* (*var_sem*) and *unlock* (*var_sem*) or *v* (*var_sem*) (TANENBAUM, 2007). These operations are required to lock and unlock a semaphore.

The *lock()* operation tries to decrease the value of *var_sem*. If the variable is 0, the executing process stops, and it is placed into the "blocked" queue waiting for I/O. The *unlock*

operation increments *var_sem* and if any process is in the "blocked" queue, that process is placed in the "ready" queue to try the execution of the operation *unlock()* in the semaphore. A different semaphore is declared for each place where there is a need for synchronization before a critical region, that is, a region in the code where there is written access to shared data. Semaphores can have different values above 0. There is a special case of semaphore that only takes values 0 and 1. Such binary semaphores are also known as MUTEX because they are applied fundamentally in mutually exclusive operations.

The barrier is another structure commonly used for the synchronization of threads or parallel processes. The barrier is used to stop the execution of a thread until all other threads have reached the barrier (GRAMA *et al.*, 2003). Barriers can be implemented using a counter, a MUTEX, or condition variable. A single integer variable is used to control the number of threads that arrived at the barrier. If the counter variable is less than the total of threads, all threads execute a hold function. When the last thread enters the barrier, the counter value becomes equal to the number of threads and awakes all other threads using a broadcast function.

A monitor provides mutual exclusion through the encapsulation of data and procedures related to a component that can only be executed by one thread at a time. Modern programmers who use languages such as Java with classes containing routines and data annotated with the *synchronized* keyword routinely implement monitor like structures. Conditions variables are an important feature of monitors because allow threads to signal each other to perform synchronization. Each condition variable provides two operation: *wait* and *notify* (sometimes called *signal*). The wait call causes a thread to enter the queue of threads that are waiting for the state change associated with the condition variable, and the notify call is used to send a signal to the threads that the state changed (SOTTILE *et al.*, 2010).

Fig. 5 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 influence the program behavior depending on the order in which the write and read operations are executed, leading to different outputs, perhaps incorrect. Tab. 2 shows some examples of those behaviors. 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 then increments the variable *x* and executes the write function. At the end of this program, the value of *x* will be the correct value 2. On the other hand, 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* (still 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 verifying

the correctness of the program and identifying possible faults in the code that may lead to errors.

Figure 5 – Representation of thread execution

Table 1 – Possible synchronization sequences

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

2.3 Errors in concurrent programs

According to [Krawczyk et al. \(1995\)](#), in addition to the errors present in sequential programs, e.g. computational and domain error ([HOWDEN, 1976](#)), there are two types of errors related to concurrent programs, namely observability error and locking error. Given a path p_x composed of a sequence of executed basic blocks in a program, $D(p_x)$ represents the input domain, formed by all possible inputs that exercise p_x and $C(p_x)$ is the computation of p_x formed by all possible outputs for the execution of p_x . The two possible errors are defined as:

1. Observability error: occurs in a program P if for some path $p_i \in P$, there is another path $p_j \in P$ such that $D(p_i) = D(p_j)$ and $C(p_i) \neq C(p_j)$. This error is related to the test

environment when the tester cannot control the behavior of the program in parallel, and it may not be possible to exercise a sequence of the program that presents the error.

2. Locking error: occurs in a program P if for some path $p \in P$, formed by the set of nodes $p = q_0, q_1, \dots, q_j$, exists a node q_i in p such that the sub-path $p_s = q_0, q_1, \dots, q_i$ containing a $D(p_s)$ domain, there is at least one element of $D(p_s)$ in which all predicates evaluated from q_i are false. This error is related when, for example, one process does not communicate with another correctly. This will cause a process to get caught waiting for communication that will never come.

Choi and Lewis (2000) performed an analysis of 180 multithreaded programs, namely semaphore, mutex, bounded buffer and cigarette smoker developed by students. The authors presented the common defects committed by the students during the development of the programs and classified them into three categories. The data race category is related to shared variables that may be not protected, redundant or accidentally shared and related to locks that may be not acquired, premature released or multiples to protect a single shared variable. The deadlock is related to locks unrelated to protect a shared variable, never released or the intent to reacquire locks that are already held and the confusion between mutexes and test-and-set primitives. The last category is related to the error in the use of functions as the unlocking of the same lock repeatedly or assumes that the test-and-set and unlock-and-stop routines are similar (the former is atomic, and the latter is not).

Farchi *et al.* (2003) present a defect taxonomy for Java concurrent programs. The authors highlight that most defects are not exclusive to Java programs and may be found in concurrent programs in other programming languages. According to this taxonomy, defects have a pattern three categories, namely concerning unprotected code (e.g., non-atomic operation assumed as atomic, two-stage access and double, wrong or absent lock), interleaving predicted never to occur (e.g., the use of sleep for synchronization and the loss of a notify), and thread blocked or dead (e.g., block of a critical section and an orphan thread).

Long and Strooper (2003) present an error taxonomy for concurrent Java programs. The authors represent a concurrent program using a Petri net and several transitions related to lock objects were analyzed using two criteria, namely failure to execute the transition and incorrect start of the transition.

Bradbury *et al.* (2006) extend the error taxonomy for concurrent Java programs (FARCHI *et al.*, 2003; LONG *et al.*, 2004; LONG *et al.*, 2005) in the shared memory paradigm by adding some features present in Java J2SE 5.0 and four new error patterns are defined. The first pattern is related to the lack of signals that can be sent between threads. For example, in a barrier, the function await must be executed by all threads, and if one does not execute it, all threads stop their execution. The error of starvation occurs when a thread is never staggered to execute. Resource exhaustion error occurs when a thread group keeps all finite resources together, and a

thread that needs an additional resource is stuck once the required resource is over. One example of incorrect count initialization error occurs when there is a wrong initialization of the thread number in a barrier. Therefore, one possible error would be the barrier waits forever threads that will never come. This problem may happen in the initialization of a semaphore, when the number of processes is set wrong.

Wu and Kaiser (2011) presented a taxonomy of errors for concurrent programs in the shared memory paradigm (CHOI; LEWIS, 2000; FARCHI *et al.*, 2003; LONG; STROOPER, 2003; LU *et al.*, 2008) used for the creation of second order mutation operators. The error model presented contains seven errors. Data race (also called interference) in which several threads read and write in the same data and the output of the execution depends on the order in which the accesses to the data occurred. Memory Inconsistency arises when different threads have different views of the same variable. Atomicity violation is caused by the concurrent execution of several threads violating the atomicity of a code region. Deadlock occurs when, for example, two or more threads are locked forever one waiting for another. Livelock occurs when two threads are busy responding one to another and thus not progressing. Starvation happens when a thread is not able to gain access to a shared resource and does not proceed. Suspension occurs when a thread suspends or stands indefinitely.

Souza *et al.* (2016) compiled the principal errors in concurrent programs using 14 categories as shown in Tab. 2.

Table 2 – Errors in concurrent programs

Error	Description
Deadlock	Two or more threads block one another
Livelock	A thread does not finish its execution or enter in a critical region for excess of workload or lack of speed
Starvation	A thread is never selected to be executed
Count of threads	Incorrect initialization of thread or incorrect initialization of the semaphore
Atomicity violation	Concurrent execution of several threads violating the atomicity of a code region
Unprotected code	Critical region unprotected, allowing several threads to modify the same datum
Data inconsistency	Two threads have different values for the same variable
Synchronization error	Unexpected synchronization of process or threads happens
Decomposition error	Incorrect data or function decomposition in the generation of concurrent processes
Message-passing error	Incorrect use of different communication forms in message passing paradigm, causing an unexpected behavior of the program
Lost message	A message sent is never received
Communication error	A message is sent incorrectly to one process

Different error taxonomies have been defined for many programming languages such

as Ada (OFFUTT *et al.*, 1996b), MPI (VETTER; SUPINSKI, 2000; LUECKE *et al.*, 2003; KRAMMER *et al.*, 2003; DESOUZA *et al.*, 2005; SAMOFALOV *et al.*, 2005; PEDERSEN, 2006), STM (LOURENÇO; CUNHA, 2007), C and C++ (LU *et al.*, 2008), and SystemC (SEN, 2009).

2.4 Java multithreading model

Concurrency in Java is achieved through the use of threads. During the life-circle of a thread, it may assume six execution states: (1) **New**: A thread that has not yet started is in this state; (2) **Runnable**: A thread executing in the JVM is in this state; (3) **Blocked**: A thread is blocked waiting for a monitor lock is in this state; (4) **Waiting for**: A thread that is waiting indefinitely for another thread to perform a particular action is in this state; (5) **Timed_waiting**: A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state; and (6) **Terminated**: A thread that has exited is in this state.

The communication among threads can be achieved using both shared memory and message passing paradigms. In the context of this work, the shared memory paradigm will be focused. When shared variables are used, a thread writes in a variable that can be read and/or updated by another thread that shares the same address space. According to Scott *et al.* (2005), in general, programming using shared memory allows uniform access to variables. While this seems to be a benefit, many threads will access this shared resource. Therefore mechanisms, such as semaphores, conditional critical section, and monitors are necessary to assure the correctness of the results.

The Thread class provides some functions. The *sleep()* function cause a thread to temporarily cease its execution and may be timed by passing the time value in the function. The *wait()*, *join()*, *notify()* and *notifyAll()* synchronization primitives are used for suspending and waking up threads. The *wait()* function suspend the execution of a thread until woken by a *notify()*, *notifyAll()* or after a specified time. A call to *notify()* notifies one thread waiting whereas a *notifyAll()* notifies all waiting threads. When a *join()* is executed, the thread is suspended until the execution of a targeted thread is completed.

In Java, the mutual exclusion among threads may come from the use of synchronized blocks. A Java class may contain synchronous and asynchronous methods, and a thread achieves mutual exclusion by locking an object every time it entries in a synchronized block. A mutual exclusion guarantees that two or more threads will never access the same critical section at the same time, providing mutually exclusive access.

To correctly write synchronized code based on synchronized blocks may be difficult, and therefore, high-level *synchronizers* are included in the concurrency utilities, e.g., countdown latch, cyclic barrier, and semaphores. A countdown latch causes one or more threads to wait until a count reaches zero (FRIESEN, 2015). The *await()* method forces the calling thread to

wait until the latch has counted down to zero. The function *countDown()* decrements the count, releasing all waiting threads when the count reaches zero.

A cyclic barrier lets a set of threads wait for each other to reach a common barrier point. The barrier is cyclic because it can be reused after the waiting threads are released (FRIESEN, 2015). A semaphore maintains a set of permits for restricting the number of threads that can access a limited resource. A thread attempts to acquire a permit using *acquire()*, *acquireUninterruptibly()*, or *tryAcquire()*. When no permits are available, the thread blocks until some other thread releases a permit. The Lock interface offers more extensive locking operations that can be obtained via the locks associated with monitors. The function *lock()* acquires the lock and the function *unlock()* releases the lock.

Code 1 presents an example of a Java multithreaded program that simulates an airline company that makes bookings in the flights using threads. The keyword *synchronize* is used in lines 14 and 35 and allows only one thread access the synchronized block allowing mutual exclusion. Methods *start* (line 17) and *join* (line 26) are used to start and synchronize threads respectively.

```
1 public class Airline implements Runnable {
2     public int numberOfSeatsSold = 0;
3     public int maximumCapacity;
4     private boolean stopSales = false;
5     private Thread threadArr[];
6
7     public void makeBookings(int numOfTickets, int numThreads) throws Exception {
8         int numOfExtra = numOfTickets / 10;
9         numOfExtra = numOfExtra == 0 && numOfTickets > 1 ? 1 : numOfExtra;
10        maximumCapacity = numOfTickets - numOfExtra;
11
12        threadArr = new Thread[numThreads];
13        for (int i = 0; i < numThreads; i++) {
14            synchronized (this) {
15                if (!stopSales) {
16                    threadArr[i] = new Thread(this);
17                    threadArr[i].start();
18                } else {
19                    ;
20                }
21            }
22        }
23
24        for (Thread thread : threadArr) {
25            if (thread != null) {
26                thread.join();
27            } else {
28                ;
29            }
30        }
31    }
32
33    public void run() {
34        while (true) {
35            synchronized (this) {
```

```
36     if (numberOfSeatsSold < maximumCapacity) {  
37         numberOfSeatsSold++;  
38     } else {  
39         stopSales = true;  
40         break;  
41     }  
42 }  
43 }  
44 }  
45 }
```

Source code 1 – Airline class in Java.

2.5 Software testing for concurrent programs

Validation, Verification, and Test activities have the objective of discovering problems before the release of the software for the use. Therefore, these activities aim to ensure that both the software development process and the product itself agrees with its specification. During the process of software development, different activities should be performed, employing techniques, methods, and tools to improve product quality. Even by applying these techniques, methods, and tools, errors still occur and persist in the software. In this way, software testing aims at identifying as many errors as possible and increase product acceptance.

Concurrent programs have some aspects that are not present in sequential programs, e.g., communication, synchronization, and non-determinism, that should be considered when defining support mechanisms for the test activity. Although concurrent programs can present ordinary bugs like incorrect bounds of loops or indexes of arrays, the testing focus on problems that occur as a result of interleaving. Aspects related to concurrency make the test more complex, and it is possible to adapt test criteria defined for sequential programs ([TAYLOR et al., 1992](#); [YANG et al., 1998](#); [CARVER, 1993](#); [SILVA, 2013](#)).

2.5.1 Structural testing

In the structural technique, the criteria and requirements are derived from the characteristics of a particular test implementation, e.g., source code, allowing a more elaborate verification of its behavior. This technique focuses on the most important or "dangerous" points of the code ([KOSCIANSKI; SOARES, 2007](#)). The technique is applied to small parts of the code, such as subroutines, or operations related to an object. The tester can analyze the code before the test and obtain information about data that may cause an error. A set of software elements is defined from the source code and must be executed. Those elements are components that must be tested as required by the test criterion. The software logic paths are tested, in which test cases are generated to test set of conditional and loops structures and a pair of definition and use of variables. The criteria of this technique are classified based on complexity, control, and data flow ([MALDONADO, 1991](#)). In general, most of the test criteria for this technique uses the Control

Flow Graph (CFG) composed of nodes representing sets of statements executed sequentially and edges representing a control transfer between statements (MYERS *et al.*, 2011).

Taylor *et al.* (1992) generated a concurrency graph (or reachability graph) from a static analysis of Ada and CSP (Communicating Sequential Processes) programs. The graph is created by identifying possible concurrency states related to communication and synchronization functions. Each concurrency state is represented as a node, and arcs link one concurrency state to another. The authors created five criteria, namely, *all-concurrency-paths*, *all-proper-cc-histories*, *all-edges-between-cc-states*, *all-cc-states* and *all-possible-rendezvous*. The technique developed is a static analysis based on the traceability analysis and has the problem in the explosion of numbers of states. This limitation is related to the graph's size and the time required to construct it.

Yang *et al.* (1998) adapted All Du-Paths (*All Du-Paths*) criterion to concurrent programs with shared memory paradigm and the authors mention that it can be applied to message passing paradigm. In the model, a parallel control flow graph (PCFG) is created through static analysis of the program. In this graph, besides the CFG of each process, it is added edges representing the communication between the processes, called inter-process edges.

Based on Yang *et al.* (1998) and Rapps and Weyuker (1985), Vergilio *et al.* (2005) and Souza *et al.* (2008) present a set of structural criteria for concurrent programs in the message passing paradigm. Criteria are related to communication and synchronization and classified into control flow, and communication and data flow. Control flow criteria exercise edges, nodes, and paths of the GFCP. Data and communication flow criteria exercise definition-use pairs of variables. The definition of a variable assigns a value to it, and the use is a reference to it that may be computational or predicative (use in repetition or decision commands). A new concept of variable use for concurrent programs was defined in Vergilio *et al.* (2005), regarding the use of variables in communication primitives, that is, those that are used to exchange messages between processes. This approach does not support non-blocking point-to-point and collective functions, common in concurrent programs. Fig. 6 presents an example of a PCFG of a program with four processes that interact using message passing paradigm. Inter-processes edges are represented by dotted lines and represents send and receive operations.

Lei and Carver (2006) present the reachability test, in which timestamps are inserted in the code, and an uncontrolled, non-deterministic execution of the program is performed, generating a synchronization sequence Q_0 . From Q_0 , "match conditions" are identified in receive events, and variants of the synchronization sequence are generated and executed without constructing any static model. Then, prefix-based controlled runs with the same input are performed with the synchronizations variants. The variant sequences are derived from a program execution, and therefore this approach does not present the problem of non-executability. Similar to Taylor *et al.* (1992), the explosion of concurrency states is a problem, due to the number of threads and synchronization.

Figure 6 – Parallel control-flow graph (SOUZA *et al.*, 2008).

Sarmanho *et al.* (2008) extended to the shared memory paradigm the test criteria defined by Souza *et al.* (2008). Communication pairs are set at run-time due to non-determinism. The test requirements are defined a priori, statically, e.g., execute all use of a certain variable of the program. Souza *et al.* (2014) proposed a new version of the criteria defined in Souza *et al.* (2008), contemplating more communication functions and presenting a new method in the generation of test requirements that allows reducing the number of elements to test.

Souza *et al.* (2013) proposed a test model that considers both communication paradigms and focus on the analysis of data flow between processes and threads. In the model, a program is composed of a hierarchy of concurrent tasks. This program may contain an undetermined number of processes, and each process may have an undetermined number of threads. Threads may communicate with other threads intra-process and inter-process. Edges may be inter-processes which represent communication between processes and inter-threads that represent the shared variables between threads.

2.5.2 Mutation testing

Mutation testing is a fault-based technique that uses mutation score as a testing criterion (YUE; HARMAN, 2011). This technique measures the quality (regarding effectiveness) of a test suite by injecting intentional bugs into a system and counting the number of caught bugs

(PARSAI, 2014). Fig 7 contains the mutation testing procedure for concurrent programs. First, the original program P is executed with the initial test case set T . If P is correct, small syntactic modifications are inserted in P and, for each modification, a modified version of the program is created - denominated mutant. Those modifications are created regarding mutation operators - a piece of software which changes the program under test, regarding a fault taxonomy of the target language or paradigm of development.

The set of mutants is executed with the test case set, and if there is an error or failure during the execution, the mutant is set as killed. The mutation score is calculated and, if all mutants are dead, the procedure ends. Otherwise, mutants are analyzed, and the equivalents are discarded. After that, there are two possibilities while testing concurrent programs (dashed boxes): (1) the test case set is weak, and it is necessary to generate more test cases to kill more mutants or (2) it is necessary to exercise different synchronization sequences to kill more mutants. When a new test case or synchronization sequence is generated, it is necessary to execute them with the original program to see if it reveals a defect. Mutation testing technique concludes by calculating the mutation score by the ratio of the number of killed mutants over the total number of mutants minus the non-equivalent mutants. Mutation testing criterion aims to get one as score, indicating the test set is good enough to detect all bugs inserted into the program.

Figure 7 – Mutation Testing procedure for concurrent programs.

In the context of concurrent programs, mutation operators were defined for Ada (OFFUTT *et al.*, 1996; SILVA-BARRADAS, 1998), Java (DELAMARO *et al.*, 2001; BRADBURY *et al.*, 2006; GLIGORIC *et al.*, 2013), PVM (Parallel Virtual Machine) (GIACOMETTI *et al.*, 2003), SystemC (SEN, 2009), MPI (Message Passing Interface) (SILVA *et al.*, 2012b), and PThreads

(KUSANO; WANG, 2013). Delamaro *et al.* (2001) defined a set of 20 mutation operators to test concurrency and synchronization aspects of the Java language. The authors state that mutation operators cover most of the aspects related to synchronization and concurrency and are also low cost (generate a small number of mutants). The main structures related to the concurrency were identified, and mutation operators were defined to execute these structures. The authors defined the operators is based on 4 groups of structures of Java related to concurrency, namely (1) Monitor lockout code, (2) Methods related to wait set manipulation that are defined in the Java core API, (3) Use of synchronized methods, and (4) Use of other methods related to synchronization and concurrency.

Bradbury *et al.* (2006) defined mutation operators for Java programs (J2SE 5.0). The authors used the fault taxonomy defined by Farchi *et al.* (2003) as a basis for the creation of the operators. From this model, the authors present five categories of mutation operators, namely (1) Modify Parameters of Concurrent Methods, (2) Modify the Occurrence of Concurrency Method Calls (removing, replacing, and exchanging), (3) Modify Keyword (adding and removing), (4) Switch Concurrent Objects and (5) Modify Concurrent Region (shifting, expanding, shrinking, and dividing).

Gligoric *et al.* (2013) explored selective mutation for concurrent mutation operators for Java programs. The authors used the mutation operators implemented by Bradbury *et al.* (2006) and created three operators. Tab. 17 presents the mutation operators defined in Gligoric *et al.* (2013).

2.5.3 Testing strategies

In order to deal with the non-determinism in testing of concurrent programs, some strategies were defined, e.g. Deterministic Execution Mutation Testing (CARVER, 1993), Multiple Execution Testing (OFFUTT *et al.*, 1996), Behavior Mutation Analysis (SILVA-BARRADAS, 1998), Record and Playback (DELAMARO, 2004), Reachability Testing (LEI; CARVER, 2010) and DMT and MET adapted (SILVA, 2013; SILVA *et al.*, 2012a).

Two basic approaches are used to execute different synchronization sequences in testing concurrent programs: MET (Multiple Execution Testing) and DET (Deterministic Execution Testing). The MET approach consists of executing a program P with an input x 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 technique does not ensure that all possible synchronization sequences were executed. Therefore, a different synchronization sequence can lead to an error in the program.

In the DET approach presented in Tai (1985), Tai (1986) and Tai *et al.* (1989), each test case is defined by an input x and a synchronization sequence s . For each test case, the execution of the x entry with the sequence s is forced, and the result is observed. If the result of forced

Table 3 – Mutation operators for Java multithreaded programs

Category	Concurrency Mutation Operators
Modify Parameters of Concurrent Methods (MPCM)	MXT – Modify Method-X Time (wait(), sleep(), join(), and await() method calls)
	MSP – Modify Synchronized Block Parameter
	ESP – Exchange Synchronized Block Parameters
	MSF – Modify Semaphore Fairness
	MXC – Modify Permit Count in Semaphore and Modify Thread Count in Latches and Barriers
	MBR – Modify Barrier Runnable Parameter
Modify the Occurrence of Concurrency Method Calls (MOCMC)	RTXC – Remove Thread Method-X Call (wait(), join(), sleep(), yield(), notify(), notifyAll() Methods)
	RCXC – Remove Concurrency Mechanism Method-X Call (methods in Locks, Semaphores, Latches, Barriers, etc.)
	RNA – Replace NotifyAll with Notify
	RJS – Replace Join with Sleep
	ELPA – Exchange Lock/Permit Acquisition
	EAN – Exchange Atomic Call with Non-Atomic
	RTS – Remove Thread Start
	MTS – Move Thread Start
Modify Keyword (MK)	ASTK – Add Static Keyword to Method
	RSTK – Remove Static Keyword from Method
	RSK – Remove Synchronized Keyword from Method
	RSB – Remove Synchronized Block
	RVK – Remove Volatile Keyword
	RFU – Remove Finally Around Unlock
Switch Concurrent Objects (SCO)	RXO – Replace One Concurrency Mechanism-X with Another (Locks, Semaphores, etc.)
	EELO – Exchange Explicit Lock Objects
Modify Critical Region (MCR)	SHCR – Shift Critical Region
	SKCR – Shrink Critical Region
	EXCR – Expand Critical Region
	SPCR – Split Critical Region
	WTIS – Replace While With If Inside Synchronized

execution is different from the expected, a defect was found in the program. The problem with this technique lies in finding all the synchronization sequences that can be exercised by the program tested and determining its executability.

Carver (1993) defined an approach using MET and DET - called Deterministic Execution Mutation Testing (DEMT) - in which the mutant is run deterministically. MET is used to generate different synchronization sequences of the original program. A pair {input, synchronization} is considered as test input and is applied to both original and mutant program. The mutant is forced to run a synchronization sequence, and it is killed if it cannot run the synchronization sequence or the result was different from the original program. In some cases, a sequence necessary to kill a mutant is infeasible for the program P, once the mutation inserted in the

code may change the structure of the code (e.g., removing a concurrent method or keyword). Therefore, it is necessary to determine the expected result for the forced execution of P with $\{\text{input, synchronization}\}$. The expected result includes the expectation of the synchronization to be executable, the termination condition of P , and the expected output. As MET technique is used to generate new synchronization sequences, it is not guaranteed that all synchronization sequences will be generated.

Offutt *et al.* (1996) present an approach in which the original program is executed n times with the same test case T to create a subset of n possible executable outputs. After that, each mutant is freely executed, and the output generated is compared with the subset. The mutant is killed if the output is not present in the subset and do not consider the synchronization sequence followed. In this approach, the mutants are freely executed, and that is what differentiates it from the approach defined by Carver (1993). This approach has the problem of mutants that can be erroneously killed, once it is not guaranteed that all possible outputs of the original program were found using MET. Another problem is related to mutants that remain alive because the synchronization sequence followed is not capable of showing the presence of the defect.

Silva-Barradas (1998) defined the Behavior Mutation Analysis approach based on behaviors that a program can show. A behavior is composed of a synchronization sequence and an output. Initially, the approach works as DMT, in which the mutant is forced to execute a synchronization sequence and is killed if the output obtained was different or if the mutant was incapable of following the sequence. Then, for each alive mutant, an input and a synchronization sequence are generated with the objective to reach the mutation point of the code, i.e., where the mutation was injected, and the mutant is killed if the result obtained was different from the one of the original program, solving the problem of erroneously killing a mutant.

Delamaro (2004) developed an approach for reproducing the run of a concurrent Java 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 recording phase. In the playback phase, the synchronization sequence guides the next event to be run when a thread enters a synchronized point. The approach was effective for replicating concurrent programs in Java with instrumentation.

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 “dispute 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 certain 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.

Silva (2013) and Silva *et al.* (2012a) developed two approaches to support mutation testing based on the DEMENT and MET approaches. Reachability test is used for both approaches in the running of all synchronization sequences from a test input, generating a set of all possible output. The Adapted DEMENT approach runs the mutant deterministically, and the output is compared with the set of all possible output. The mutant is killed if it presents a different output or could not follow the synchronization sequence. The Adapted MET approach runs the mutant freely several times with the objective to perform different synchronization sequences, and the mutant is killed if it shows an output not presented by the original program. Reachability testing solves the problem of erroneously killing a mutant, once all possible solutions for the original program are found, however, the problem comes at the cost of generating and running all possible synchronization sequences.

Souza *et al.* (2015) developed a composite approach that uses reachability testing to guide the selection of the synchronization sequences for covering a specific structural testing criterion. The information about synchronization coverage is used for the selection of a particular test case to be part of the test case set. The objective is to reduce the test activity costs by reducing the number of tests necessary for code coverage. The authors described an experiment that evaluates the cost and effectiveness of the composite approach in comparison to both structural and reachability testing.

Despite the benefits of techniques to deal with the non-determinism, the computational costs are high for the execution of different synchronization sequences for each input. This cost increases in the mutation testing, once each mutant has different synchronization sequences that must be considered during the test.

2.6 Verification of concurrent programs

System verification is used to establish that the design or product under consideration possesses certain properties (BAIER; KATOEN, 2008). An example of a property a system must hold is that it should never be able to reach a situation in which no progress can be made, i. e. representing a deadlock scenario. To verify the system, properties are obtained from the specification in which it is defined what the system must and must not do. If the system fails to fulfill one of the properties specified, a bug is found. The system is considered correct if all properties are satisfied, but this does not guarantee that the system does not present an error, for instance, one property that may be erroneously specified due to inconsistent system requirements.

Model-based verification techniques are based on models describing the possible system behavior in a mathematically precise and unambiguous manner (BAIER; KATOEN, 2008). Algorithms are used to explore all states of the model systematically, and different verification techniques are used to explore the model, e.g., simulation (execution of scenarios in the model), testing (real execution) and model checking (exhaustive exploration).

2.6.1 Model checking

Model checking is a verification technique that explores all possible system states in a brute-force manner (BAIER; KATOEN, 2008). By systematically exploring all states in the model, it is possible to affirm that a system satisfies a certain property. Model checking is an automatic approach for verifying systems with finite states, and its performance is proportional to the size of the system state space. Therefore, one challenge in model checking is dealing with the state space explosion problem. The number of states grows exponentially for a system with many threads interacting with each other or data structures with many different values. For instance, a concurrent program with p processes can have a state graph of size $\exp(p)$.

Since its development in 1980s by Clarke and Emerson (1982) and Queille and Sifakis (1982), different model checking approaches have been developed, e.g. probabilistic model checking (HANSSON; JONSSON, 1990), symbolic model checking (BURCH *et al.*, 1992), bounded model checking (CLARKE *et al.*, 2001) and explicit model checking (EDELKAMP *et al.*, 2001). Besides that, different techniques have been developed to improve model checking approaches, e.g., abstraction, binary decision diagrams (BRYANT, 1986), partial order reduction (PELED, 1994), and symmetry reduction (EMERSON; SISTLA, 1997). The model checking process is composed of three main activities: system modeling, specification, and verification.

2.6.1.1 System modeling

The first step of the model checking process consists in the conversion of the system design into a formal form accepted by the model checking tool. Models describe the behavior of the system accurately and unambiguously and are expressed using finite-state automata, consisting of a set of states and transitions (BAIER; KATOEN, 2008). A state is an instantaneous description of the system, and a transition is the result of some action of the system that makes the state change. A state transition graph called Kripke structure is used to model a system formally.

2.6.1.2 Specification and verification

A specification is a property that the system's design must satisfy, and the specification activity ensures that the properties that the design must satisfy are stated. Properties are described using a property specification language, e.g., temporal logic, which models the behavior of the system over time. As a result, the specification may embrace system properties such as functional correctness, reachability, safety, liveness, fairness, and real-time properties (BAIER; KATOEN, 2008).

A safety property, for instance, states that "something bad never happens" and include the absence of deadlocks and states that make the system to present an error. A deadlock can be identified if the complete system is in a terminal state, although at least one process is in a (local) nonterminal state (YANG, 2009). To detect safety properties, a depth-first search can be

used to verify states, and if an undesirable state is found, the trace that leads to this state can be used as a counterexample.

A deadlock scenario occurs when processes mutually wait for each other to progress. Safety property can be stated as a logic formula of the atomic propositions. In a mutual exclusion problem, where two process access a same critical region, a safety property is that both processes cannot be in the critical section at the same time, defined as $\neg (c_{s0} \wedge c_{s1})$. A depth-first search can be done in the state space to validate this property. During the search, if a state is undesirable, then an evidence trace was detected as a counterexample.

A liveness property states that “something good will eventually happen” and is violated in infinite time by infinite system runs. Liveness properties are modeled using Temporal Logics, e.g. Computation Tree Logic (CTL) (CLARKE; EMERSON, 1982) and Linear Temporal Logic (LTL) (SISTLA; CLARKE, 1985). In a mutual exclusion algorithm, it is possible to specify tree liveness properties: (1) (eventually) each process will eventually enter its critical section; (2) (repeated eventually) each process will enter its critical section infinitely often; (3) (starvation freedom) each waiting process will eventually enter its critical section.

2.6.2 Java PathFinder tool

Java PathFinder (JPF) is an explicit-state model checker for Java programs and works directly in Java bytecode (BRAT *et al.*, 2000). As a model-checking tool, it identifies a point in the program from where execution could proceed differently and systematically explores them all. Typical choices are different scheduling sequences or random values. Fig. 8 presents the JPF-core - a Java virtual machine that supports model checking. As input, the JPF receives the Java bytecode and a configuration file in which are defined the properties that the tool must verify. The model checker has two modules, namely the model checking Java virtual machine (MC-JVM) that executes the bytecode and the depth-first algorithm that goes through the state-graph. The algorithm informs the MC-JVM to move forward one step, move backward one step and evaluate the current state (BRAT *et al.*, 2000).

Figure 8 – Overview of JPF-core

The JPF-core checks two non-functional properties, namely deadlocks and unhandled

exceptions. However, the user can define new properties using plug-ins (listeners) that monitor all actions taken by JPF.

JPF, as a model checker suffers from the state explosion problem. To handle this problem, JPF employs four techniques to improve its performance. With symmetry reduction, JPF evaluates the states, and if it has already been explored, the path is abandoned and backtracked to a previous choice point that has still unexplored choices. JPF has an automated abstraction tool that converts the Java program into an abstracted program. The user specifies abstraction criteria by adding or removing variables in the source code and adding abstract variables. Therefore, the user abstracts complicated sub-components, avoiding a global abstraction.

Static analysis is a technique that can be used to reduce the size of a program. Three techniques are used in JPF: Slicing and partial evaluation that generate a functionally equivalent smaller program and partial order reduction to identify statements that can safely be executed in parallel. The last technique is runtime analysis that executes the program once and extracts information based on the trace executed. That information is used to predict different traces that may violate properties of interest.

2.6.2.1 JPF extensions

JPF was designed in a modular way so that it is easy to extend. Several “JPF extensions” can be used as plug-ins to the JPF-core module that is always required. JPF extensions implement features such as different execution modes, special properties or alternative library models and can be found in the JPF repository (JPF, 2017). Two extensions were used for the execution of the experiments in Chapter 4 and Chapter 5, namely `jpf-shell` and `jpf-inspector`.

`jpf-inspector`

JPF-Inspector is a tool for monitoring and control of JPF during traversal of the program state space (JANČÍK *et al.*, 2011). It is a tool for debugging Java programs and allows inspection and control of JPF execution. The user can examine and modify program state and control thread scheduling explicitly. Inspector provides a mechanism for exploring program state, including values of heap objects and call stacks of individual threads. A custom expression language is used internally to allow the navigation over references (object tree). One important feature is the possibility to run Inspector in a mode where it prints all options (enabled transitions) at each non-deterministic choice, and the user must select one of them. Besides that, Inspector allows the record of commands to be replayed later in another execution.

`jpf-shell`

The `jpf-shell` provides a basis for the development of GUIs (Graphical User Interface) for JPF. The advantage of using `jpf-shell` is the possibility to launch a graphic inter-

face for JPF and minimal need to modify current property configurations for JPF. For its utilization, it is necessary to add a specification in the configuration file (.jpf). The specification `shell=.shell.basicshell.BasicShell` permits the use of JPF with a graphic interface and `shell=.shell.frontends.cmd.CommandLineShell` to use JPF in a command line terminal. Jpf-shell is required to execute jpf-inspector.

2.7 Search-Based Software Engineering

Meta-heuristic techniques are used to solve optimization problems, and however, they are not suitable for all kind of problems due to complexity, execution time and cost. Talbi (2009) defends the use such techniques when the problem shows an internal complexity, i.e., with a high number of possible solutions, which avoids the use of an exhaustive technique. Meta-heuristic techniques are not complete algorithms, but strategies to be adapted to specific problems.

Glover (1986) was the first author to introduce the term meta-heuristic (also called generic heuristics), defining it as a set of search algorithms to be used in different problems. The search process for the optimization of problems is inspired in different contexts, as Simulated Annealing technique (KIRKPATRICK *et al.*, 1988), which uses concepts of metallurgy, and Genetic Algorithm (HOLLAND, 1992), which uses fundamentals of biology and genetics (e.g., crossover, mutation, and evolution). The use of meta-heuristics can be justified when the internal complexity of a problem hampers the use of an exhaustive technique due to the large number of possible solutions (TALBI, 2009).

Glover and Kochenberger (2003) addressed the general process of execution of a meta-heuristic in which a search for solutions is guided by a fitness function. A fitness function is a mathematical function that assigns values to each solution present in the search space. When an exhaustive technique is applied, all solutions are visited, and the best solution is returned. On the other hand, when a meta-heuristic is applied, only some solutions are visited during the search for the optimal solution. Therefore, each meta-heuristic follows a specific search process, but hopefully in a clever way so that good solutions can be found. Since only a few solutions are visited during the search, there is no guarantee the best solution for the search space will be returned.

Meta-heuristics are classified into two categories: mono-objective and multi-objective. The difference between them is the number of objective functions that are managed by the technique. An objective function is used to consider whether a solution is good or not by using an ordinal scale in which is possible to compare two solutions. To solve a search problem, it is necessary to find a set of parameters that maximizes (or minimizes) an objective function (or functions).

In the mono-objective optimization, the search considers only an objective function and in multi-objective optimization, the search considers several objective functions. Some

examples of mono-objective optimization are Hill-Climbing (BLUM; ROLI, 2003), Simulated Annealing (KIRKPATRICK *et al.*, 1983), and Genetic Algorithm (HOLLAND, 1992). Regarding the multi-objective optimization, the *NSGA-II* (Non-dominated Sorting Genetic Algorithm II) is one of them.

The use of meta-heuristics for the optimization of problems in software engineering began in 1976 (MILLER; SPOONER, 1976) with the generation of test data using numerical maximization. However in 2001 (HARMAN; JONES, 2001) the term *Search based Software Engineering* was employed and from then on, several works were developed investigating the union of Software Engineering and search algorithms.

Freitas *et al.* (2009) conducted a bibliographic review on works related to the use of meta-heuristics in Software Engineering. As a result, 23 papers that investigate several problems of Software Engineering were found. These papers are related to eight categories in software engineering, namely Requirements Engineering (Requirements Analysis and Requirements Selection), Software Testing (Test Data Generation, Test Case Selection, Test Case Prioritization, Non-Functional Testing, and Functional Testing), Software Estimate (Estimation of Size and Estimation of Software Cost), Project Planning (Resource Allocation), Source Code Optimization (Parallelization), Software Maintenance (Software Re-engineering and Automated Maintenance), Service-Oriented Software Engineering (Software Development), and Compiler Optimization (Heap Allocation and Code Size).

2.7.1 Meta-heuristics search techniques

Genetic Algorithm (HOLLAND, 1992) is a search technique that uses concepts of evolutionary biology, such as heredity, mutation, natural selection, and crossover. Its operation is summarized in the use of two genetic operations, namely crossover and mutation. An individual represents a possible solution to a problem and is coded as an array of characters called chromosome. Fig. 9 shows the execution flow of the GA, which consists of 6 main tasks.

Figure 9 – Execution flow of the Genetic Algorithm

In the first task, the initial set of individuals (possible solutions to the problem) is generated randomly, and the number of individuals of the population is set by the user. A fitness function is a mathematical function is used to assigns values to each individual present in the

initial population. The value of it is related to how fit it is in the solving of the problem. For example, if a performance problem must be solved, solutions with low execution time have higher fitness value.

The individuals with the best fitness value are selected in the second task to be crossed in the third one. Crossover consists in crossing the information of two solutions for the generation of new solutions. Greedy Crossover is a specific type of crossover that can be applied only if all genes in the chromosome are different, the set of genes for both chromosomes is identical, and only their order varies. The operator selects the first allele of a parent, compares the next allele of both parents and chooses the closest one. If this allele has already appeared in the chromosome, the operator selects the one of the other parent. If both alleles have appeared, the operator randomly selects a non-selected allele.

Fig. 10 shows an example of the generation of a new individual. The mother individual is used as a template so that the first allele (value 3) is selected. The next allele of each parent is then compared, and the closest one is selected. In this case, between 4 and 6, the closest is 4; between 5 and 6, 5 is selected; between 6 and 2 and 3 and 1, 2 and 1 are chosen, respectively. The missing allele is 6, which is selected for finishing the chromosome.

Chromosome						
1	2	3	4	5	6	7
Allele						
Crossover				Mutation		
Mother:	3	5	2	1	4	6
Father:	2	3	4	5	6	1
Child:	3	4	5	2	1	6
				3	4	5
				3	4	6

Figure 10 – Representation of a chromosome, Greedy crossover and Swapping mutation operators

The Mutation process randomly selects and changes a given individual. This procedure generates a different solution that was not coming from the crossing of the individuals of the population. The objective is to avoid the stagnation of the search and explore different locations of the search space. The Swapping mutation operator selects two alleles of the chromosome and swaps them instead of mutating them. Fig. 10 shows an example of the mutation of a chromosome, in which, alleles 5 and 6 are selected and swapped.

Hill-Climbing technique (BLUM; ROLI, 2003) operates as a method for locally finding a solution. The name of this meta-heuristic corresponds to its operation, in which a solution is selected and from this, the algorithm searches for better ones. Fig. 11 presents the concept applied in the algorithm. Being considered a simple search strategy, the quality of the solutions returned may not be high. This algorithm does not run through the entire search space and, as a result, it can only find a good local solution, but not the optimum global solution (of the entire search space).

Figure 11 – Hill-Climbing search process (FREITAS *et al.*, 2010).

The Simulated Annealing technique (KIRKPATRICK *et al.*, 1983) is inspired by the physical process of cooling or annealing a material. The process has two steps; the first is to raise the temperature of the solid to a temperature of 1100°C. In the second step, cooling is performed slowly, causing the atoms to organize into a uniform structure with minimal internal energy, that is, together with great attraction energy and small repulsion energy.

Comparing it with the algorithm, a search from a predefined solution is performed in the neighborhood to find better solutions, as in Hill-Climbing technique. The difference is a variable T used in the search. This variable corresponds the temperature, and how high it is, more is the probability of the algorithm to accept a worse solution. Therefore, the algorithm searches in a different space and as the value of the T decreases, the probability of accepting a worse solution also decreases. Fig. 12 shows the operation of this algorithm, in which the objective is to find the smallest value for the fitness function (f).

Figure 12 – Simulated Annealing technique (FREITAS *et al.*, 2010).

2.7.2 Search Based Software Testing

Studies in the area of Search Based Software Engineering (SBSE) have shown the application of meta-heuristics is promising in the software testing context (FREITAS *et al.*, 2010). Search Based Software Testing (SBST) describes the use of meta-heuristic techniques for supporting software testing and Harman *et al.* (2015) provided data on the increase in the

number of publications on SBST. According to the authors, if the trend continues, over 1,700 SBST papers will have been published before the end of this decade.

Some issues addressed by SBST are test data generation (KOREL, 1990; TRACEY *et al.*, 1998; MICHAEL *et al.*, 2001; HERMADI, 2003; KHOR; GROGONO, 2004), test case selection (ROTHERMEL; HARROLD, 1998; MANSOUR *et al.*, 2001; YOO; HARMAN, 2007; MAIA *et al.*, 2009), test case prioritization (ROTHERMEL *et al.*, 2001; WALCOTT *et al.*, 2006; YOO; HARMAN, 2007; LI *et al.*, 2007; MAIA *et al.*, 2010), functional testing (COHEN *et al.*, 2003) and non-functional testing (WEGENER *et al.*, 1997; WEGENER; GROCHTMANN, 1998; BRIAND *et al.*, 2004). Search Based Mutation Testing (SBMT) describes the use of meta-heuristic techniques for supporting mutation testing.

Silva *et al.* (2017) conducted a systematic review of SBMT and identified meta-heuristic techniques for the selection of mutation operator, test data generation, mutant generation and simultaneous mutant and test data generation for sequential programs. The application of SBST for the selection of mutation operators aims at finding a good set of mutation operators so that the number of mutants generated can be reduced without a loss in the quality of the test case set. A multi-objective meta-heuristic is used, and the mutation score and number of mutants generated are used as a fitness function. Other information, as the number of test cases and available time for the application of tests, can be used as a fitness function, because it influences the final cost of the test application.

Banzi *et al.* (2012) introduced a multi-objective approach to select mutation operators. It used mutation score and number of mutants as objective functions. The search aims at finding a set of mutation operators that maximizes the mutation score and minimizes the number of mutants. Three multi-objective algorithms - Pareto Ant Colony, MTabu Search and Non-dominated Sorting Genetic Algorithm II (NSGAI) - were applied. An experiment was conducted with six programs written in C language. The results show the algorithm based on Tabu Search achieved the best result in comparison with other techniques.

The application of SBST for test data generation in mutation testing aims at the generation of test data that kill all non-equivalent mutants. The test data generation can be reformulated as a search problem whose objective is to find a good test set among all the possible data for a given program. The application of SBST for mutant generation aims at the generation of hard to kill mutants.

Adamopoulos *et al.* (2004) addressed the concept of co-evolution. Test cases and mutants are evolved in parallel, and one set is used to improve the other, and vice versa. The methodology aims at 1) solving the problem of selecting effective mutants and test cases, 2) reducing the number of mutants generated and the computational cost associated with it and 3) preventing the generation of equivalent mutants. Genetic Algorithm is used to generate mutants and test cases and co-evolves both the population of mutants and test cases. First, a set of mutants of the original program is generated, and each mutant is validated for a fixed set of test cases. A score

related to the performance of the mutant in the test case is calculated. The higher the score, the harder the killing of the mutant. The main idea is to construct subsets of mutants to be considered individuals in the Genetic Algorithm, which evolves by the combination of subsets of mutants so that the maximum total score can be reached. The fitness function is calculated for a subset of mutants through the sum of the mutation scores of mutants present in the subset divided by the total of mutants present in it. If any mutant belonging to the subset has score 1, it has not been killed by any test case and is most likely to be an equivalent mutant. Therefore, the fitness value is penalized and receives value 0, so that equivalent mutants are eliminated. The same process is applied to the test cases. A problem of this approach is a set of mutants must be generated a priori and, in most cases, it is done randomly, and therefore, they may not be good enough to improve the quality of the test case set. Another problem is the process of scoring value 0 to the fitness function when there are mutants with score 1 in the subset. The mutant should be scored 1 because the test data set is weak, and no datum can kill the mutant.

Yue and Harman (2008) introduced the concept of subsuming HOMs (High Order Mutant). A HOM is composed of the union of two or more FOMs (First Order Mutant) and may be used in place of all FOMs that compose it, which reduces the number of mutants to be executed with no loss of effectiveness of the set of test cases. The concept of subsuming is related to a HOM harder to be killed in comparison with all the FOMs that compose it. The authors applied Genetic Algorithm, Greedy Algorithm, Hill Climbing and Random Search to find good subsuming HOMs. The fitness function used was the ratio between the “capacity to be killed” (number of test cases that kill the mutant divided by the number of all test cases) of the HOM and the sum of the “capacity to be killed” of all FOMs that compose the HOM. An experiment with six programs in C language applied 28 mutation operators and created HOMs whose orders ranged from 2 to 13. The results showed Genetic Algorithm was the best meta-heuristic to find subsuming HOMs. The authors associated its good performance with the easy generation of subsuming HOMs from other subsuming HOMs.

Domínguez-Jiménez *et al.* (2009), Domínguez-Jiménez *et al.* (2009), Domínguez-Jiménez *et al.* (2010), and Domínguez-Jiménez *et al.* (2011) created an approach called Evolutionary Mutation Testing to reduce the number of mutants using Genetic Algorithm. An individual is encoded as a mutation that can be divided into three parts: Operator, which contains the mutation operator identifier, Location, which indicates where the mutation will be applied, and Attribute, which indicates the exact modification to be made. The fitness function is calculated regarding the “weakness of the mutant” (how many mutants were killed by the same test case that killed the mutant). Therefore, equivalent mutants and hard-to-kill mutants - called strong mutants - are favored because their fitness function provides a high value.

Blanco-Munoz *et al.* (2011) extended the study conducted in Domínguez-Jiménez *et al.* (2011) and used Genetic Algorithm to generate HOMs. The concept of the individual was changed and comprehended five fields namely individual order, the value of fitness, mutation

operator identifier, an identifier of the location of the mutation to be applied and additional values that modify the behavior of the mutation operator. The fitness function is calculated by the “weakness of the mutant”.

Schwarz *et al.* (2011) defined an approach that uses Genetic Algorithm to find a set of high-impact mutants scattered throughout the code. The fitness function is calculated according to the impact of the mutant about other mutants. The impact measurement compares the execution of the original program with the execution of the mutants. A high-impact mutation damages the program execution seriously. Three impact measures were defined for the calculation of the impact of a mutant: invariant impact, impact on return values, and coverage impact. For instance, the coverage impact is calculated through the number of methods that have a difference in the coverage of the mutant in comparison with the execution of the original program. The number of mutants that modify each method of the program must be known for the calculation of their impact on a program. Such an impact is related to the number of methods the mutant can modify.

In one of the most recent studies on SBST for mutation testing, Oliveira *et al.* (2013) presented an approach that uses the concept of co-evolution addressed in Adamopoulos *et al.* (2004) with some differences: an individual continues to be a set (data or mutant), and however, each gene contains the fitness value used by the Genetic Algorithm. The fitness function is calculated through the division of the number of dead mutants by the total number of mutants. The authors created two genetic operators, namely effective son crossover and muta genes mutation. Co-evolution occurs through interactions between the best individuals of mutants and the best individuals of test data and vice versa.

Omar *et al.* (2013) applied Genetic Algorithm, Guided Local Search and Restricted Random Search to find hard-to-kill HOMs. The fitness function used the number of defects detected by the test data. The value is defined by the difference between the number of defects identified in the HOM and number of defects identified in all FOMs that compose the HOM. Regarding the fitness value, mutants can be classified as 1) Entirely Coupled HOMs, when there is no difference between the set of test cases that kill the HOM and the union of all sets of test cases that kill the individual constituent FOMs; 2) Partially Coupled HOMs, when the set of test cases that kill the HOM and the union of all sets of test cases that kill the individual constituent FOMs are not identical, but they overlap; 3) Decoupled HOMs, when they are killed by a set of test cases totally different from the union of all sets of test cases that kill the individual constituent FOMs; and 4) Subtle HOMs, when they are not killed by any test case in the given test set. They represent a new faulty behavior that has not been tested. An experiment conducted with four AspectJ programs showed Local Search was the most successful meta-heuristic technique in finding low-degree HOMs. However, Genetic Algorithm found subtle HOMs of higher degrees.

2.7.3 Search Based for testing concurrent programs

Meta-heuristics have been used in concurrent programs to search for sequences in model checking, noise injection configuration, contracts and test data. However, none of them is applied in the context of mutation testing. Tab. 4 shows the studies that apply meta-heuristic techniques in the context of testing concurrent programs.

Table 4 – Studies on the application of meta-heuristic techniques for the testing of concurrent programs

<i>Approach</i>	<i>Meta-heuristic</i>	<i>Reference</i>
Model checking	Genetic Algorithm	Alba and Troya (1996) Godefroid and Khurshid (2004)
	Ant Colony huge graphs	Alba and Chicano (2007) Chicano and Alba (2008a)
	Ant Colony huge graphs liveness	Chicano and Alba (2008b)
	Estimation of Distribution	Staunton and Clark (2010)
		Eytani (2006)
Noise injection	Genetic Algorithm	Hrubá et al. (2012a) Letko (2013)
	Hill Climbing	Křena et al. (2010)
	NSGA-II	Hrubá et al. (2014)
Contract	-	Poskitt and Poulding (2013)
Data generation	Genetic Algorithm	Steenbuck and Fraser (2013) Silva (2014)

Initially, meta-heuristic techniques were applied to concurrent programs for controlling the exploration space in the model checking technique, in which an abstract model of the program is constructed and determines all possible states and transitions between states of the concurrent program. [Alba and Troya \(1996\)](#) applied Genetic Algorithm to detect deadlocks and unused transitions and states in the Transmission Control Protocol. The individual is a trace generated by the run of the model in a simulator. The fitness function is calculated with three values. D_FITNESS value is set if a deadlock is encountered, the NV_FITNESS value is set for each unvisited state, and NF_FITNESS value is set for each unused transition. An experiment showed the approach proved good at detecting unused states and transitions. Deadlocks were also efficiently identified, and livelocks were reported.

[Godefroid and Khurshid \(2004\)](#) used Genetic Algorithm to search for deadlock and assertion violation errors. Each path generated by the run of the model is an individual, and the fitness function used for the identification of deadlock is the sum of all state transitions present in the path - the lower the value, the greater the chance of a deadlock occurring. The fitness function for the identification of assertion violation considers individuals with more assertion evaluations. The authors emphasize when message-passing protocols are analyzed, the searching for individuals with the higher number of messages sent is a good heuristic. An experiment showed GA proved significantly better than random search and systematic search.

Alba and Chicano (2007) used Ant Colony for huge graphs (ACOhg) for checking Linear Temporal Logic models. The objective is to search for an execution path that leads to an error and causes the program to fail. To guide the search for LTL security properties, the fitness function is calculated by the number of transitions required to achieve a specific node. For identification of deadlocks, the fitness value of a state is the number of active processes or the number of executable transitions. An experiment showed ACOhg achieved better results in comparison with Genetic Algorithm. Chicano and Alba (2008a) used Partial Order Reduction to reduce the memory usage of the ACOhg algorithm. Chicano and Alba (2008b) addressed the use of Ant Colony for the search of liveness violations.

Alba *et al.* (2008) employed Genetic Algorithms for the identification of deadlocks in models. Each path is an individual, and the fitness function is calculated according to the number of blocked threads generated by the path, number of transitions in the path, and a binary value, that indicates a possible deadlock. An experiment compared the approach with two exhaustive search algorithms, namely Depth First Search (DFS) and Breadth First Search (BFS). GA found short paths with a deadlock, but slowly, in comparison with DFS. GA was faster than BFS and showed paths of similar sizes. A problem with the technique is the very high use of memory. The authors solved the problem by implementing a memory to store large paths.

Rungta and Mercer (2008) developed an approach that uses warnings provided by a tool of static model analysis to search for errors in multithreaded Java programs. It also uses Greedy depth-first search for reducing the efforts required for the verification of the warnings. An individual is a sequence of locations, and Greedy depth-first algorithm guides the decisions of execution control using two heuristics. The first considers the number of locations observed in the sequence. A secondary heuristic is then used to rank states that contain the same number of locations in the sequence. The secondary heuristic guides the search toward the next location in the input sequence. An experiment showed Greedy depth-first found a larger number of errors in comparison with the Randomized depth-first search.

Staunton and Clark (2010) applied Estimation of Distribution Algorithm for the search of paths in the transition system of the code under test. The search space consists of a set of all paths in the code transition system under test. A good solution in this approach is one that leads the program to a deadlock state and is short (low number of transitions). The fitness function is calculated by the presence of a deadlock and size of the path. Paths that contain a state of error will always be favored. If both paths have an error, the smaller will be favored. If no path has an error, the path with the largest number of blocked threads is favored. An experiment showed the proposed algorithm achieved better results in comparison with the breadth-first algorithm, depth-first algorithm, and random search.

The noise injection technique inserts some delays in the program so that it runs different synchronization sequences. Eytani (2005) and Eytani (2006) applied GA to increase the probability of execution of a defect and export debugging information. Křena *et al.* (2010) and

Letko (2010) used Hill Climbing to cover concurrent errors that only appear when a particular synchronization sequence is running. The noise injection technique adds delays to the run for influencing the synchronization sequence, and however, the place where the noise must be inserted is difficult to be known.

Bhattacharya *et al.* (2012) applied Hill Climbing and Simulated Annealing to identify possible race conditions in a dynamic test. An error may occur when two or more concurrent threads access the same shared variable - at least one access is written, and no mechanism ensures exclusive access. Hrubá *et al.* (2012b) and Letko (2013) developed an approach using GA for noise injection application to concurrent programs for the run of new synchronization sequences and considered three coverage metrics, namely Synchro, Avio, and HBPair. Hrubá *et al.* (2014) applied Multi-Objective Genetic Algorithm (NSGA-II), in which several metrics can be used as a fitness function.

Poskitt and Poulding (2013) proposed contracts as the fitness function to be used while applying meta-heuristic techniques in concurrent programs in Eiffel. In a contract design, the system is seen as a set of Eiffel classes, i.e., complementary components based on precise statements of obligations, which cooperate through contracts, i.e., precise statements of obligations and mutual benefits. Eiffel directly enforces the project based on contract through constructions, such as class invariants, preconditions, and post-conditions. The authors did not report the application of the approach and the results.

Steenbuck and Fraser (2013) applied Genetic Algorithm for test data generation during a unit test. The technique produces a test set for a given Java class. Each test case consists of a prefix that initializes the class (usually a call to the constructor method), a sequence of methods (a sequence per thread), and an execution sequence deterministically performed. Silva (2014) proposed an approach using Genetic Algorithm for the automatic generation of test data for concurrent programs in MPI. The fitness function is calculated with information on the number of required elements covered by each test data. An experiment showed the approach revealed more defects in comparison with a random approach.

Meta-heuristic techniques have been effectively employed to reduce computational cost in testing of concurrent programs. Model checking approaches use a trace generated by the run of the model as an individual. The problem in those techniques is the need of an abstract model of the program before the execution. Furthermore, model checking has the problem of state explosion when the number of threads increases.

In fault injection approach, different delays are inserted in the code to execute different synchronization sequences and identify defects. The problem in this technique is the need to configure the amount of delay necessary to exercise different thread schedules and where to put it in order to make the thread scheduling change.

The use of meta-heuristic techniques in the testing of concurrent programs do not consider

mutation testing. Therefore, this lack in the search field motivates the definition of an approach similar to test data generation approach that focus on the search of synchronization sequences.

2.8 Final considerations

In this chapter, concepts about concurrent programming, software testing for concurrent programs and search based software testing were presented. It has been described the main concepts of concurrent programs, and a Java multithreaded program was shown as an example to demonstrate concurrent functions in Java. The main concepts and terminology of software testing for concurrent programs have also been described, emphasizing the mutation test criterion, due to its importance in the context of this work. Finally, the chapter addressed the main concepts of search based software testing and how it has been applied in the context of concurrent programs.

The next chapter presents a case study conducted to demonstrate the difference in the mutation score of different synchronization sequences. The objective is to observe if there is synchronization sequences more interesting, i.e., sequences that when executed may identify a high number of defects in comparison with others. This study is the base for the definition of the testing approach for the generation and selection of synchronization sequences using meta-heuristic techniques.

CASE STUDY

3.1 Introduction

In the context of concurrent programs, testing activity is complex when compared with sequential programs due to the non-determinism in which multiple runs of a program with the same input can perform different synchronization sequences and produce different results. 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 as a consequence, it is necessary to select a subset of synchronization sequences to execute. A raised problem in software testing is how to select a set of synchronization sequences and guarantee that the software is well tested.

This chapter presents a case study to investigate the Deterministic Execution Testing approach and evaluate the differences in the ability to reveal faults (measured by mutation score) of different synchronization sequences. It is essential to understand how to identify them and support the selection of synchronization sequences to be executed in the testing. The case study considers a set of nine Java multithreaded programs, a set of mutants and a set of test cases taken from Gligoric et al. ([GLIGORIC *et al.*, 2013](#)).

The remainder of the chapter is organized as follows: Section 2 describes the case study design; Section 4 analyzes the result data; Section 5 discusses the results; finally, Section 6 provides the conclusions.

3.2 Case study design

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. The higher the number of faults caught is, the better is the test case. The main research question being addressed by this study is:

- RQ1: What are the benefits of executing Deterministic Execution Testing of concurrent programs with different synchronization sequences?

The objectives of the case study are:

- To apply the Deterministic Execution Testing in a set of concurrent programs
- To learn from the application about the following:
 - Impact of synchronization sequences
 - Benefit of using Deterministic Execution Testing

In the deterministic execution of programs, one test case is composed of an input and a synchronization sequence. The number of faults caught by the execution is used as a metric to evaluate the effectiveness of a synchronization sequence.

The case study is a strategy for doing research that involves an empirical investigation of a particular contemporary phenomenon within its context using multiple sources of evidence (ROBSON, 2002). For this case study, we are investigating the effectiveness of the Deterministic Execution Testing approach, in which several synchronization sequences are executed to increase the reliability of the test case.

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 faults caught by the execution is used as a metric to evaluate the effectiveness of a synchronization sequence; therefore a set of faulty programs is used to evaluate the effectiveness of a synchronization sequence. The set of programs, test cases and faulty programs (called mutants) were obtained in the experiment conducted by Gligoric et al. (GLIGORIC *et al.*, 2013). A mutant is a version of the original program with a fault inserted based on a fault taxonomy for Java multithread programs. Tab. 6 presents the mutation operators defined by Bradbury et al. (BRADBURY *et al.*, 2006) and used to inject faults in the programs. These mutation operators represent the mistakes programmers make when developing concurrent code in practice. The presence of bugs in concurrent code can have severe consequences including deadlock, starvation, livelock, dormancy, and incoincidence (calls occurring at the wrong time) (LONG *et al.*,). Tab. 5 presents the multithread programs, the number of mutants and the number of test cases for each program.

This case study is an embedded case study where multiple units of analysis are studied within a case. The context of the study case is the application of Deterministic Execution Testing

Table 5 – Objects programs used in the study

Objects	#LOC	#Mutant	#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

Table 6 – Mutation operators for Java multithread programs

Category	Concurrency Mutation Operators
Modify Parameters of Concurrent Methods	Modify Method-X Time
	Modify Synchronized Block Parameter
	Exchange Synchronized Block Parameters
	Modify Semaphore Fairness
	Modify Permit Count in Semaphore and Modify Thread Count in Latches and Barriers
	Modify Barrier Runnable Parameter
Modify the Occurrence of Concurrency Method Calls	Remove Thread Method-X Call
	Remove Concurrency Mechanism Method-X Call
	Replace NotifyAll with Notify
	Replace Join with Sleep
	Exchange Lock/Permit Acquisition
	Exchange Atomic Call with Non-Atomic
	Remove Thread Start
	Move Thread Start
Modify Keyword	Add Static Keyword to Method
	Remove Static Keyword from Method
	Remove Synchronized Keyword from Method
	Remove Synchronized Block
	Remove Volatile Keyword
	Remove Finally Around Unlock
Switch Concurrent	Replace One Concurrency Mechanism-X with Another
	Exchange Explicit Lock Objects
Objects Modify Critical Region	Shift Critical Region
	Shrink Critical Region
	Expand Critical Region
	Split Critical Region
	Replace While With If Inside Synchronized

approach in a set of Java multithread programs. As each program has its test cases and mutants, each one is considered a unit, i.e., an object of study. Fig. 13 presents an overview of how the case study is organized.

Figure 13 – Overview of the context, case, and units of analysis

3.2.1 Case study protocol

The protocol used in the case study follows the procedure used in the Deterministic Execution Testing described as follows: for each program (unit), one test case is selected from the test case set. 1) The execution of the program with the test case. If the output of the execution is correct, the synchronization sequence is collected and stored. 2) The synchronization sequence stored is executed with all the mutants in the mutant set, and the result is stored. 3) The data collected is analyzed, and the mutation score (number of mutants killed) 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. Fig. 14 presents an overview of each iteration of the protocol.

Figure 14 – Overview of the protocol of the case study

JPF-inspector tool was used to support the deterministic execution of the program and mutants. The deterministic execution of the original program was necessary to generate different

synchronization sequences. If a free execution of the program were performed, it was possible to get the same synchronization sequence for all the executions. Therefore, the deterministic execution was used to guarantee the execution of different synchronization sequences. However, as the selection of the threads to be executed was made randomly, we called this execution as pseudo-random.

JPF-Inspector is a tool for debugging Java programs and allows inspection and control of the Java PathFinder (JPF) model checker (JANČÍK *et al.*, 2011). Java PathFinder works directly in Java byte-code, and as a model-checking tool, it identifies a point in the program from where execution could proceed differently and systematically explores them all. Typical choices are different scheduling sequences or random values. The user can examine and modify program state and control thread scheduling explicitly. Using JPF-Inspector, the command *enable ask scheduling choice generators* is invoked and it allows the user to select the next thread to be executed. The debugging tool stops and presents the possible threads to be executed, asking the user to specify which choice to use and presents the range of threads that can be selected (*first thread - last thread*). It is possible to select a thread using the command *cg select <index>*.

Fig. 15 illustrates the execution of a program. The thread with the symbol `>` was the one selected to be executed. A synchronization sequence is different from another if the thread selected to execute is different in one point, e.g., a different synchronization sequence can be achieved if the thread 1 is selected in the second point instead of 0. During the first step of the experiment, five synchronizations sequences were generated by selecting random threads to execute. A set of 5 synchronization sequence was generated for each test case of each program.

```

JPF Shell - AccountTest.jpf
commence

Properties Config Site Report Test Output Verify Output Logger JPF Inspector Breakpoints

This is the JPF Inspector console for debugging the target "mtmt.account.AccountTest".
Type "hello" to test if the Inspector is working or "help" to get a list of commands.

cmd>enable ask scheduling cg
cmd>enable ask scheduling choice_generators

cmd>run
cmd>run
INFO: JPF created and connected, SuT is started
ChoiceGeneratorAdvance - scheduling CG - ROOT (0x2ca34246) :
>0-ThreadInfo [name=main,id=0,state=RUNNING]
Execution is halted. Specify which choice to use (0-0):
Hint: Use the command 'cg select CHOICE_INDEX'.

cmd>cg select 0
cmd>choice_generators select 0
ChoiceGeneratorAdvance used values - scheduling CG - ROOT (0x2ca34246) : >1-ThreadInfo [name=main,id=0,state=RUNNING]
ChoiceGeneratorAdvance - scheduling CG - START (0x25342f09) :
>0-ThreadInfo [name=main,id=0,state=RUNNING]
1-ThreadInfo [name=Thread-1,id=1,state=RUNNING]
Execution is halted. Specify which choice to use (0-1):

cmd>cg select 1
cmd>choice_generators select 1
ChoiceGeneratorAdvance used values - scheduling CG - START (0x25342f09) : >2-ThreadInfo
[name=Thread-1,id=1,state=RUNNING]
ChoiceGeneratorAdvance - scheduling CG - SHARED_OBJECT (0x4b734e78) :
>0-ThreadInfo [name=main,id=0,state=RUNNING]
1-ThreadInfo [name=Thread-1,id=1,state=RUNNING]
Execution is halted. Specify which choice to use (0-1):

```

Figure 15 – JPF-inspector selecting thread scheduling

After that, mutants were also executed using the JPF-inspector with the command *enable ask scheduling choice generators* as shown in Fig. 15. However, the thread scheduling follows the synchronization sequences obtained in the execution of the original program. The mutants were deterministically executed to verify if the synchronization sequence used was able to identify the mutation inserted in the program, proving thus, its effectiveness. The mutant was considered dead if it presents a result different from the original program or it cannot follow the synchronization sequence.

As JPF-inspector does not provide a reliable interface for reproduction of a synchronization sequence, one interface was developed using ExpectIt tool. It is a pure Java 1.6+ implementation of the Expect tool, and it is designed to be simple, easy to use and extensible (GAVRILOV, 2018). ExpectIt is used to read the information of the threads possible to be executed and send the command with the thread to be executed. This tool is used in the execution of both original and mutant programs.

Tab. 7 presents the total number of executions for each program and for the whole experiment. Column “Exec i” corresponds to the initial execution of the program to generate the synchronization sequences and “Exec m” corresponds to the execution of the mutants.

Table 7 – Number of executions performed in this experiment

Subject	#Test	#Sync	#Mutant	Exec i	Exec m
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

3.3 Data Analysis

In the execution of mutation testing, DeMillo and Offutt (DEMILLO; OFFUTT, 1991) defined three conditions that a test case t must satisfy to kill a mutant m . (1) Reachability: when executing the mutant with a test case, the mutation point must be executed. (2) Necessity: the state of the mutant after the execution of the mutation point must be different from the state of P after the execution of the same point. (3) Sufficiency: the difference in the state of P and m immediately following the execution of the mutation point must propagate to the end of execution of P or m such that $P(t) \neq m(t)$. In this experiment, mutants were considered killed 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 types of errors were identified while applying mutation 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 mutation makes the mutant to present a different output from the expected.
2. Not enough sync - This error occurs when the synchronization sequence finishes, however, the mutant is still running.
3. Error sync - This error occurs when the mutant cannot execute the thread shown in the synchronization sequence.

Tab. 8 presents the number of errors identified in the execution of the *buffer* program. The mutants were executed with five different sequences, and the table presents the synchronization sequences that showed a difference in the type of error. For example, in the five executions of test case *T1*, three mutants cannot execute the synchronization sequence *S1*, whereas *S2* made one mutant present an error in the output and two mutants cannot execute it.

Table 8 – Mutation score for different test cases for *buffer* program

Test case	Sync	# Killed mutant		
		Error	Not enough	Error sync
T1	S1	-	-	3
	S2	1	-	2
T2	S1	-	-	-
T3	S1	-	-	8
	S2	1	-	7
T4	S1	2	-	3
	S2	3	-	4
	S3	2	-	4
	S4	3	-	3
T5	S1	2	-	3
	S2	-	-	5
T6	S1	2	-	-
T7	S1	1	-	3
	S2	2	-	2
T8	S1	1	-	3
	S2	2	-	2

Tab. 9 presents a partial result of the executions. The first column represents the program under testing. The second column corresponds to the test case id, whereas the third one presents the total number of mutants. Finally, the last column describes the number of killed mutants for the five synchronization sequences executed. The table shows the cases in which a variation in the number of killed mutants is different and confirms that there are synchronization sequences

better than others, related to the number of killed mutants. For the other test cases, the mutation score was the same in all executions.

Table 9 – Number of killed mutants per synchronization sequence

Program	Test	Mut	Mutation score				
			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

It is possible to get information about the mutants by observing individually the results. Tab. 10 presents the results of the executions for the *buffer* program. The column *alive* presents in how many executions the mutant remains alive, the column *# execution* presents the number of executions done in total. *Buffer* program has 8 test cases, and for each test case 5 executions were performed, therefore 40 executions in total. The column *dead* presents how many executions the mutant was dead and *kill score* corresponds the number of *dead* per number of all executions. In this table, it is possible to observe that mutant *M7* was not killed in an execution. It is supposed that the test set is mutation-adequate, and therefore, there is a test case (test input and synchronization sequence) that should kill the mutant. Due to synchronization sequences were randomly generated, probably the one that would lead to killing the mutant was not generated. Results indicates the existence of strong and weak synchronization sequences considering the mutation score, i.e., there are synchronization sequences that reveal more faults than others.

With the data collected, it is possible to answer the research Question 1 that investigates the benefits of executing Deterministic Execution Testing of concurrent programs with different synchronization sequences. It is possible to affirm that for this set of programs, exists synchronization sequences with different mutation score, therefore, it is essential to execute different synchronization sequences during the testing of concurrent programs.

Table 10 – Results of the Buffer program execution

Mutants	# Executions	Alive	Dead	Killing score
M1	40	20	20	50%
M2	40	25	15	37.5%
M3	40	33	7	17.5%
M4	40	20	20	50%
M5	40	23	17	42.5%
M6	40	5	35	87.5%
M7	40	40	0	0%
M8	40	25	15	37.5%
M9	40	10	30	75%

Tab. 11 presents the best and worst mutation score for test cases that exhibited synchronization sequences with different mutation score. Taking test case *T11* from *tree* program as an example, from the five synchronization sequence obtained in the initial execution, the best one killed 15 mutants, reaching a mutation score of 39%. On the other hand, the worst synchronization killed only 11 mutants, reaching a mutation score of 29%. Fig. 16 presents the number of killed mutants for each synchronization sequence of test case *T11*. Fig. 17 displays the results presented in Tab. 11.

Table 11 – Mutation score for best and worst synchronization sequences

Program	Test case	Worst score	Best score
Account	T1	0%	33%
	T12	17%	33%
Accounts	T1	0%	50%
	T3	0%	50%
Airline	T4	40%	80%
	T5	60%	100%
	T11	40%	80%
	T13	40%	80%
	T15	60%	100%
Bubble	T11	0%	100%
Buffer	T4	56%	78%
Shop	T1	38%	50%
	T2	38%	50%
	T3	38%	50%
Tree	T6	29%	37%
	T7	26%	32%
	T8	29%	37%
	T9	32%	37%
	T11	29%	39%
	T12	32%	39%

Figure 16 – Killed mutants for each synchronization sequence of test case *T11*

Figure 17 – Mutation score for best and worst synchronization sequences

3.3.1 Threats to validity

In this study, only internal and external validity threats were identified. Threats to **internal validity** inspect if the relationship between the treatment and the output is casual and is not affected by any other factor not considered and measured in the case study. The set of mutation used in the experiment is a threat in our context once they are different for each program. Mutants were generated based on mutation operators defined by Bradbury et al. ([BRADBURY et al., 2006](#)), and all of them were executed with each synchronization sequence executed in the

respective program to mitigate this threat.

External validity refers to the generalization of the results (WOHLIN *et al.*, 2012). Threats to external validity concern the ability to generalize results. A threat in this experiment 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 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).

3.4 Summary and discussion

The objective of the case study was to apply the Deterministic Execution Testing in a set of concurrent programs and investigate the impact of synchronization sequences. Therefore, the mutation score was used to evaluate if different synchronization sequences present different ability to reveal faults. As a result, it was possible to affirm that different synchronization sequences can achieve different mutation score, being crucial to the performance of the test case. Furthermore, the execution of different synchronization sequences does improve the testing effectiveness of Java multithread programs. This result is essential and motivates the investigation about how to select the best ones for using during the concurrent program testing.

In our case study, we identified three different error situations that can happen in the mutation execution. Moreover, another situation was observed in which the mutation finished its execution without error, but it does not follow the complete synchronization sequence. This situation was called “finished before”, once the mutant 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 mutant is considered as alive.

Some of the situations could be considered better, regarding the killing process in mutation testing. The primary objective of mutation testing is to obtain a different output from the mutant in comparison with the original program. In this context, the error situations found in the study can be classified due to its significance.

The more important is Situation 1, once the output present by the mutant is different from the original and means that reachability, necessity, and sufficiency conditions were reached. Situation 2 is not considered an error, once the mutant finishes its execution without showing an error even presenting a synchronization sequence shorter than the one presented by the original program. On the other hand, Situation 3 is the opposite, in which the synchronization sequence of the mutant is larger than the one presented by the original program. This situation is significant because the mutant did not finish its execution. Situation 4 is more significant than the other two, once the mutant 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. Tab. 12 presents the errors situations found in mutation testing of concurrent programs.

Table 12 – Error situations found in mutation testing of concurrent programs

Situation	Error	Description	Significance
1	Error	The mutant presents a different output from the expected	1
2	Finished before	The synchronization of the mutant is shorter than the original	4
3	Not enough sync	The synchronization sequence finishes, however, the mutant is still running	3
4	Error sync	The mutant cannot execute the thread shown in the synchronization sequence	2

In the testing of concurrent programs, the tester has a predefined set of test cases. The mutation testing evaluates the set of test cases using several faults inserted in the code. In the mutation testing procedure, the mutants are generated based on a set of mutation operators. Then, the mutants are executed with the test case set, and the output is evaluated. After that, the equivalent mutants are identified, and the mutation score is calculated. At this point, the tester must decide if (1) the mutation score reached is acceptable (1 or close to 1) or (2) new test cases should be generated to improve the mutation score. While testing concurrent programs, it is necessary to execute some (or all) synchronization sequences to evaluate the behavior of the mutants. 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 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, faults in the code are more likely to be found.

In this case study, the mutants were executed five times with each test case, and as a result, it was demonstrated that different synchronization sequences could provide different mutation scores. 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 that, a challenge in this field is how to find interesting sequences.

3.5 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 need to be considered during the test activity. The concurrency in Java is achieved by using of 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 non-determinism of concurrent programs must be evaluated through the use of testing techniques.

In this chapter, Deterministic Execution Testing approach test was used to evaluate the mutation score 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 mutants, and the mutation score was calculated. A set of three errors were defined to decide if a mutant is considered killed or not. Synchronization sequences were classified based on the mutation score as good (high mutation score) and bad (low mutation score).

As a result, it was possible to affirm that different synchronization sequences can reach different mutation score 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 even this study is focused on mutation testing, the choosing of synchronization sequences is an issue present in most testing techniques for concurrent programs. This study is an initial contribution for studies that propose selection and generation of synchronization sequences during testing activity of concurrent programs.

SEARCH BASED BEHAVIORAL MUTATION TESTING

4.1 Initial considerations

This chapter presents the definition of the SSBMuT (Search Based Behavioral Mutation Testing) approach. Therefore, experiment conducted to demonstrate the variation of mutation score in different synchronization sequences for a single test case. Section 4.2 addresses the SSBMuT approach, describing each step of it. Section 4.3 describes the instantiation of the BeMutation tool for the test of Java multithread programs using genetic algorithm as meta-heuristic technique. Finally, Section 4.5 summarizes the conclusions of this chapter.

4.2 The definition of SSBMuT approach

Several testing criteria were defined for concurrent programs, and mutation testing showed high effectiveness in revealing defects, similarly to what happens in sequential programs (DEMILLO, 1980; OFFUTT *et al.*, 1996; DELAMARO; MALDONADO, 1999; DEREZINSKA, 2003; VINCENZI *et al.*, 2006). A problem in the testing of concurrent programs is the need for the testing of multiple synchronization sequences, to evaluate if the obtained output for all synchronization is correct concerning the program specification. However, the running of all synchronization sequences is impractical due to the number of possible synchronizations and its associated cost. Approaches, as random mutation (ACREE JR., 1980), restricted mutation (MATHUR, 1991), selective mutation (OFFUTT *et al.*, 1993), and essential mutation operators (OFFUTT *et al.*, 1996a) aim at reducing the number of mutants and test cases, however, the problem of the high number of synchronization sequences remain.

The use of meta-heuristic techniques, e.g., Genetic Algorithm, Hill Climbing, and Ant Colony for solving problems can be justified when the problem shows a large number of possible

solutions that prevent the use of an exhaustive technique. A meta-heuristic technique uses only a few solutions to select the best one; on the other hand, an exhaustive technique utilizes all possible solutions.

The Search Based Behavioral Mutation Testing (SBBMuT) approach identifies synchronization sequences of concurrent programs and applies them during software testing. The approach uses a meta-heuristic technique to select a set of synchronization sequences for each test case and prevents the execution of all sequences. An interesting synchronization sequence maximizes the degree for finding an error and reduces the number of synchronization sequences to be run. [Silva et al. \(2017\)](#) identified six important characteristics while conducting experimental studies in SBMT. Such characteristics (Fig. 18) described below can be used to guide the instantiation of the approach.

Figure 18 – Important characteristics in SBMT ([SILVA et al., 2017](#))

1. **Programs** - The set of programs is the initial characteristic to be observed and defined. Several approaches have been used in the development of concurrent programs in both shared memory and message passing paradigms. ADA, PVM, Pthreads, MPI, and Java multithread are examples of languages for the development of concurrent programs.
2. **Mutation Operators** - Based on the language chosen, it is necessary to identify the mutation operators related to the language. Mutation testing was defined for Ada ([OFFUTT et al., 1996](#); [SILVA-BARRADAS, 1998](#)), PVM ([GIACOMETTI et al., 2003](#)), MPI ([SILVA](#)

et al., 2012b), Pthreads (KUSANO; WANG, 2013), and Java multithread (DELAMARO *et al.*, 2001; BRADBURY *et al.*, 2006; GLIGORIC *et al.*, 2010).

3. **Objective** - The objective of SBBMuT approach is to use search based techniques for the selection of synchronization sequences and reduces the testing costs for concurrent programs. The output of the approach is a set of synchronization sequence that can be used to support different testing approaches and techniques, such as mutation testing and structural testing.
4. **Meta-heuristic** - Different search based techniques have been used to solve problems in which an exhaustive search is impractical. Genetic Algorithm, Hill Climbing, Simulate Annealing, and NSGAI are examples of search based techniques that can be used in the search for synchronization sequences in the testing of concurrent programs.
5. **Fitness function** - The fitness function corresponds to the way in which the metrics will be used to identify and select good solutions for the problem. This is a characteristic related to the objective. It can be a single metric or a set of metrics with or without weights.
6. **Metrics** - The metric is strict related to the objective selected. Besides the mutation score used as a metric to select good sequences, if the objective is related to mutation testing, necessity, reachability, and sufficient conditions may be useful. In structural criteria, a specific node to be reached can be used. Besides that, synchronization sequence characteristics can be used as size and execution time.

The SBBMuT approach is based on mutation testing and uses mutation score as a metric to select good synchronization sequences. Therefore, the approach includes primary activities of mutation testing. The process is iterative, i.e., the approach is applied to each test case in the test case set. Fig. 19 shows an overview of the SBBMuT approach divided into four steps: (1) Program execution, (2) Generation and selection of mutants, (3) Meta-heuristic optimization, and (4) Testing execution. These steps are described in detail in the following subsections.

4.2.1 Program execution

The first step consists in the execution of the original program with the test case set. First, one test case is selected from the test data input. The entire procedure will be based on this test case. As the process is iterative, it will be applied in each test case in the test case set. The original program is run only once in this step with the test case and the information about synchronization sequence is stored to be used in the generation of new sequences (Step 3). If the program presents an incorrect output, the testing approach stops, and the program must be corrected.

In this step, the programming language used in the program to be tested must be selected. Besides that, a support tool for the execution of the program must be selected. It is essential

Figure 19 – Overview of the SBBMuT approach

for the tool to offer a feature to save the synchronization sequence executed. There are several programming languages for concurrent programs as ADA, PVM, Pthreads, MPI, and Java multithread. The tool used in the approach is related to the programming language used to develop the program. Testing tools have been developed to support the testing of concurrent programs. [Silva-Barradas \(1998\)](#) presents a Behavior Mutation Analysis prototype tool for testing ADA multithread programs, [Sarmanho *et al.* \(2008\)](#) presents the ValiPthreads tool for testing concurrent programs in Pthread, and [Hausen \(2005\)](#) presents a testing tool for MPI programs. JPF Inspector is a tool for inspection and execution control of the Java Path Finder ([JPF, 2017](#)) that applies model checking verification in Java multithread programs.

Fig. 20 presents the first step of the approach. The hot spots are the points where the approach must be instantiated depending on the user's objective. In this step, the programming language and the supporting tool depends on the testing environment. Algorithm 1 presents the pseudo-code and Tab. 13 presents the input and output of this first step.

Table 13 – Input and output of Step 1

Input	Original program Test case
Output	Output Synchronization sequence

Figure 20 – First step of the SBBMuT approach

Algorithm 1 – Original program execution

```

1: function EXECUTEORIGINAL( )
2:   for each test input do
3:     execute the input in the original program
4:     save the output
5:     save the sync-seq
6:   end for
7: end function

```

4.2.2 Generation and selection of mutants

The second step consists in the generation and selection of the mutants. Given a set of mutation operators, mutants are generated for the program under test. To reduce the number of mutants executed, an approach may be used to select a subset of mutants to be used in search of synchronization sequences.

In this step, the mutation operators are related to the programming language of the program in test. In the context of concurrent programs, mutation operators have been defined for Ada ([OFFUTT et al., 1996](#); [SILVA-BARRADAS, 1998](#)), PVM ([GIACOMETTI et al., 2003](#)), MPI ([SILVA et al., 2012b](#)), Pthreads ([KUSANO; WANG, 2013](#)), and Java multithread ([DELAMARO et al., 2001](#); [BRADBURY et al., 2006](#); [GLIGORIC et al., 2010](#)). Some techniques that reduce the number of mutants, such as random mutation ([ACREE JR., 1980](#)), restricted mutation ([MATHUR, 1991](#)), selective mutation ([OFFUTT et al., 1993](#); [GLIGORIC et al., 2013](#)) and essential mutation operators ([OFFUTT et al., 1996a](#)) may select the subset of mutants.

Fig. 21 presents the second step of the approach. The hot spots are the points where the approach must be instantiated depending on the user's objective. In this step, the mutation operators are dependent on the programming language used in the program under testing. Some techniques for the mutation reduction are dependent on the mutation operators used. For example, [Gligoric et al. \(2013\)](#) defined a set of selective mutation operators for Java multithread programs based on mutation operators defined by [Bradbury et al. \(2006\)](#). Tab. 14 presents the input and

output of this step.

Figure 21 – Second step of the SBBMuT approach

Table 14 – Input and output of Step 2

Input	Original program
Output	Mutant subset

4.2.3 Meta-heuristic optimization

Initially, the synchronization sequence collected in the run of the original program is represented as an individual, from which an initial population is formed. Then, the meta-heuristic technique searches for interesting individuals using a fitness function as a guide. The fitness function evaluates the solutions and uses metrics to compare if one solution is better than the other to solve the problem. The composition of the fitness function is related to the objective and may be represented by a metric or set of metrics combined. One metric available in the approach is the mutation score, once it measures the quality of test cases. Besides that, if the objective is related to mutation testing, necessity, reachability and sufficient conditions may be useful. In structural criteria, a specific node to be reached can be used. Finally, synchronization sequence characteristics may be used, e.g., size and execution time.

Fig. 22 presents the third step of the approach. The hot spots are the points where the approach must be instantiated depending on the user's objective. In this step, it is necessary the

deterministic execution of the mutants; therefore, a support tool is necessary. Tab. 15 presents the input and output of this step.

Figure 22 – Third step of the SBBMuT approach

Table 15 – Input and output of Step 3

Input	Original program
	Test case set
	Mutant subset
	Synchronization sequence
Output	Set of synchronization sequences

4.2.4 Testing execution

The last step consists in the using of the synchronization sequences set to improve the testing activity. The output of the SBBMuT approach is a set of synchronization sequences that can be used to support different testing approaches and techniques, such as mutation testing and structural testing. [Arora et al. \(2015\)](#) classified approaches for testing concurrent programs in 8 categories, namely reachability testing ([LEI; CARVER, 2010](#)), structural testing ([CARVER, 1993; TAYLOR et al., 1992; YANG et al., 1998; SOUZA et al., 2014](#)), model-based testing, mutation testing ([SILVA-BARRADAS, 1998; BRADBURY et al., 2006; SILVA et al., 2012b](#)), slicing based testing, formal methods, random testing, and search based testing ([HRUBÁ et al., 2014](#)). Fig. 23 presents the fourth step of the approach. The hot spots are the points where the approach must be instantiated depending on the user's objective. In this step, the user applies

the synchronization sequences set to support a testing technique. Algorithm 2 presents the pseudo-code of this step and Tab. 16 presents the input and output.

Figure 23 – Fourth step of the SBBMuT approach

Algorithm 2 – Testing execution

```

1: function EXECUTETEST
2:   for each test input do
3:     for each sync-seq do
4:       execute the program with test input and sync-seq
5:       evaluate program's output
6:     end for
7:   end for
8:   calculate testing results
9: end function
  
```

Table 16 – Input and output of Step 4

Input	Original program's output
	Test case set
	Synchronization sequence set
Output	Testing results

4.3 Implementation of the SBBMuT: BeMutation testing tool

SBBMuT is an approach to support the testing of concurrent programs. As a generic approach, it can be instantiated for different testing environments. In this section, the BeMutation (Behavior Mutation) testing tool is presented as an instance of SBBMuT approach to support mutation testing of Java multithread programs using Genetic Algorithm as meta-heuristic. The details of the SBBMuT instantiation into the BeMutation tool is described as follows.

4.3.1 Program execution

BeMutation aims at supporting the mutation testing of Java concurrent programs. Therefore, JPF Inspector model checker tool is used as support for the execution of the programs. JPF Inspector is a tool for inspection and execution control of the Java Path Finder (JPF, 2017) that applies model checking verification in Java byte-code. This tool is used to support the execution of the original program. The information about the thread scheduling is saved in a text file.

The JPF Inspector allows the tester to run a program showing the synchronization sequence performed (using the command: *enable print scheduling cg*). The information about thread scheduling is collected. Fig. 24 illustrates the execution of a program when the command *enable print scheduling choice_generators* is enabled. The thread with the symbol > was the one selected to be executed. A synchronization sequence is different from another if the thread selected to execute is different in one point, e.g., a different synchronization sequence can be achieved if the thread 1 is selected in the second point instead of 0. During the first step of the experiment, five synchronizations sequences were generated by selecting random threads to execute.

```

JPF Shell - AccountTest.jpf
commence

Properties Config Site Report Test Output Verify Output Logger JPF Inspector Breakpoints

This is the JPF Inspector console for debugging the target "mtmt.account.AccountTest".
Type "hello" to test if the Inspector is working or "help" to get a list of commands.

cmd>enable print scheduling cg
cmd>enable print scheduling choice_generators

cmd>run
cmd>run
INFO: JPF created and connected, SuT is started
ChoiceGeneratorAdvance - scheduling CG - ROOT (0x3e35922f) :
>0-ThreadInfo [name=main,id=0,state=RUNNING]
ChoiceGeneratorAdvance - scheduling CG - START (0x112ab5d) :
>0-ThreadInfo [name=main,id=0,state=RUNNING]
1-ThreadInfo [name=Thread-1,id=1,state=RUNNING]
ChoiceGeneratorAdvance - scheduling CG - START (0x54fc4c8e) :
>0-ThreadInfo [name=main,id=0,state=RUNNING]
1-ThreadInfo [name=Thread-1,id=1,state=RUNNING]
2-ThreadInfo [name=Thread-2,id=2,state=RUNNING]
ChoiceGeneratorAdvance - scheduling CG - JOIN (0x49ac57a7) :
>0-ThreadInfo [name=Thread-1,id=1,state=RUNNING]
1-ThreadInfo [name=Thread-2,id=2,state=RUNNING]
ChoiceGeneratorAdvance - scheduling CG - SHARED_OBJECT (0x4170e248) :
>0-ThreadInfo [name=Thread-1,id=1,state=RUNNING]
1-ThreadInfo [name=Thread-2,id=2,state=RUNNING]
ChoiceGeneratorAdvance - scheduling CG - LOCK (0x6aca84ae) :
>0-ThreadInfo [name=Thread-1,id=1,state=RUNNING]
1-ThreadInfo [name=Thread-2,id=2,state=RUNNING]
ChoiceGeneratorAdvance - scheduling CG - SHARED_OBJECT (0x1bc247af) :
>0-ThreadInfo [name=Thread-1,id=1,state=RUNNING]
1-ThreadInfo [name=Thread-2,id=2,state=RUNNING]
ChoiceGeneratorAdvance - scheduling CG - SHARED_OBJECT (0x1ed91a86) :

```

Figure 24 – JPF-inspector execution listing thread scheduling

4.3.2 Generation and selection of mutants

BeMutation tool is instantiated to Java multithread domain. Therefore, mutation operators for Java concurrent programs must be used to generate mutants. BeMutation has implemented

27 mutation operators (Tab. 17) defined in [Bradbury et al. \(2006\)](#) and extended in [Gligoric et al. \(2013\)](#). BeMutation has no mutation-selection approach implemented so far.

Table 17 – Mutation operators for Java multithreaded programs

Category	Concurrency Mutation Operators
Modify Parameters of Concurrent Methods (MPCM)	MXT – Modify Method-X Time (wait(), sleep(), join(), and await() method calls)
	MSP – Modify Synchronized Block Parameter
	ESP – Exchange Synchronized Block Parameters
	MSF – Modify Semaphore Fairness
	MXC – Modify Permit Count in Semaphore and Modify Thread Count in Latches and Barriers
	MBR – Modify Barrier Runnable Parameter
Modify the Occurrence of Concurrency Method Calls (MOCMC)	RTXC – Remove Thread Method-X Call (wait(), join(), sleep(), yield(), notify(), notifyAll() Methods)
	RCXC – Remove Concurrency Mechanism Method-X Call (methods in Locks, Semaphores, Latches, Barriers, etc.)
	RNA – Replace NotifyAll with Notify
	RJS – Replace Join with Sleep
	ELPA – Exchange Lock/Permit Acquisition
	EAN – Exchange Atomic Call with Non-Atomic
	RTS – Remove Thread Start
Modify Keyword (MK)	MTS – Move Thread Start
	ASTK – Add Static Keyword to Method
	RSTK – Remove Static Keyword from Method
	RSK – Remove Synchronized Keyword from Method
	RSB – Remove Synchronized Block
	RVK – Remove Volatile Keyword
Switch Concurrent Objects (SCO)	RFU – Remove Finally Around Unlock
	RXO – Replace One Concurrency Mechanism-X with Another (Locks, Semaphores, etc.)
Modify Critical Region (MCR)	EELo – Exchange Explicit Lock Objects
	SHCR – Shift Critical Region
	SKCR – Shrink Critical Region
	EXCR – Expand Critical Region
	SPCR – Split Critical Region
	WTIS – Replace While With If Inside Synchronized

4.3.3 Meta-heuristic optimization

The meta-heuristic selected to be implemented in BeMutation is Genetic Algorithm due to its capability of adaption to the problem and the good results in other papers on the field of search based mutation testing. The genetic algorithm presented in this subsection was developed based on the examples described in [Jacobson and Kanber \(2015\)](#). Algorithm 3 presents the pseudo-code of this step.

Algorithm 3 – Genetic Algorithm

```

1: initializePopulation (populationSize)
2: evaluatePopulation (population)
3: while TerminationConditionIsNotMet do
4:   select parents
5:   crossover parents
6:   mutate
7:   evaluate population
8: end while

```

4.3.3.1 *The Problem*

The problem to be solved is the finding of synchronization sequences that maximizes the mutation score of a given test case. A concurrent program is formed by different threads that communicate to solve a given problem increasing the application performance. Threads execute “read” and “write” operations in a shared memory space. A synchronization sequence is defined as the thread execution order in shared memory locations. A single difference in the thread scheduling produce a new synchronization sequence and may make the program present an error. In the mutation testing of a concurrent program, different synchronization sequences can kill a different number of mutants, as described in Chapter 3. The search for those good synchronization sequences is difficult due to the high number of possible synchronization sequences and the necessity to execute them all to calculate the mutation score.

To apply search algorithms to solve the problem, the first step is to define a representation of the valid solutions for that problem. In our case, a solution is a synchronization sequence set, which is represented as a set S of synchronization sequences S_i . Given that S is composed of n synchronization sequences, we have $S = \{s_1, s_2, \dots, s_n\}$.

4.3.3.2 *Encoding*

The entire goal of the genetic algorithm is to mimic the natural process of evolution to produce good solutions. The chromosome represents a potential solution to the target problem and is divided into multiple genes. Genes represent different aspects of a solution. Distinct genetic processes, e.g., mutation and crossover, are applied to the chromosome during the evolution process, and a natural selection is applied to select the best chromosomes based on their fitness, i.e., how optimal that solution is relative to other potential solutions.

The encoding will encode a list of synchronization sequences. Each sequence has a unique ID to reference it in a chromosome. This type of encoding is known as permutation encoding, and every chromosome is a string of numbers, which represents a synchronization sequence. Fig. 25 presents the representation of the chromosome. The real synchronization sequence is stored as a text file and accessed by the tool when necessary.

Figure 25 – Chromosome with five genes

4.3.3.3 Parameters

The primary parameters in a genetic algorithm are population size, mutation rate, crossover rate and elitism rate. In the planning of the chromosome, it is important to define the number of genes and the representation of each one. In the problem of searching synchronization sequences, each chromosome will be formed by five genes, and each gene will represent a synchronization sequence.

4.3.3.4 Initialization

This initial population is generated randomly using the information of the synchronization sequence generated in Step 1. The JPF-inspector tool is used to support the generation of the new synchronization sequences for the initial set of individuals. Using the command *enable ask scheduling choice generators* in the JPF-inspector tool, it is possible to select the thread to be executed. To generate the random synchronization sequence, ExpectIt tool ([GAVRILOV, 2017](#)) is used to identify the possible threads and select one randomly. ExpectIt is an implementation of the Expect tool for Java 1.6+ and is a program to automate interactions with programs that expose a text terminal interface.

4.3.3.5 Evaluation

In the evaluation stage, each individual in the population has its fitness value calculated and stored for future use ([JACOBSON; KANBER, 2015](#)). In the approach, an individual is formed by a set of synchronization sequences and its fitness is calculated by the union of killed mutants from synchronization sequences present in an individual. All the mutants in the set are executed with each synchronization sequence present in the set. The objective is to generate and select a set of synchronization sequence that reaches a high mutation score.

Each synchronization sequence is deterministically executed with the subset of mutants to calculate the mutation score. The JPF-inspect tool is used to help in this step. The tool provides controllability by executing the calls in the order specified. The mutant is considered as killed if it presents (1) Error - the mutant presents a different output -, (2) Not enough sync - the synchronization of the mutant is shorter than the original -, or (3) Error sync - the mutant cannot execute the scheduling thread. Fig. 26 describes the calculation of the fitness function of two chromosomes. The chromosome on the right can kill more mutants, therefore has higher fitness.

Figure 26 – Fitness calculation for two chromosomes

4.3.3.6 Termination Check

The termination condition defines when the genetic algorithm should stop the search. In the search for interesting synchronization sequences, the ideal termination would find a set of synchronization sequence with the higher mutation score. However, it is not possible to obtain this score a prior and, in most cases, a single test case is not sufficient to kill all mutants. Therefore, the termination condition is defined by the number of iterations the algorithm will execute or the number of mutation score reached one.

4.3.3.7 Selection Method and Crossover

In the synchronization sequence problem, the genes in the chromosome are important. In fact, a chromosome cannot have more than one copy of a specific gene because that would generate a set of duplicate synchronization sequences in it. The ordered crossover was used to generate new individuals. In this crossover method, a subset of the first parent's chromosome is selected and added to the child chromosome in the same position. The second parent's chromosome information is added from the end position of the selected subset and then added each gene from parent 2 not presented in the child chromosome.

Fig. 27 presents an example of the crossover operation. In the example, the last gene of Parent 2 is selected and checked if it is already in the child chromosome. As the value “1” is not presented in the chromosome, this goes to the first gene space of the child chromosome. As the end of Parent 2 was reached, the first gene, “5”, is selected and checked. As this gene is not presented, it goes to the second position. The next gene to be selected is the one with value “3”. This gene is already in the child chromosome, so that, we select the next gene with value “2” that goes to the last space in the child chromosome.

Tournament selection provides a method for selecting individuals based on their fitness value. The higher the fitness of an individual, the higher the chance that the individual will be chosen for crossover. Tournament selection selects its parents by running a series of “tournaments”. First, individuals are randomly selected from the population and entered into a tournament. Next, these individuals can be thought to compete with each other by comparing their fitness values, then choose the individual with the highest fitness for the parent (JACOBSON; KANBER, 2015).

Figure 27 – Crossover operator example

4.3.3.8 Elitism

A basic genetic algorithm will often lose the best individuals in a population between generations due to the effects of the crossover and mutation operators. However, these operators are needed to find better solutions. One simple optimization technique used to tackle this problem is always to allow the fittest individual, or individuals, to be added unaltered to the next generation's population (JACOBSON; KANBER, 2015). This avoids the loss of good individuals from generation to generation. This process is called elitism, and even if an individual is not selected to crossover, it can still be a parent in the next generation and share its genetic information with the population.

4.3.3.9 Mutation

The mutation is the last genetic operator that is applied to the population. There are many different mutation methods to be chosen. For the mutation of a synchronization sequence set, the operator randomly selects one gene and change the synchronization sequence present in it for a new synchronization sequence generated randomly, and that is not present in any other chromosome.

4.3.4 Testing execution

As a result of the genetic algorithm search, each test case is now formed by input and a set of the synchronization sequence. BeMutation testing tool was implemented to support the use of SBBMuT in the context of mutation testing of Java multithread programs. Therefore, the last step consists in the execution of this synchronization sequence in the whole set of mutants generated in Step 2. Each mutant is deterministically executed with each test case, formed by input, a set of synchronization sequence and expected outputs. A mutant is considered dead if it presented (1) Error, (2) Not enough sync, or (3) Error sync. After the execution, the mutation score is calculated, and if it was not satisfying, new test cases could be added, and the step 3

and 4 should be executed for each new test input. The complete set of mutants is executed in a controlled way, forcing the execution of the synchronization sequence generated.

4.3.5 BeMutation Architecture

BeMutation testing tool was developed in Java and uses the JPF Inspector tool as a support for the deterministic execution of the programs. Fig. 28 presents the architecture of the BeMutation testing tool. The gray-filled boxes represent the modules of the tool, described as follows:

- **Deterministic execution:** The SBBMuT approach executes the original program is executed deterministically to generate and store different synchronization sequences. JPF-Inspector tool supports this activity.
- **Mutant generation:** The BeMutation tool generates mutants based on eight mutation operators defined by Bradbury *et al.* (2006).
- **Genetic algorithm:** The genetic algorithm is used to search for synchronization sequences and to calculate the fitness function, it uses a deterministic execution of mutants.
- **Deterministic execution of mutants:** The SBBMuT uses synchronization sequences obtained by the execution of the original program and force the in the deterministic execution of mutants. JPF-Inspector tool supports this activity.

Figure 28 – Architecture of BeMutation tool

Fig. 29 presents the class diagram of the developed tool. The classes can be divided into two categories. The first one (on the left) consists of the classes used to execute the original program and the mutants. The most important class is the JPF that connects the SBBMuT with the JPF inspector tool.

The second category (on the right) apply the concepts of the genetic algorithm. Three classes were generated, namely, Individual, Population, and GeneticAlgorithm. Those classes contain the information about the functioning of the genetic algorithm. These classes are used to manage and create the population and individuals, calculate the individual's fitness, and get the fittest individual in the population. The Individual class represents a single candidate solution and is primarily responsible for storing and manipulating a chromosome.

Figure 29 – BeMutation Class Diagram

4.4 Discussion

Meta-heuristics search techniques have been used in concurrent programs to search for sequences in model checking, noise injection configuration, contracts and test data. The SBBMuT is an approach to apply the meta-heuristic techniques in the context of software testing. SBBMuT has some differences and similarities in comparison with the other techniques. Model checking approaches use a trace generated by the run of the model as an individual. The SBBMuT approach works similarly by using a synchronization sequence generated by one run of the program. The difference relies on the objective, once in model checking is to execute different states and apply structural testing, whereas in SBBMuT is to find a synchronization sequence that can reveal the highest number of defects.

In fault injection approach, different delays are inserted in the code to execute different synchronization sequences and identify defects. The BeMutation tool considers programs developed using shared memory paradigm so that different synchronization sequences can be generated without the insertion of delays. An individual in noise injection is the amount of the noise and the position where it is placed in the code. On the other hand, in BeMutation, a synchronization sequence is used as an individual.

SBBMut is similar to test data generation approach, once synchronization sequences are generated to test a program. However, SBBMut search for sequences that kill the higher number of mutants, whereas a coverage criterion is used in test data generation. BeMutation tool uses JPF-Inspector tool to generate synchronizations sequences. Therefore, only feasible synchronization sequences were generated. JPF-Inspector gives the user the option to choose the thread to be executed in each non-deterministic state so that the synchronization sequence is created by randomly selecting one thread to be executed until the end of the execution.

4.5 Final considerations

This chapter presented the SBBMuT approach. The SBBMuT was described step by step, presenting the hot spots where the approach may be instantiated to a specific domain. BeMutation testing tool is an instantiation of the SBBMuT approach to support mutation testing on Java multithread programs. BeMutation focuses on the instantiation of the genetic algorithm for the searching of synchronization sequences.

The next chapter presents an experiment to validate the approach by comparing the SBBMut approach using BeMutation testing tool with JPF model checking tool.

EXPERIMENTAL STUDY

5.1 Initial considerations

This chapter presents the definition and conduction of an experiment to evaluate the SBBMuT approach in comparison with the execution of the Java Path Finder model checking tool. Section 5.2 introduces the design of the experiment and how it was conducted. Section 5.3 demonstrates the execution of the experiment. Section 5.4 presents the data analysis and interpretation. Finally, the final considerations are given in Section 5.5.

5.1.1 Objective

Different synchronization sequences can reveal different types of errors in the testing of concurrent programs. In mutation testing, the selection of the synchronization sequences to be executed is crucial for achieving a high mutation score. In Chapter 4, the SBBMuT approach was defined in which a genetic algorithm is used together with Java Path Finder Inspector tool to generate and select a set of interesting synchronization sequences that reach a high mutation score for each test case.

The objective of the experiment is to evaluate the SBBMuT approach and compare its effectiveness against the execution of the Java Path Finder (JPF) model checking tool on the mutation testing of concurrent programs. JPF is a model checking tool for program state inspection and (systematic) exploration of non-determinism in programs. It is supposed to explore all threads interleaving, however, a limitation in the data race detector makes JPF unable to detect unsynchronized parallel accesses to fields. Therefore, interleavings of synchronized fields are not explored. This limitation reveals a gap in the testing, in which different synchronization sequences must be tested. The SBBMuT approach explores different thread scheduling in the JPF state exploration to identify errors through the use of mutation testing.

In the planning of the study, some artifacts derived from the experiment described in

Gligoric *et al.* (2013) and Dourado and Souza (2015) were used which define a set of selective mutation operators for Java multithreaded programs. Although our study uses some artifacts, e.g., programs, mutants and test cases, it is not considered a replication because it was not executed in the context of this experiment, and only the data was collected and used to compare with the SBBMuT approach. Despite this, the use of artifacts from another experiment and compare it with other proposed approaches is relevant in the software testing domain.

5.2 Experiment design

An experimental study regarding principles of Evidence-Based Software Engineering was conducted to obtain empirical evidence of the effectiveness and practical applicability of the SBBMuT approach. The experiment design follows the template defined in Melo *et al.* (2016) for experimental studies in the context of testing concurrent programs.

The design of the experiment is a factorial (1x2). It considers the effects of one factor: execution approach, each factor subdivision (factor level), and the possible combinations between the factor in their different levels (treatments).

In this study, Factor A represents the execution approach (SBBMuT or JPF). The objects of this study correspond to 14 software project used to conduct the experiment: *Account*, *Accounts*, *Airline*, *Allocation*, *Bubble*, *Buffer*, *LinkedList*, *Shop*, *Tree*, *Barber*, *H2o*, *Microall*, *Microsm*, and *Writers*.

5.2.1 Definition

The definition of the experiment objective uses fundamentals of the GQM (Goal Question Metrics) process (BASILI, 1992). This study aims to **analyze** the Search Based Behavior Mutation Testing (SBBMuT) approach (object of study) **to** compare it with the JPF tool (purpose) **regarding its** effectiveness (quality focus) **from the perspective of** the researcher (perspective) **in the context of** test specialists finding errors in their concurrent software (context).

5.2.2 Planning

After the definition of the experiment, it is necessary to plan it, determining the foundation for the experiment. This phase can be divided into six steps, namely context selection, research question definition, hypothesis formulation, variable selection, object selection, and design selection.

5.2.2.1 Context selection

The context of the experiment is a set of Java multithreaded programs, and it addresses a real problem: find a set of synchronization sequences that reach a high mutation score (and,

therefore, this set can reveal more defects in the program). In a real testing scenario, the tester would choose an approach to deal with the non-determinism inherent in concurrent programs. Techniques such as reachability testing are effective, however time-consuming. Conversely, approaches such as multiple execution testing may be faster, but not effective in most cases. Java Path Finder model checking tool explores all possible schedules and supports a limited number of thread preemptions. It works as reachability testing but uses Partial Order Reduction to reduce the state space in the concurrent program. However, even with this mechanism, it is impractical to use in large programs due to state explosion problem.

In this study, the SBBMuT approach is compared with the application of JPF. As described in Chapter 4, different thread scheduling can reveal different mutants even with the use of JPF. Therefore, this experimental study aims at identifying if the application of deterministic execution and different thread scheduling can be more effective in mutation revealing.

5.2.2.2 Research question definition

The research question was defined to evaluate the proposed mutation execution and validation scheme regarding its effectiveness in the process of mutation testing. The main research question is described below:

- **RQ1. Is SBBMuT more effective than JPF?**

Metric 1. Mutation score.

5.2.2.3 Hypothesis formulation

The basis for the statistical analysis of an experiment is hypothesis testing. The hypothesis is stated formally, and the data collected during the experiment will be used to, if possible, reject the null hypothesis (WOHLIN *et al.*, 2000). The null hypothesis states that there are no real differences in the experiment settings and that each eventual that may happen is coincidental. This hypothesis is the one the experimenter wants to reject with as high significance as possible. The alternative hypothesis is the one in favor of one difference when the null hypothesis is rejected.

This study compares the SBBMuT approach (for the generation of a synchronization sequence set) with the Java Path Finder model checking tool (JPF, 2017). This study is necessary to compare these approaches regarding mutation score to evaluate if the execution of different synchronization sequences can increase the mutation testing in Java multithread programs. Therefore, the result of the execution of both approaches is collected, and the mutation score is compared. The null hypothesis is based on the effectiveness of the mutation score of both approaches. It is possible to state the hypothesis formally and define what measure is needed to evaluate the hypothesis stand on the informal statement afore-described.

- Null Hypothesis (H_0): The effectiveness of SBBMuT is equal to JPF.
- Alternative Hypothesis 1 (H_{A1}): The effectiveness of SBBMuT is different from the effectiveness of the JPF.
- Measure needed: mutation score.

5.2.2.4 Variable selection

Dependent and independent variables are used to measure the effect of treatments on the outcome of the study. The experiment's treatments are defined so that the independent variable affects the values of the dependent variable. In this study, the dependent variable observed is the mutation score. This variable was selected because it is the measurement used to evaluate the efficiency of mutation testing.

The value of independent variables does not change during the execution of the experiment and should not affect the results obtained for the dependent variables. In this study, the independent variables are:

- Mutants: The set of mutants generated for each program is the same in all executions and will not change during the execution of the experiment. Those mutants were generated based on mutation operators defined by [Bradbury *et al.* \(2006\)](#).
- Test cases: The test set used to test the programs is the same in the execution of both approaches and are taken from the experiment package obtained with the program set in [Gligoric *et al.* \(2013\)](#) and [Dourado and Souza \(2015\)](#).
- Synchronization sequences: The approach used to generate the synchronization sequences corresponds to the treatment of the experiment and are composed of two tools: BeMutation and JPF.

5.2.2.5 Object selection

[Gligoric *et al.* \(2013\)](#) defined sets of selective mutation operators for Java multithreaded programs. The authors applied JPF to execute mutants and calculate mutation score. The authors affirm that JPF was used to execute all scheduling up to 6 thread preemptions. However, this was done creating different test cases with more threads running, and not changing the execution of threads.

The objects used in this study were taken from two different sources. Nine programs were taken from the experiment conducted by [Gligoric *et al.* \(2013\)](#) obtained online in the authors' web-page ([GLIGORIC *et al.*, \(2018\)](#)). Five programs were taken from [Dourado and Souza \(2015\)](#) in which the authors defined a benchmark to evaluate testing tools for concurrent software. As these programs were not part of the first group, a pre-processing was necessary. The programs

were executed using JPF, and the mutation score was stored as happened in the experiment conducted by [Gligoric et al. \(2013\)](#).

The set of programs selected is representative because they contain different communication and synchronization functions and were previously used in studies of testing multithreaded programs ([GLIGORIC et al., 2010](#); [JAGANNATH et al., 2011](#); [GLIGORIC et al., 2013](#)). Tab. 18 present the programs used in the study and some characteristics. Column LOC represents Lines Of Code, TC shows the number of test cases, and in the source column, 1 is for [Gligoric et al. \(2013\)](#) and 2 is for [Dourado and Souza \(2015\)](#).

Table 18 – Object programs used in the study

Object	#LOC	#Mutant	#TC	Primitive	#States	#Transitions	Source
Account	52	6	22	synchronized	503 691	2 343 256	1
Accounts	43	6	8	synchronized	1 645	4 679	1
Airline	38	5	18	synchronized	88 654	320 434	1
Allocation	78	5	20	synchronized	66 998	141 179	1
Bubble	37	2	12	synchronized	81 629	249 985	1
Buffer	89	9	8	synchronized	98 887	223 918	1
LinkedList	179	4	32	synchronized	2 717 344	11 610 491	1
Shop	113	8	9	synchronized	33 187	65 864	1
Tree	122	38	15	synchronized	437 695	670 085	1
Barber	177	102	1	semaphore	n/a	n/a	2
H2o	113	63	1	semaphore	n/a	n/a	2
Microall	65	15	1	barrier	n/a	n/a	2
Microsm	43	15	1	semaphore	n/a	n/a	2
Writters	145	120	1	semaphore	n/a	n/a	2

5.2.2.6 Design selection

The design of this experiment is the type “one factor with two treatments” with a paired comparison in which both treatments are applied to each object. SBBMuT approach generates a set of synchronization sequences considering different thread scheduling, and JPF executes all possible states freely for each mutant for each test case. The mutation score is compared, and the results are evaluated. Tab. 19 presents the experiment design.

Table 19 – Experiment design

Factor	Treatment	
A	SBBMuT	JPF

5.3 Operation

The procedure developed to conduct this experiment consists of three steps. For each program, each test case is selected, and the SBBMuT approach is executed to generate a set of

synchronization sequences. After that, the results are compared based on mutation score obtained. The experiment organization is composed of the following activities:

1. **Object selection:** The programs were selected from [Gligoric et al. \(2013\)](#) and [Dourado and Souza \(2015\)](#) and consists in the program set presented in Tab. 18.
 2. **Mutant generation:** Mutants from [Gligoric et al. \(2013\)](#) were generated by Comutation ([GLIGORIC et al., 2013](#)) testing tool and selected from the experiment package available online. Mutants from [Dourado and Souza \(2015\)](#) were generated by BeMutation testing tool developed for the application of SBBMuT approach. The mutants were generated based on mutation operators defined by [Bradbury et al. \(2006\)](#).
 3. **Test case generation:** The test case set was selected from [Gligoric et al. \(2013\)](#) and [Dourado and Souza \(2015\)](#).
 4. **Synchronization sequence generation using genetic algorithm:** The genetic algorithm searches for a set of interesting synchronization sequences using mutation score as fitness function. The genetic algorithm follows the subsequent steps:
 - a) The genetic algorithm (Fig. 30) is initialized with the following parameters: the population of 5 individuals each with 5 genes, the crossover rate of 0.95, the mutation rate of 0.01, and the elitism rate of 2 and the tournament size of 5.
 - b) Generation of the initial population;
 - c) Fitness evaluation of each individual;
 - d) Crossover and mutation operator application;
 - e) Evaluation of the terminal condition of the execution of 10 generations.
 5. **Mutation score calculation:** The mutation score is calculated by the union of different mutants killed by the best set of synchronization sequence.
- shown in and is

In this experiment, 14 programs were executed. Each program has a different number of mutants and, the experiment executed a total of 412 programs (original plus mutants). Furthermore, each program has a set of test cases that must be executed for each mutant. In this study, 1700 executions were necessary to test each program (original and mutant) with each test case in the respective test set, as shown in Tab. 20. Each execution of SBBMuT exercised 35 different synchronization sequences independently of the program size or the number of threads. Therefore, the number of executions was multiplied by 35. After the execution, the data about the mutation score reached by each approach was collected, so that the effectiveness of the approaches could be evaluated as shown in the next section.

Figure 30 – Genetic Algorithm procedure

Table 20 – Total of execution in the experiment for each evaluation

Program	Mutants	Test case	Executions	
			JPF	SBBMuT
Account	6	22	154	5 390
Accounts	6	8	56	1 960
Airline	5	18	108	3 780
Allocation	5	20	120	4 200
Bubble	2	12	36	1 260
Buffer	9	8	80	2 800
LinkedList	4	32	160	5 600
Shop	8	9	81	2 835
Tree	38	15	585	20 475
Barber	102	1	103	3 605
H2o	63	1	64	2 240
Microall	15	1	16	560
Microsm	15	1	16	560
Writers	120	1	121	4 235
Total	412		1 700	59 500

5.4 Analysis and interpretation

After collecting experimental data during the operation phase, it is necessary to interpret the data to conclude. Quantitative interpretation is conducted using descriptive analysis - that deals with the presentation and numerical processing of data set - and hypothesis testing, in which the hypotheses of the experiment are evaluated statically, at a given level of significance.

5.4.1 Descriptive analysis

In this section, the results of the experiment are presented and compared. Tab. 21 presents the mutation score reached using JPF and SBBMuT approaches. In this table, test cases in which the SBBMuT and JPF reached the same scores were omitted to improve the readability. In the table, it is possible to see the dominance of the SBBMuT in comparison with JPF in killing mutants for Java multithreaded programs. Fig. 31 presents a chart showing the mutation score for each approach and demonstrating the difference between the approaches, considering mutation score. The x-axis presents the programs and the test cases executed, and the y-axis demonstrates the mutation score. The score for JPF is presented in the orange line and the SBBMuT in the blue one. In the next section, statistical methods are used to prove whether this hypothesis is true.

Table 21 – Mutation score for the SBBMuT and JPF approaches

Program	Test case	Score	
		SBBMuT	JPF
Account	T1	33%	0%
	T2	33%	0%
	T3	33%	0%
	T4	33%	0%
	T16	67%	33%
	T17	67%	33%
	T18	67%	33%
	T19	67%	33%
	T20	67%	50%
	T21	67%	50%
	T22	67%	50%
Accounts	T1	50%	0%
	T2	67%	0%
	T3	50%	0%
	T4	67%	17%
Airline	T4	80%	40%
Bubble	T1	40%	0%
	T2	40%	0%
	T5	40%	0%
	T11	40%	0%
Buffer	T1	33%	11%
	T3	89%	44%
	T4	78%	44%
	T5	56%	44%
	T6	22%	33%
	T7	44%	22%
	T8	44%	22%
Program	Test case	Score	
		SBBMuT	JPF
LinkedList	T1	75%	50%
	T3	50%	25%
	T4	50%	25%
	T5	75%	50%
	T7	50%	25%
	T8	50%	25%
	T9	75%	50%
	T11	50%	25%
	T12	50%	25%
	T13	75%	50%
	T15	50%	25%
	T26	100%	75%
Shop	T1	50%	0%
	T2	50%	0%
	T3	50%	0%
	T6	50%	25%
	T8	50%	0%
Tree	T1	76%	63%
	T2	82%	79%
	T3	82%	79%
	T6	37%	29%
	T7	32%	21%
	T8	39%	29%
	T9	39%	29%
	T11	39%	29%
	T12	39%	29%
h2o	T1	70%	68%
microall	T1	60%	40%
microsm	T1	93%	87%

Figure 31 – Mutation score for SBBMuT and JPF

5.4.2 Hypothesis testing

The objective of hypothesis testing is to see if it is possible to reject the null hypothesis H_0 , based on a sample from some statistical distribution (WOHLIN *et al.*, 2000). In this study, a non-parametric test was used once the data do not follow a normal distribution. Wilcoxon non-parametric test is used in case of one factor with two treatments. The sample used in this experiment is formed by 56 items. As a result of the application of the Wilcoxon test, the sum of the positive ranks is 1585, the sum of the negative ranks is 11, the Z-value is -6.4196 and the p-value is 0. Therefore, the result is significant at $p \leq 0.05$.

SBBMuT vs. JPF. Based on the results, it is possible to accept ($H_A 1$) once the efficiency of SBBMuT is higher than the efficiency of JPF regarding mutation score.

5.4.3 Summary and discussions

The SBBMuT is an approach that uses a genetic algorithm and deterministic execution for the application of mutation testing in Java multithreaded programs. In Java multithread programs, different threads can reveal different defects. Therefore, it is necessary to test different thread scheduling (or synchronization sequences). The SBBMuT approach uses deterministic execution supported by JPF Inspector tool to execute different synchronization sequences in mutant programs. JPF model checking tool executes all possible states in a mutant. However, it is possible that the synchronization sequence performed is not capable of making the program to present a different behavior in comparison with the original program, demonstrating, thus, an error state. As shown by Tab. 21, the mutation score reached by the SBBMuT is higher than the mutation score achieved in the execution of JPF. This situation demonstrates the necessity to execute different synchronization sequences even executing all possible states from the graph.

In the execution of JPF, a mutant is considered dead when it presents a different output from the expected once the JPF executes the mutants freely (i.e., not forcing a synchronization sequence). Besides this error, SBBMuT considers other two different kind of errors due to the deterministic execution: (1) Not enough sync, when the synchronization sequence finishes,

however, the mutant is still running; and (2) Error sync, when the mutant cannot execute the thread shown in the synchronization sequence. This situation brings some discussions about the deterministic execution and free execution of mutants.

When it comes to the execution of mutants during the mutation testing of concurrent programs, there are two strands of thought. The first is the execution of mutants deterministically (DET - Deterministic Execution Testing) when the mutant is forced to execute a specific synchronization sequence executed previously by the original program. The advantage of using this technique is the possibility of defining test cases as a pair composed of an input and a synchronization sequence. If the mutant cannot follow the synchronization sequence or present a different output, it is considered as dead. The disadvantage of this technique is that once synchronization and communication structures were changed in the program, the expected behavior of the program will also change. However, this does not mean that the mutant is incorrect once it still can present a correct output. Therefore, the mutant could be killed incorrectly. A solution to this problem would be the free execution in which mutants are executed freely, and only the output is observed. This approach allows the mutants to execute synchronization sequences that may be infeasible to the original program. Techniques as MET (Multiple Execution Testing) are applied to try to kill the mutant, and however, it is not guaranteed that different synchronization sequences will be executed, returning to the problem of MET approach.

Reachability testing is too costly, MET is sometimes ineffective, and DET may be unfair, then how to select one? In the SBBMuT approach, the mutants are executed deterministically. However, the user can select how strict the killing will be. The options available to consider a mutation as killed is shown in Table 22, and they are classified as levels of strict. With these options, the user can guide the fitness function, for example, to find synchronization sequences that kill the mutants considering just one kind of error or make an incremental approach.

Table 22 – Classification of errors in the mutation testing of concurrent programs

Error	Description	Class
Error	The mutant presents a different output from the expected	Not strict
Error sync	The mutant cannot execute the thread shown in the synchronization sequence	Normal
Not enough sync	The synchronization sequence finishes, however, the mutant is still running	Strict
Finished before	The synchronization of the mutant is shorter than the original	Very Strict

In the experiment conducted in this chapter, all mutants were executed with all test cases. This situation is called the research mode of testing tools, used, mostly to conduct experiments. However, this is not the most conservative way to perform mutation testing, in which a mutant considered killed is not executed with other test cases. The BeMutation testing tool has this function too in which an incremental approach is used to select synchronization sequences not

executing dead mutants. However, a bias can be observed in this case once the approach will prioritize the synchronization sequences executed first.

During the deterministic execution of the mutants, it was observed that mutants once considered equivalents were identified and killed by the “error sync” error. For example, consider the Java class presented in Code 2 from *microall* program. If the RTX operator is applied to this class, a mutant will be created without the line “Thread.sleep(1000);”. In the context of this program, the missing line will not influence the execution of the program, once this line is used to exercise different thread scheduling. However, when the deterministic execution is applied, this missing function changes the state graph generated by the JPF, and the mutant can be identified by the “error sync” kind of error. This is good because it decreases the number of equivalent mutants that the tester must identify, being only necessary to identify equivalent mutants that do not change the state structure of the program.

```
1 class Runner1 extends Thread {
2     int i;
3     CyclicBarrier barrier;
4
5     public Runner1() {
6     }
7
8     public void setBarriers(CyclicBarrier barrier, int i) {
9         this.barrier = barrier;
10        this.i = i;
11    }
12
13    public void run() {
14        try {
15            Thread.sleep(1000);
16            System.out.println(Thread.currentThread().getName() + " waiting at barrier " + i);
17            barrier.await();
18        } catch (InterruptedException e) {
19            e.printStackTrace();
20        } catch (BrokenBarrierException e) {
21            e.printStackTrace();
22        }
23    }
24 }
```

Source code 2 – Runner1 in Java.

The SBBMuT approach uses an evolutionary technique (Genetic Algorithm) in which, instead of evolving each synchronization sequences individually, it evolves all the synchronization sequences in a set at the same time, and the fitness function considers all the synchronization sequences simultaneously. The approach starts with an initial population of randomly generated synchronization sequences. Each sequence is unique and different from the others in the set. Then, the approach uses a Genetic Algorithm to optimize the set using the mutation score as the fitness function. In the end, the synchronization sequence set with the best mutation score is shown. The objective of the search is not to find only one synchronization sequence that has

a high mutation score, in fact, the objective is to find a set of synchronization sequence that complements each other and have a high mutation score as a whole.

In the execution of the programs taken from [Gligoric et al. \(2013\)](#), the authors defined a set of test cases and tested each test case with a different number of threads. The increasing of the number of threads to get interleaving coverage is a form to “stress” the system artificially and is a common and recommended practice in testing concurrent systems ([MUSUVATHI et al., 2008](#)). Tab. 23 presents the influence of the number of threads in the number of mutants killed. It is possible to see that there is no difference in the mutation killing by increasing the number of threads in test cases executed. The SBBMuT approach, by using different synchronization sequences can decrease the use of such techniques, once it is not necessary to stress the system by using different thread numbers but test different synchronization sequences with a few threads is enough.

Table 23 – Influence of number of threads in the mutation killing

Program	Test Case	#Threads	#Killed Mutants
Airline	2TicketsFullInvariant	2	5
		3	5
	3TicketsFullInvariant	2	5
		3	5
		5	5
	5TicketsFullInvariant	3	5
		5	5
	3TicketsNotTooMany	3	5
		5	5
LinkedList	mytestAdd	2	2
		3	2
		5	2
		6	2
	mytestRemove	2	1
		3	1
		5	1
		6	1
	mytestEmpty	3	1
		5	1
		6	1
Tree	1Exist1	1	11
		2	11
	1Exist2	2	11
		3	11
	2Exist2	2	30
		6	30

Although it is efficient, the SBBMuT approach has a high cost due to the technologies implemented. The JPF tool, as being a model checker, executes all possible schedules identified by the race condition algorithm implemented in the tool. Therefore, JPF suffers from the state

explosion problem, and its execution is time-consuming even for small programs. Besides that, the genetic algorithm is a search technique that demands computational processing and executes JPF for each fitness evaluation. In this experiment, 35 fitness evaluations are done. Therefore, the SBBMuT approach was 35 times slower than a single execution of JPF. To provide a fair comparison, the JPF was executed 35 times to be the same number of executions executed by SBBMuT. Tab. 24 presents the result of multiple executions of JPF in comparison with one execution of SBBMuT. The results demonstrate that the mutation score remains the same in multiple executions of JPF.

Table 24 – Multiple executions of JPF versus SBBMuT execution

Execution	Score	
	JPF	SBBMuT
5	29%	39%
10	29%	39%
15	29%	39%
20	29%	39%
25	29%	39%
30	29%	39%
35	29%	39%

The results of SBBMuT showed in Fig. 24 presents a linear tendency in the mutation score results. This is due to four main reasons: deterministic execution, use of JPF as support, use of a synchronization sequence set as individual and complexity of programs. These points cause the approach to reach good results very fast.

1. **Deterministic execution:** The SBBMuT approach reaches different feasible synchronization sequences through deterministic execution of the original program. Therefore, it may kill more mutants in comparison with one single execution of JPF. Furthermore, the deterministic execution of mutants and the killing of mutants that cannot follow the synchronization sequence generated by the execution of the original program may increase the mutation score, once the JPF does not evaluate this kind of error.
2. **JPF as support:** The JPF is a model checking tool, and it executes all possible state path interleaving found by the race condition algorithm. Consequently, the mutation score is already high due to the several synchronization sequences executed. However, it is important to highlight that even if all state path interleaving found by the race condition algorithm, the execution of different synchronization sequences is important and may increase the mutation score.
3. **Synchronization sequence set:** One feature of the SBBMuT is the utilization of a set of synchronization sequences instead of a single one as an individual in the genetic algorithm. As a consequence, the mutation score is calculated by the sum of all single mutants killed

by the synchronization sequences set. Therefore, the set of synchronization sequences may reach a higher mutation score in comparison with a single synchronization sequence.

4. *Complexity of programs*: Due to the limitation of JPF to handle big programs, the SBBMuT approach used small programs to execute. Therefore, a small number of mutants was generated, and most of the mutants were killed with a few executions of JPF.

A more in-depth analysis of the synchronization sequences generated by the SBBMuT shows that different synchronization sequences were executed and had different mutation scores. Furthermore, it was possible to group those sequences related to the number of mutants killed and the type of the error. Tab. 25 presents the group of synchronization sequences for Test case 6 of Tree program. It is possible to identify six groups related to the number of mutants killed and the kind of error found. Fig. 32 presents a 3D scatter plot in which it is possible to see the grouping of synchronization sequences.

Figure 32 – 3D scatter plot for synchronization sequence grouping

A limitation of SBBMuT comes from a limitation on JPF. Due to the state explosion problem in model checking, JPF cannot handle the verification of big programs. Therefore, this is a limitation present in all tools that use JPF as support for model checking.

Table 25 – Synchronization sequences grouped by error for test case 6 of Tree program

Group	Sync-Seq	Error	Error sync	No error
1	Sync-Seq 1	11	4	0
	Sync-Seq 19	11	4	0
	Sync-Seq 22	11	4	0
	Sync-Seq 29	11	4	0
	Sync-Seq 31	11	4	0
2	Sync-Seq 10	11	3	1
	Sync-Seq 17	11	3	1
	Sync-Seq 30	11	3	1
3	Sync-Seq 4	11	2	2
	Sync-Seq 5	11	2	2
	Sync-Seq 8	11	2	2
	Sync-Seq 9	11	2	2
	Sync-Seq 12	11	2	2
	Sync-Seq 14	11	2	2
	Sync-Seq 15	11	2	2
	Sync-Seq 21	11	2	2
	Sync-Seq 23	11	2	2
	Sync-Seq 25	11	2	2
4	Sync-Seq 6	11	1	3
	Sync-Seq 7	11	1	3
	Sync-Seq 11	11	1	3
	Sync-Seq 13	11	1	3
	Sync-Seq 16	11	1	3
	Sync-Seq 20	11	1	3
	Sync-Seq 24	11	1	3
	Sync-Seq 27	11	1	3
	Sync-Seq 33	11	1	3
5	Sync-Seq 26	10	2	3
6	Sync-Seq 2	11	0	4
	Sync-Seq 3	11	0	4
	Sync-Seq 18	11	0	4
	Sync-Seq 28	11	0	4
	Sync-Seq 32	11	0	4
	Sync-Seq 34	11	0	4

5.4.3.1 Threats to validity

A fundamental question concerning results from an experiment refers to their validity. It ensures the results found can be considered valid for the population of interest (in this case, the academic and scientific community) (WOHLIN *et al.*, 2000). Wohlin *et al.* (2000) defined four different threats to the validity of empirical studies: construct, internal, external and conclusion threats.

Construct validity refers to the relation between theory and observation (WOHLIN *et al.*, 2000). Threats to construct validity regard the extent to which the experimental setting

reflects the construct under study. In this experiment, a threat to construct is related to the genetic algorithm setting, for example, the number of the evolution of the algorithm. This number is related to the number of fitness evaluations the algorithm will perform. If the number of evaluations is low, it is possible that the algorithm does not find the optimum solution. However, if the number of evaluation is high, the execution time will be high as well. To mitigate this thread, we chose values that can find good solutions and do not demand much time to execute.

Internal validity evaluates the relationship between the treatment and the output. Threats to internal validity concern issues that may indicate a false causal relationship among the results. The relationship between them must be casual and not an effect of a factor of which we have no control or have not measured (WOHLIN *et al.*, 2000). A threat in our context is related to the mutants generated, being different for each program. To mitigate this threat, mutants from both approaches were generated based on mutation operators defined by Bradbury *et al.* (2006). Another threat comes from the randomness of the genetic algorithm. Due to the use of heuristics in the search, it is not possible to always find the best solution. To mitigate this threat, each approach was executed five times. Besides that, statistic test was used to evaluate the hypothesis and assure the reliability of the results.

External validity refers to the generalization of the results (WOHLIN *et al.*, 2000). Threats to external validity concern our ability to generalize experimental results outside the experimental setting. A threat in this experiment is the representativeness of the programs. To mitigate this threat, 13 programs were chosen from two sources (GLIGORIC *et al.*, 2013; DOURADO; SOUZA, 2015) used in different studies in testing of concurrent programs of different size and complexity. These programs present different concurrent functions that represent the context of Java multithread programs.

The **conclusion validity** refers to the relationship between the treatment and the outcomes (WOHLIN *et al.*, 2000). Threats to conclusion validity are concerned with issues that affect our ability to draw correct conclusions about an experiment. A threat in our context is minimized by the use of this experiment plan instantiated to concurrent programs defined in Melo *et al.* (2016). Besides that, fundamentals defined in Wohlin *et al.* (2000) to minimize this threat.

5.5 Final considerations

This chapter presents an experiment to validate the effectiveness of the SBBMuT approach. The study considered 14 Java multithread program and compared the effectiveness - measured by mutation score - of the SBBMuT approach and JPF model checking tool. SBBMuT uses deterministic execution for the generation of synchronization sequences and the reproducing of those synchronizations on the mutants. On the other hand, JPF executes the mutants freely. However, there is no guarantee that different synchronization will be executed.

The results of the experiment showed that SBBMuT found a synchronization sequence set that has a higher mutation score in comparison with JPF. The conduction of this experiment was essential to validate the efficiency of the SBBMuT approach, and new experiments may be conducted to expand the results in other contexts.

CONCLUSION

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 need to be considered during the test activity. The concurrency in Java is achieved by using of 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 non-determinism of concurrent programs must be evaluated through the use of testing techniques.

In this Ph.D thesis, the research theme was the generation of synchronization sequences in the mutation testing of concurrent programs. The execution of different synchronization sequences in concurrent programs is necessary for the increase in the mutation score because different synchronization sequences can kill (or not) mutants. Approaches were defined to deal with the non-determinism in concurrent programs, e.g., multiple execution testing, reachability testing, and deterministic testing. However, there is not an excellent approach that improves both cost and efficiency. Therefore, in this thesis, a new approach was proposed to increase the efficiency of mutation testing.

Therefore, the objective of this thesis can be summarized in three main points:

1. the discovery of the existence of interesting synchronization sequences regarding mutation score in Java multithreaded programs,
2. the development of the SBBMuT approach and the BeMutation testing tool that increases the efficiency in mutation testing of Java multithreaded programs, and
3. the analyzes of the approach regarding the efficiency in reveling defects in concurrent programs in comparison with another approach.

The SBBMuT (Search Based Behavioral Mutation Testing) approach identifies and

generates synchronization sequences of concurrent programs and applies them during mutation testing. A Genetic Algorithm is used to select interesting synchronization sequences for each test case and prevents the execution of all sequences. An interesting synchronization sequence maximizes the possibility of finding an error and reduces the number of synchronization sequences to be run.

This approach is based on the statement that different synchronization sequences have different mutation scores. Therefore, it is important to select a set of synchronization sequences that can kill different mutants and together can reach a high mutation score. The use of meta-heuristic techniques is efficient when the search space is high, and an exhaustive technique proves to be ineffective.

In this context, the SBBMuT approach was defined in which combines model checking with deterministic execution and genetic algorithm to select a set of different synchronization sequences that reaches a high mutation score. The fitness function used was the mutation score. The Java Path Finder model checker tool supported the execution of the programs, and the Java PathFinder Inspector supported the deterministic execution of different synchronization sequences.

As a validation of the research, an experimental study was conducted aiming at evaluating the effectiveness of the SBBMuT approach in comparison with JPF single execution. The objective was to evaluate if the execution of different threads synchronization could increase the mutation score. The results indicate that the SBBMuT approach found a set of synchronization sequence that reaches a high mutation score.

6.1 Contributions

The main contribution of this thesis is the proposition of an approach to deal with the non-determinism of concurrent programs. As partial contributions, it is important to mention the following:

1. The definition of the SBBMuT approach for the generation and search of synchronization sequences during the testing of concurrent programs. The approach uses meta-heuristic techniques for the search of interesting synchronization sequences.
2. The implementation BeMutation testing tool for the search of synchronization sequences in Java multithread programs using genetic algorithm as a meta-heuristic. The tool implements the mutation operators defined by [Bradbury *et al.* \(2006\)](#) for java multithread programs.
3. A systematic review in the field of search based software testing in the context of mutation testing (Search Based Mutation Testing) ([SILVA *et al.*, 2017](#)).

4. The (pseudo-random) deterministic execution of programs using JPF Inspector debugging tool.
5. The deterministic execution of mutants using JPF Inspector debugging tool.
6. The definition of possible errors in the mutation testing of concurrent programs and classification of them into classes of importance.
7. The experimental results on the efficiency of SBBMuT in comparison with JPF, which can be used in other similar studies to compare the SBBMuT with other approaches.

6.2 Difficulties and Limitations

During the execution of the work, some difficulties occurred. When a testing approach in concurrent programs is defined, some aspects must be considered for the feasibility of the project. Unfortunately, the technologies available nowadays make the conduction of projects in this field challenging. We enumerate five decision points that represent the most challenging difficulties and crucial for the development of this thesis.

1. **Context:** The context means the kind of concurrent programs the approach will be applied to. There are several libraries and APIs that support concurrent programming, such as MPI (Message Passing Interface), PVM (Parallel Virtual Machine), PThreads, Java multithreading, etc. In this thesis, the first attempt was to apply our approach to programs that use MPI library. This decision was made once we had a set of mutation operators implemented and a tool that partially supported the execution of mutants. However, some difficulties arose from the set of programs used, once they are not representative and the high time needed to execute programs. After some advice from Professor Dr. Gordon Fraser, during the internship at the Sheffield University, the decision to change the context to Java multithreaded programs was made.
2. **Program selection:** To mitigate the external validity threat concerning the possibility to generalize the results of the experiment, a set of representative programs must be selected. In the conduction of this thesis, a set of representative programs were selected from the experiment conducted by [Gligoric *et al.* \(2013\)](#), and [Dourado and Souza \(2015\)](#) and were already used in other experiments conducted in the context of testing java multithreaded programs. In the attempt to increase of the program set, a search was done in the Software-artifact Infrastructure Repository ([SIR, 2018](#)), but the original version of the programs are not available, only the faulty version. This was a problem, once it was impractical to correct the programs before the use in the experiment due to the complexity of the programs.

3. **Mutation generation:** The mutation generation of the programs was difficult during the conduction of this thesis. The set of mutants available for the programs taken from [Gligoric et al. \(2013\)](#) was low, and it was necessary to generate mutants for programs taken from [Dourado and Souza \(2015\)](#). This problem was solved through the development of a subset of mutation operators in the BeMutation testing tool.
4. **Test case generation:** The set of programs taken from [Dourado and Souza \(2015\)](#) was part of a benchmark for testing concurrent tools and each program has just one test case. This was a problem once there is no guarantee that different features of the program will be exercised (and tested). Furthermore, there is a lack of testing tools for the generation of test input for Java multithread programs.
5. **Non-determinism solution:** In the context of testing concurrent programs, several approaches were defined to deal with the non-determinism inherent in such programs. However, there is a difference in efficiency and cost among them (i. e. an approach with high efficiency have a high cost and *vice-versa*). This project was based on the concept of one execution of the program. However, to the best of our knowledge, no tool supports the execution of a single synchronization sequence and that allows deterministic execution of threads in Java programs. Therefore, the Java Path Finder model checking tool was the most useful tool that fixed the approach's necessities. The problem with JPF is that it is not a testing tool, but a model checking tool, therefore, it uses a graph with states and transitions to execute all possible paths. This includes a cost in our approach, once it would not be necessary the execution of all paths, but just one with different synchronization sequences. One thing that is important to highlight is that even executing all possible paths, JPF does not execute all possible synchronization sequences due to the race condition algorithms used to identify states with possible thread interleaving.
6. **Deterministic execution:** The deterministic execution is an important aspect of the SBB-MuT approach, once it is used to generate different feasible synchronization sequences for the original program and is used to force these synchronization sequences in the mutants. The deterministic execution was a difficulty faced during the development of this project. To the best of our knowledge, no tool permits the deterministic execution of Java multithreaded programs. In this project, the JPF Inspector debugging tool was used to force different thread scheduling. However, once it is an extension of the Java Path Finder tool, one synchronization sequence is an execution of all possible states of the graph, which makes this approach costly.

6.3 Future Work

As described in Section 6.2 the use of JPF to execute the Java programs increases the cost related to the applicability of the SBBMuT approach. As a future work, the execution module of

the approach must be changed to consider only one single path in the execution of the programs. One way to do that is by changing the search algorithm implemented by JPF. As JPF can be extended, a new search algorithm can be implemented to consider only one path in the execution of the program. This can decrease the cost related to the JPF execution allowing the approach to be executed with real programs.

Another improvement in the BeMutation tool is the parallelism of the synchronization sequence execution. Once a mutant is set to execute, different instances of the JPF can be executed in parallel with different synchronization sequences. This can decrease the time for the genetic algorithm execution decreasing the time for the approach to execute.

In the experiment conducted, a set of programs was selected and executed. However, the size of these programs is small, and they can be considered as toy examples. Due to the state explosion problem found in model checking, JPF cannot execute big programs due to lack of memory or execution time high. After adaptations aforementioned, it is necessary to execute the approach with real programs. Another alternative is the JPF execution with a time-out in which just a few branches of the graph are executed in a period.

During the development of this project, only a few defined mutation operators were implemented. As future work, it would be important to implement all the operators defined by [Bradbury et al. \(2006\)](#). Also, it would be interesting to add a graph interface to the BeMutation tool, to make it available for the community.

The SBBMuT approach uses Genetic Algorithm as the meta-heuristic technique. This decision was taken based on the efficiency of this technique in the context of mutation testing and concurrent program, as shown in [Oliveira et al. \(2013\)](#), [Harman et al. \(2014\)](#). However, other techniques may be applied, and experimental studies can be conducted to evaluate the best meta-heuristic technique to be used in this context.

Due to the cost of JPF in the model checking execution of programs, an improvement in the BeMutation tool would be the substitution of the JPF for another tool in which the deterministic execution of programs can be executed. There is no necessity of a tool that does model checking verification of the program, once this aspect of the tool is not important in the execution of approach. One option is the ValiJava tool ([SOUZA et al., 2013](#)) is an alternative to be used as soon as the deterministic execution of programs be mature enough to be extracted and used as support in a new version of the BeMutation tool.

6.4 Scientific Production

As a partial result of this project, some scientific production and a book chapter were developed and are shown as follows:

1. SILVA, R. A.; SOUZA, S. R. S.; SOUZA, P. S. L. Search-based mutation testing for

- concurrent programs. In: IVWORKSHOP DE TESES E DISSERTAÇÕES DO CBSOFT. (WTDSOFT), 2014. p. 109–116. In Portuguese. ([SILVA *et al.*, 2014](#))
2. MELO, S. M.; SOUZA, S. R. S.; SILVA, R. A.; SOUZA, P. Concurrent software testing in practice: A catalog of tools. In: 6th International Workshop on Automating Test Case Design, Selection and Evaluation. (A-TEST), 2015. p. 31–40. ([MELO *et al.*, 2015](#))
 3. SOUZA, S. R. S.; SOUZA, P. S. L.; MELO, S. M.; SILVA, R. A.; VERGILIO, S. R. Teste de programas concorrentes. In: DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. (Ed.). Introdução ao Teste de Software. [S.l.]: Elsevier Editora Ltda, 2016. v. 2. ([SOUZA *et al.*, 2016](#))
 4. SILVA, R. A.; SOUZA, S. R. S.; SOUZA, P. S. L. A systematic review on search based mutation testing. Information and Software Technology, v. 81, p. 19 – 35, 2017. ([SILVA *et al.*, 2017](#))

BIBLIOGRAPHY

ACREE JR., A. T. **On mutation**. Phd Thesis (PhD Thesis) — Georgia Institute of Technology, 1980. Citations on pages [77](#) e [81](#).

ADAMOPOULOS, K.; HARMAN, M.; HIERONS, R. M. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In: **GECCO, VOLUME 3103 OF LECTURE NOTES IN COMPUTER SCIENCE**. [S.l.]: Springer, 2004. p. 1338–1349. Citations on pages [55](#) e [57](#).

ALBA, E.; CHICANO, F. Finding safety errors with ACO. In: **Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation**. [S.l.]: ACM, 2007. (CGEC), p. 1066–1073. Citations on pages [58](#) e [59](#).

ALBA, E.; CHICANO, F.; FERREIRA, M.; GOMEZ-PULIDO, J. Finding deadlocks in large concurrent java programs using genetic algorithms. In: **Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation**. [S.l.]: ACM, 2008. p. 1735–1742. Citation on page [59](#).

ALBA, E.; TROYA, J. M. Genetic algorithms for protocol validation. In: **Proceedings of the 4th International Conference on Parallel Problem Solving from Nature**. [S.l.]: Springer-Verlag, 1996. (PPSN IV), p. 870–879. Citation on page [58](#).

ALMASI, G. S.; GOTTLIEB, A. **Highly parallel computing**. [S.l.]: Benjamin-Cummings Publishing Co., Inc., 1989. ISBN 0-8053-0177-1. Citations on pages [23](#) e [30](#).

ANDREWS, G. **Foundations of Multithreaded, Parallel, and Distributed Programming**. [S.l.]: Addison-Wesley, 2001. Citation on page [29](#).

ARCURI, A.; FRASER, G. On the effectiveness of whole test suite generation. In: **Search-Based Software Engineering**. [S.l.]: Springer International Publishing, 2014. p. 1–15. Citation on page [27](#).

ARORA, V.; BHATIA, R.; SINGH, M. A systematic review of approaches for testing concurrent programs. **Concurrency and Computation: Practice and Experience**, v. 28, n. 5, p. 1572–1611, 2015. Citations on pages [24](#) e [83](#).

BAIER, C.; KATOEN, J.-P. **Principles of Model Checking**. [S.l.]: The MIT Press, 2008. Citations on pages [47](#) e [48](#).

BANZI, A. S.; NOBRE, T.; PINHEIRO, G. B.; ÁRIAS, J. C. G.; POZO, A.; VERGILIO, S. R. Selecting mutation operators with a multiobjective approach. **Expert Syst. Appl.**, v. 39, n. 15, p. 12131–12142, 2012. Citation on page [55](#).

BASIL, V. R. **Software Modeling and Measurement: The Goal/Question/Metric Paradigm**. [S.l.], 1992. Citation on page [96](#).

BHATTACHARYA, N.; EL-MAHI, O.; DUCLOS, E.; BELTRAME, G.; ANTONIOL, G.; DIGABEL, S. L.; GUÉHÉNEUC, Y.-G. Optimizing threads schedule alignments to expose the interference bug pattern. In: **Search Based Software Engineering**. [S.l.]: Springer Berlin Heidelberg, 2012, (Lecture Notes in Computer Science, v. 7515). p. 90–104. Citation on page [60](#).

BLANCO-MUNOZ, E.; GARCIA-DOMINGUEZ, A.; DOMINGUEZ-JIMENEZ, J.; MEDINA-BULO, I. Towards higher-order mutant generation for WS-BPEL. In: **2011 Proceedings of the International Conference on e-Business**. [S.l.: s.n.], 2011. (ICE-B), p. 1–6. Citation on page [56](#).

BLUM, C.; ROLI, A. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. **ACM Comput. Surv.**, v. 35, n. 3, p. 268–308, 2003. Citations on pages [52](#) e [53](#).

BRADBURY, J. S.; CORDY, J. R.; DINGEL, J. Mutation operators for concurrent Java (J2SE 5.0). In: **Proceedings of the Second Workshop on Mutation Analysis**. [S.l.]: IEEE Computer Society, 2006. p. 11–20. Citations on pages [24](#), [36](#), [43](#), [44](#), [64](#), [72](#), [79](#), [81](#), [83](#), [86](#), [91](#), [98](#), [100](#), [110](#), [114](#) e [117](#).

BRAT, G.; HAVELUND, K.; VISSER, W. Java pathfinder - second generation of a java model checker. p. 1–8, 01 2000. Citation on page [49](#).

BRIAND, L. C.; LABICHE, Y.; SHOUSHA, M. **Performance Stress Testing of Real-Time Systems Using Genetic Algorithms**. [S.l.], 2004. Citation on page [55](#).

BRYANT, R. E. Graph-based algorithms for boolean function manipulation. **IEEE Transactions on Computers**, C-35, n. 8, p. 677–691, 1986. Citation on page [48](#).

BURCH, J.; CLARKE, E.; MCMILLAN, K.; DILL, D.; HWANG, L. Symbolic model checking: 1020 states and beyond. **Information and Computation**, v. 98, n. 2, p. 142 – 170, 1992. Citation on page [48](#).

CARVER, R. Mutation-based testing of concurrent programs. In: **Proceedings of Test Conference**. [S.l.: s.n.], 1993. (ITC), p. 845–853. Citations on pages [24](#), [25](#), [40](#), [44](#), [45](#), [46](#) e [83](#).

CHAUSSUMIER, F.; DESPREZ, F.; PRYLLI, L. Asynchronous communications in MPI - the BIP/myrinet approach. In: **Recent Advances in Parallel Virtual Machine and Message Passing Interface**. [S.l.]: Springer-Verlag, 1999, (Lecture Notes in Computer Science, v. 1697). p. 485–492. Citation on page [31](#).

CHEN, Z.; XU, B. Slicing concurrent java programs. **SIGPLAN Not.**, ACM, v. 36, n. 4, p. 41–47, Apr. 2001. Citation on page [24](#).

CHENG, J. Slicing concurrent programs - a graph-theoretical approach. In: **Proceedings of the First International Workshop on Automated and Algorithmic Debugging**. [S.l.]: Springer-Verlag, 1993. (AADEBUG '93), p. 223–240. Citation on page [24](#).

CHICANO, F.; ALBA, E. Ant colony optimization with partial order reduction for discovering safety property violations in concurrent models. **Information Processing Letters**, v. 106, n. 6, p. 221 – 231, 2008. Citations on pages [58](#) e [59](#).

_____. Finding liveness errors with ACO. In: **2008 IEEE Congress on Evolutionary Computation**. [S.l.]: IEEE Computer Society, 2008. (IEEE World Congress on Computational Intelligence), p. 2997–3004. Citations on pages [58](#) e [59](#).

CHOI, S.-E.; LEWIS, E. C. A study of common pitfalls in simple multi-threaded programs. **SIGCSE Bull.**, ACM, v. 32, p. 325–329, 2000. Citations on pages [36](#) e [37](#).

CLARKE, E.; BIERE, A.; RAIMI, R.; ZHU, Y. Bounded model checking using satisfiability solving. **Form. Methods Syst. Des.**, Kluwer Academic Publishers, v. 19, n. 1, p. 7–34, Jul. 2001. Citation on page [48](#).

CLARKE, E. M.; EMERSON, E. A. Design and synthesis of synchronization skeletons using branching-time temporal logic. In: **Logic of Programs, Workshop**. [S.l.]: Springer-Verlag, 1982. p. 52–71. Citations on pages [48](#) e [49](#).

COHEN, M. B.; GIBBONS, P. B.; MUGRIDGE, W. B.; COLBOURN, C. J. Constructing test suites for interaction testing. In: **Proceedings of the 25th International Conference on Software Engineering**. [S.l.: s.n.], 2003. (ICSE), p. 38–48. Citation on page [55](#).

DELAMARO, M.; MALDONADO, J.; PEZZÈ, M.; VINCENZI, A. Mutant operators for testing concurrent Java programs. In: **XV Simpósio Brasileiro de Engenharia de Software**. [S.l.: s.n.], 2001. (SBES). Citations on pages [25](#), [43](#), [44](#), [79](#) e [81](#).

DELAMARO, M.; MALDONADO, J. C. Interface mutation: assessing testing quality at interprocedural level. In: **XIX International Conference of the Chilean Computer Science Society**. [S.l.: s.n.], 1999. (SCCC '99), p. 78–86. Citation on page [77](#).

DELAMARO, M. E. Using instrumentation to reproduce the execution of Java concurrent programs. In: **Simpósio Brasileiro de Qualidade de Software**. [S.l.: s.n.], 2004. (SBQS). Citations on pages [44](#) e [46](#).

DELAMARO, M. E.; OLIVEIRA, R. A. P.; BARBOSA, E. F.; MALDONADO, J. C. Teste de mutação. In: DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. (Ed.). **Introdução ao Teste de Software**. [S.l.]: Elsevier Editora Ltda, 2016. v. 2, p. –. Citation on page [25](#).

DEMILLO, R. Mutation analysis as a tool for software quality assurance. In: **The IEEE Computer Software & Applications Conference**. [S.l.]: IEEE Computer Society, 1980. (COMP-SAC80), p. 7. Citation on page [77](#).

DEMILLO, R. A.; OFFUTT, A. J. Constraint-based automatic test data generation. **IEEE Transactions on Software Engineering**, v. 17, n. 9, p. 900–910, 1991. Citation on page [68](#).

DEREZINSKA, A. Object-oriented mutation to asses the quality of tests. In: **Proceedings of the 29th Conference on EUROMICRO**. [S.l.]: IEEE Computer Society, 2003. (EUROMICRO '03), p. 417–. Citation on page [77](#).

DESOUZA, J.; KUHN, B.; SUPINSKI, B. R. de; SAMOFALOV, V.; ZHELTOV, S.; BRATANOV, S. Automated, scalable debugging of MPI programs with intel message checker. In: **Proceedings of the second international workshop on Software engineering for high performance computing system applications**. [S.l.]: ACM, 2005. p. 78–82. Citation on page [38](#).

DOMÍNGUEZ-JIMÉNEZ, J.; ESTERO-BOTARO, A.; GARCÍA-DOMÍNGUEZ, A.; MEDINA-BULO, I. Evolutionary mutation testing. **Information and Software Technology**, v. 53, n. 10, p. 1108 – 1123, 2011. Citation on page [56](#).

DOMÍNGUEZ-JIMÉNEZ, J. J.; ESTERO-BOTARO, A.; GARCÍA-DOMÍNGUEZ, A.; MEDINA-BULO, I. Gamera: An automatic mutant generation system for ws-bpel compositions. **Web Services, European Conference on**, p. 97–106, 2009. Citation on page [56](#).

_____. Gamera: A tool for ws-bpel composition testing using mutation analysis. **Lecture Notes in Computer Science**, v. 6189, p. 490–493, 2010. Citation on page [56](#).

DOMÍNGUEZ-JIMÉNEZ, J. J.; ESTERO-BOTARO, A.; MEDINA-BULO, I. A framework for mutant genetic generation for ws-bpel. **Lecture Notes in Computer Science**, v. 5404, p. 229–240, 2009. Citation on page [56](#).

DOURADO, G. G. M.; SOUZA, P. S. L. **Contribuindo para a Avaliação do Teste de Programas Concorrentes: uma abordagem usando benchmarks**. Phd Thesis (Master's thesis) — University of São Paulo – ICMC/USP, 2015. In Portuguese. Citations on pages [96](#), [98](#), [99](#), [100](#), [110](#), [115](#) e [116](#).

EDELKAMP, S.; LAFUENTE, A. L.; LEUE, S. Directed explicit model checking with hsf-spin. In: **Proceedings of the 8th International SPIN Workshop on Model Checking of Software**. New York, USA: Springer-Verlag New York, Inc., 2001. (SPIN 01), p. 57–79. Citation on page [48](#).

EMERSON, E. A.; SISTLA, A. P. Utilizing symmetry when model-checking under fairness assumptions: An automata-theoretic approach. **ACM Trans. Program. Lang. Syst.**, ACM, v. 19, n. 4, p. 617–638, 1997. Citation on page [48](#).

EYTANI, Y. Concurrent java test generation as a search problem. In: **5th workshop of runtime verification**. [S.l.]: Elsevier Science Publishers, 2005. p. 1–15. Citation on page [59](#).

_____. Concurrent java test generation as a search problem. **Electron. Notes Theor. Comput. Sci.**, v. 144, n. 4, p. 57–72, 2006. Citations on pages [58](#) e [59](#).

FARCHI, E.; NIR, Y.; UR, S. Concurrent bug patterns and how to test them. In: **Proceedings of the 17th International Symposium on Parallel and Distributed Processing**. [S.l.]: IEEE Computer Society, 2003. Citations on pages [36](#), [37](#) e [44](#).

FIEDOR, J.; HRUBÁ, V.; KŘENA, B.; LETKO, Z.; UR, S.; VOJNAR, T. Advances in noise-based testing of concurrent software. **Softw. Test. Verif. Reliab.**, John Wiley and Sons Ltd., v. 25, n. 3, p. 272–309, 2015. Citation on page [26](#).

FRASER, G.; ARCURI, A. Efficient mutation testing using whole test suite generation. **Empirical Software Engineering**, 2013. Citation on page [27](#).

FRASER, G.; ZELLER, A. Mutation-driven generation of unit tests and oracles. In: **Proceedings of the 19th International Symposium on Software Testing and Analysis**. [S.l.]: ACM, 2010. (ISSTA '10), p. 147–158. Citation on page [27](#).

_____. Mutation-driven generation of unit tests and oracles. **IEEE Trans. Softw. Eng.**, v. 38, n. 2, p. 278–292, 2012. Citation on page [27](#).

FREITAS, F. G.; MAIA, C. L. B.; CAMPOS, G. A. L.; SOUZA, J. T. Otimização em teste de software com aplicação de metaheurísticas. **Revista de Sistemas de Informação da FSMA**, p. 3–13, 2010. Citations on pages [11](#) e [54](#).

FREITAS, F. G.; MAIA, C. L. B.; COUTINHO, D. P.; CAMPOS, G. A. L.; SOUZA, J. T. Aplicação de metaheurísticas em problemas da engenharia de software: Revisão de literatura. In: **II Congresso Tecnológico InfoBrasil**. [S.l.: s.n.], 2009. (II CTInfoBrasil). Citation on page [52](#).

FRIESEN, J. **Java Threads and the Concurrency Utilities**. [S.l.]: Apress Press, 2015. 518 p. Citations on pages 38 e 39.

GAVRILOV, A. **Yet Another Expect for Java**. 2017. Available: <<https://github.com/Alexey1Gavrilov/ExpectIt>>. Citation on page 88.

_____. **Yet Another Expect for Java**. 2018. Available: <<https://github.com/Alexey1Gavrilov/ExpectIt>>. Citation on page 68.

GIACOMETTI, C.; SOUZA, S. R. S.; SOUZA, P. S. L. Teste de mutação para a validação de aplicações concorrentes usando PVM. In: **Revista Eletrônica de Iniciação científica**. [S.l.: s.n.], 2003. (REIC, 3). Citations on pages 43, 78 e 81.

GLIGORIC, M.; JAGANNATH, V.; MARINOV, D. Mutmut: Efficient exploration for mutation testing of multithreaded code. In: **Third International Conference on Software Testing, Verification and Validation**. [S.l.: s.n.], 2010. (ICST), p. 55–64. Citations on pages 73, 79, 81 e 99.

GLIGORIC, M.; ZHANG, L.; PEREIRA, C.; POKAM, G. Selective mutation testing for concurrent code. In: **Proceedings of the 2013 International Symposium on Software Testing and Analysis**. [S.l.: s.n.], 2013. (ISSTA), p. 224–234. Citations on pages 25, 43, 44, 63, 64, 73, 81, 86, 96, 98, 99, 100, 106, 110, 115 e 116.

_____. **Comutation: Selective Mutation Testing for Concurrent Code**. 2018. Available: <<http://mir.cs.illinois.edu/comutation/>>. Citation on page 98.

GLOVER, F. Future paths for integer programming and links to artificial intelligence. **Comput. Oper. Res.**, v. 13, n. 5, p. 533–549, 1986. Citation on page 51.

GLOVER, F.; KOCHENBERGER, G. **Handbook of Metaheuristics**. [S.l.]: Springer, 2003. Citation on page 51.

GODEFROID, P.; KHURSHID, S. Exploring very large state spaces using genetic algorithms. **Int. J. Softw. Tools Technol. Transf.**, Springer-Verlag, v. 6, n. 2, p. 117–127, 2004. Citations on pages 26 e 58.

GRAMA, A.; GUPTA, A.; KARYPIS, G.; KUMAR, V. **Introduction to parallel computing**. 2.. ed. [S.l.]: Benjamin-Cummings Publishing Co., Inc., 2003. Citations on pages 11, 30, 31, 32, 33 e 34.

HANSSON, H.; JONSSON, B. **A Logic for Reasoning About Time and Reability**. [S.l.], 1990. Citation on page 48.

HARMAN, M.; JIA, Y.; MATEO, P. R.; POLO, M. Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation. In: **29th ACM/IEEE International Conference on Automated Software Engineering**. [S.l.]: IEEE Computer Society, 2014. (ASE 2014), p. 397–408. Citation on page 117.

HARMAN, M.; JIA, Y.; ZHANG, Y. Achievements, open problems and challenges for search based software testing. In: **8th IEEE International Conference on Software Testing, Verification and Validation**. [S.l.]: IEEE Computer Society, 2015. (ICSTVV), p. 1–12. Citation on page 54.

HARMAN, M.; JONES, B. F. Search-based software engineering. **Information and Software Technology**, v. 43, p. 833–839, 2001. Citation on page 52.

HAUSEN, A. C. **ValiMPI : uma ferramenta de teste estrutural para programas paralelos em ambiente de passagem de mensagem**. [S.l.], 2005. Citation on page 80.

HERMADI, I. Genetic algorithm based test data generator. In: **Proceedings of the 2003 Congress on Evolutionary Computation**. [S.l.: s.n.], 2003. (CEC2003), p. 85–91. Citation on page 55.

HOLLAND, J. H. **Adaptation in natural and artificial systems**. [S.l.]: MIT Press, 1992. Citations on pages 51 e 52.

HOWDEN, W. E. Reliability of the path analysis testing strategy. **IEEE Transactions on Software Engineering**, SE-2, n. 3, p. 208–215, Sept 1976. Citation on page 35.

HRUBÁ, V.; KŘENA, B.; LETKO, Z.; UR, S.; VOJNAR, T. Testing of concurrent programs using genetic algorithms. In: **Search Based Software Engineering**. [S.l.]: Springer Berlin Heidelberg, 2012, (Lecture Notes in Computer Science, v. 7515). p. 152–167. Citation on page 58.

_____. **Testing of Concurrent Programs Using Genetic Algorithms**. [S.l.], 2012. Technical report FIT-TR-2012-01. Citation on page 60.

HRUBÁ, V.; KŘENA, B.; LETKO, Z.; PLUHÁČKOVÁ, H.; VOJNAR, T. Multi-objective genetic optimization for noise-based testing of concurrent software. In: **Search-Based Software Engineering**. [S.l.]: Springer International Publishing, 2014, (Lecture Notes in Computer Science, v. 8636). p. 107–122. Citations on pages 24, 58, 60 e 83.

JACOBSON, L.; KANBER, B. **Genetic Algorithms in Java Basics**. [S.l.]: Apress, 2015. 154 p. Citations on pages 86, 88, 89 e 90.

JAGANNATH, V.; GLIGORIC, M.; JIN, D.; LUO, Q.; ROSU, G.; MARINOV, D. Improved multithreaded unit testing. In: **Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering**. [S.l.]: ACM, 2011. (ESEC/FSE '11), p. 223–233. Citations on pages 73 e 99.

JANČÍK, P.; PARÍZEK, P.; KOFROŇ, J. Advanced debugging with JPF inspector. In: **Proceedings of the 2011 Java Pathfinder Workshop**. [S.l.: s.n.], 2011. (ASE 2011). Citations on pages 50 e 67.

JPF. **Java Path Finder**. 2017. Available: <<http://jpf.byu.edu/index.html>>. Citations on pages 50, 80, 85 e 97.

KHOR, S.; GROGONO, P. Using a genetic algorithm and formal concept analysis to generate branch coverage test data automatically. In: **19th IEEE International Conference on Automated Software Engineering**. [S.l.]: IEEE Computer Society, 2004. (ASE 2004), p. 346–349. Citation on page 55.

KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. Optimization by simulated annealing. **Science**, American Association for the Advancement of Science, v. 220, n. 4598, p. 671–680, 1983. Citations on pages 52 e 54.

- KIRKPATRICK, S.; GELATT JR., C. D.; VECCHI, M. P. Optimization by simulated annealing. In: ANDERSON, J. A.; ROSENFELD, E. (Ed.). **Neurocomputing: foundations of research**. [S.l.]: American Association for the Advancement of Science, 1988. p. 551–567. Citation on page 51.
- KOREL, B. Automated software test data generation. **IEEE Trans. Softw. Eng.**, v. 16, n. 8, p. 870–879, 1990. Citation on page 55.
- KOSCIANSKI, A.; SOARES, M. S. **Qualidade de software: aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software**. 2. ed. [S.l.]: Novatec Editora, 2007. Citation on page 40.
- KRAMMER, B.; BIDMON, K.; MÜLLER, M. S.; RESCH, M. M. MARMOT: An MPI analysis and checking tool. **Advances in Parallel Computing**, v. 13, p. 493–500, 2003. Citation on page 38.
- KRAWCZYK, H.; WISZNIEWSKI, B.; MORK, P. Classification of software defects in parallel programs. **HPCTI Progress Report 1**, 1995. Faculty of Eletronics, Technical University of Gdansk. Citation on page 35.
- KŘENA, B.; LETKO, Z.; VOJNAR, T.; UR, S. A platform for search-based testing of concurrent software. In: **Proceedings of the 8th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging**. [S.l.]: ACM, 2010. -, p. 48–58. Citations on pages 24, 58 e 59.
- KUSANO, M.; WANG, C. Cmutator: A mutation generator for concurrency constructs in multithreaded c/c++ applications. In: **28th IEEE/ACM International Conference on Automated Software Engineering**. [S.l.]: IEEE Computer Society, 2013. (ASE 2013), p. 722–725. Citations on pages 44, 79 e 81.
- LEI, J.; CARVER, R. H. A stateful approach to testing monitors in multithreaded programs. **9th IEEE International Symposium on High-Assurance Systems Engineering**, IEEE Computer Society, v. 00, p. 54–63, 2010. ISSN 1530-2059. Citations on pages 24, 44, 46 e 83.
- LEI, Y.; CARVER, R. H. Reachability testing of concurrent programs. **IEEE Trans. Softw. Eng.**, IEEE Press, v. 32, p. 382–403, 2006. Citation on page 41.
- LETKO, Z. Sophisticated testing of concurrent programs. In: **Second International Symposium on Search Based Software Engineering**. [S.l.: s.n.], 2010. (SSBSE), p. 36–39. Citation on page 60.
- _____. Analysis and testing of concurrent programs. In: **Information Sciences and Technologies Bulletin of the ACM Slovakia**. [S.l.]: ACM, 2013. p. 1–7. Citations on pages 58 e 60.
- LI, Y. **Remote Procedure Call (csc6320 survey paper)**. 2000. Available: <www.cs.gsu.edu/~cscyip/csc8320/rpcD/rpc.doc>. Citation on page 33.
- LI, Z.; HARMAN, M.; HIERONS, R. M. Search algorithms for regression test case prioritization. **IEEE Trans. Softw. Eng.**, v. 33, n. 4, p. 225–237, 2007. Citation on page 55.
- LONG, B.; DUKE, R.; GOLDSON, D.; STROOPER, P.; WILDMAN, L. Mutation-based exploration of a method for verifying concurrent Java components. In: **18th International Parallel and Distributed Processing Symposium**. [S.l.: s.n.], 2004. (PDPS). Citation on page 36.

LONG, B.; STROOPER, P. A classification of concurrency failures in Java components. In: **17th International Symposium on Parallel and Distributed Processing**. [S.l.]: IEEE Computer Society, 2003. Citations on pages 36 e 37.

LONG, B.; STROOPER, P.; WILDMAN, L. A method for verifying concurrent java components based on an analysis of concurrency failures. **Concurrency and Computation: Practice and Experience**, v. 19, n. 3, p. 281–294. Citation on page 64.

_____. A method for verifying concurrent Java components based on an analysis of concurrency failures: Research articles. **Concurr. Comput.: Pract. Exper.**, John Wiley and Sons Ltd., 2005. Citation on page 36.

LOURENÇO, J.; CUNHA, G. Testing patterns for software transactional memory engines. In: **Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging**. [S.l.]: ACM, 2007. p. 36–42. Citation on page 38.

LU, S.; PARK, S.; SEO, E.; ZHOU, Y. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: **Proceedings of the 13th international conference on Architectural support for programming languages and operating systems**. [S.l.]: ACM, 2008. p. 329–339. Citations on pages 37 e 38.

LUECKE, G. R.; CHEN, H.; COYLE, J.; HOEKSTRA, J.; KRAEVA, M.; ZOU, Y. MPI-CHECK: a tool for checking Fortran 90 MPI programs. **Concurrency and Computation: Practice and Experience**, v. 15, n. 2, p. 93–100, 2003. Citation on page 38.

MAIA, C. L. B.; CARMO, R. A. F. do; FREITAS, F. G. de; CAMPOS, G. A. L. de; SOUZA, J. T. de. A multi-objective approach for the regression test case selection problem. In: **Proceedings of Anais do XLI Simpósio Brasileiro de Pesquisa Operacional**. [S.l.: s.n.], 2009. (SBPO), p. 1824–1835. Citation on page 55.

_____. Automated test case prioritization with reactive GRASP. **Adv. Soft. Eng.**, p. 2:1–2:13, 2010. Citation on page 55.

MALBURG, J.; FRASER, G. Search-based testing using constraint-based mutation. **Software Testing, Verification and Reliability**, v. 24, n. 6, p. 472–495, 2014. Citation on page 27.

MALDONADO, J. C. **Crterios potenciais usos: Uma contribuio ao teste estrutural de software**. Phd Thesis (PhD Thesis) — Campinas University, 1991. Citation on page 40.

MANSOUR, N.; BAHSOON, R.; BARADHI, G. Empirical comparison of regression test selection algorithms. **Journal of Systems and Software**, v. 57, p. 79–90, 2001. Citation on page 55.

MATHUR, A. P. Performance, effectiveness, and reliability issues in software testing. In: **Proceedings of the Fifteenth Annual International Computer Software and Applications Conference**. [S.l.: s.n.], 1991. (ICSAC), p. 604–605. Citations on pages 77 e 81.

MCMILLAN, K. L. A technique of state space search based on unfolding. **Form. Methods Syst. Des.**, Kluwer Academic Publishers, v. 6, n. 1, p. 45–65, Jan. 1995. Citation on page 24.

MELO, S. M.; SOUZA, P. S. L.; SOUZA, S. R. S. Towards an empirical study design for concurrent software testing. In: **Proceedings of the Fourth International Workshop on Software Engineering for HPC in Computational Science and Engineering**. [S.l.]: IEEE Press, 2016. (SE-HPCCSE '16), p. 1–1. Citations on pages 96 e 110.

- MELO, S. M.; SOUZA, S. R. S.; SILVA, R. A.; SOUZA, P. Concurrent software testing in practice: A catalog of tools. In: **6th International Workshop on Automating Test Case Design, Selection and Evaluation**. [S.l.: s.n.], 2015. (A-Test), p. 31–40. Citations on pages [24](#) e [118](#).
- MICHAEL, C. C.; MCGRAW, G.; SCHATZ, M. A. Generating software test data by evolution. **IEEE Trans. Softw. Eng.**, v. 27, n. 12, p. 1085–1110, 2001. Citation on page [55](#).
- MILLER, W.; SPOONER, D. L. Automatic generation of floating-point test data. **IEEE Trans. Softw. Eng.**, v. 2, n. 3, p. 223–226, 1976. Citation on page [52](#).
- MUSUVATHI, M.; QADEER, S.; BALL, T.; BASLER, G.; NAINAR, P. A.; NEAMTIU, I. Finding and reproducing heisenbugs in concurrent programs. In: **Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation**. [S.l.]: USENIX Association, 2008. (OSDI'08), p. 267–280. Citation on page [106](#).
- MYERS, G. J.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. 3rd. ed. [S.l.]: Wiley Publishing, 2011. Citations on pages [24](#) e [41](#).
- OFFUTT, A. J.; LEE, A.; ROTHERMEL, G.; UNTCH, R. H.; ZAPF, C. An experimental determination of sufficient mutant operators. **ACM Trans. Softw. Eng. Methodol.**, v. 5, n. 2, p. 99–118, 1996. Citations on pages [77](#) e [81](#).
- OFFUTT, A. J.; PAN, J.; TEWARY, K.; ZHANG, T. An experimental evaluation of data flow and mutation testing. **Softw. Pract. Exper.**, v. 26, n. 2, p. 165–176, 1996. Citation on page [38](#).
- OFFUTT, A. J.; ROTHERMEL, G.; ZAPF, C. An experimental evaluation of selective mutation. In: **15th International Conference on Software Engineering**. [S.l.: s.n.], 1993. (ICSE), p. 100–107. Citations on pages [77](#) e [81](#).
- OFFUTT, A. J.; VOAS, J.; PAYNE, J. **Mutation Operators for Ada**. [S.l.], 1996. Citations on pages [25](#), [43](#), [44](#), [46](#), [77](#), [78](#) e [81](#).
- OLIVEIRA, A. A. L.; CAMILO-JUNIOR, C. G.; VINCENZI, A. M. R. A coevolutionary algorithm to automatic test case selection and mutant in mutation testing. In: **IEEE Congress on Evolutionary Computation**. [S.l.]: IEEE Computer Society, 2013. p. 829–836. Citations on pages [57](#) e [117](#).
- OMAR, E.; GHOSH, S.; WHITLEY, D. Constructing subtle higher order mutants for Java and AspectJ programs. In: **IEEE 24th International Symposium on Software Reliability Engineering**. [S.l.]: IEEE Computer Society, 2013. (ISSRE), p. 340–349. Citation on page [57](#).
- PARSAI, A. **Literature Survey on Mutation Testing**. [S.l.], 2014. Citations on pages [24](#) e [43](#).
- PEDERSEN, J. B. Classification of programming errors in parallel message passing systems. In: **Communicating Process Architectures**. [S.l.: s.n.], 2006. (CPA), p. 363–376. Citation on page [38](#).
- PELED, D. Combining partial order reductions with on-the-fly model-checking. In: **Proceedings of the 6th International Conference on Computer Aided Verification**. [S.l.]: Springer-Verlag, 1994. (CAV '94), p. 377–390. Citation on page [48](#).
- POSKITT, C.; POULDING, S. Using contracts to guide the search-based verification of concurrent programs. In: **Search Based Software Engineering**. [S.l.]: Springer Berlin Heidelberg, 2013, (Lecture Notes in Computer Science, v. 8084). p. 263–268. Citations on pages [58](#) e [60](#).

QUEILLE, J.-P.; SIFAKIS, J. Specification and verification of concurrent systems in cesar. In: **Proceedings of the 5th Colloquium on International Symposium on Programming**. [S.l.]: Springer-Verlag, 1982. p. 337–351. Citation on page 48.

RAPPS, S.; WEYUKER, E. J. Selecting software test data using data flow information. **IEEE Trans. Softw. Eng.**, IEEE Computer Society, v. 11, p. 367–375, 1985. Citation on page 41.

ROBSON, C. **Real world research**. 2. ed. [S.l.]: Blackwell, 2002. Citation on page 64.

ROJAS, J. M.; FRASER, G.; ARCURI, A. Seeding strategies in search-based unit test generation. **Software Testing, Verification and Reliability**, v. 26, n. 5, p. 366–401, 2016. Citation on page 27.

ROSCOE, A. W.; HOARE, C. A. R.; BIRD, R. **The Theory and Practice of Concurrency**. [S.l.]: Prentice Hall PTR, 1997. ISBN 0136744095. Citation on page 30.

ROTHERMEL, G.; HARROLD, M. J. Empirical studies of a safe regression test selection technique. **IEEE Trans. Softw. Eng.**, p. 401–419, 1998. Citation on page 55.

ROTHERMEL, G.; UNTCH, R. H.; CHU, C.; HARROLD, M. J. Prioritizing test cases for regression testing. **IEEE Transactions on Software Engineering**, v. 7, p. 929–948, 2001. Citation on page 55.

RUNGTA, N.; MERCER, E. G. A meta heuristic for effectively detecting concurrency errors. In: **Proceedings of Hardware and Software: Verification and Testing: 4th International Haifa Verification Conference**. [S.l.]: Springer Berlin Heidelberg, 2008. p. 23–37. Citation on page 59.

SAMOFALOV, V.; KRUKOV, V.; KUHN, B.; ZHELTOV, S.; KONOVALOV, A.; DESOUSA, J. Parallel computing: Current and future issues of high-end computing. **Proceedings of the International Conference**, 2005. Citation on page 38.

SARMANHO, F. S.; SOUZA, P. S.; SOUZA, S. R.; SIMÃO, A. S. Structural testing for semaphore-based multithread programs. In: **8th International Conference on Computational Science, Part I**. [S.l.: s.n.], 2008. (ICCS), p. 337–346. Citations on pages 42 e 80.

SCHWARZ, B.; SCHULER, D.; ZELLER, A. Breeding high-impact mutations. In: **IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops**. [S.l.]: IEEE Computer Society, 2011. (ICSTW), p. 382–387. Citation on page 57.

SCOTT, L. R.; CLARK, T.; BAGHERI, B. **Scientific Parallel Computing**. [S.l.]: Princeton University Press, 2005. ISBN 069111935X. Citations on pages 33 e 38.

SEN, A. Mutation operators for concurrent SystemC designs. In: **Proceedings of the 2009 10th International Workshop on Microprocessor Test and Verification**. [S.l.]: IEEE Computer Society, 2009. p. 27–31. Citations on pages 38 e 43.

SEN, K. Effective random testing of concurrent programs. In: **Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering**. [S.l.]: ACM, 2007. (ASE '07), p. 323–332. Citation on page 24.

_____. Race directed random testing of concurrent programs. **SIGPLAN Not.**, ACM, v. 43, n. 6, p. 11–21, Jun. 2008. Citation on page 24.

SHOUSHI, M.; BRIAND, L.; LABICHE, Y. A uml/marte model analysis method for uncovering scenarios leading to starvation and deadlocks in concurrent systems. **IEEE Transactions on Software Engineering**, v. 38, n. 2, p. 354–374, March 2012. Citation on page 24.

SILVA-BARRADAS, S. **Mutation analysis of concurrent software**. Phd Thesis (PhD Dissertation) — Doutorado de Pesquisa em Engenharia Informática e Automática, Politécnico de Milano, 1998. Citations on pages 24, 25, 43, 44, 46, 78, 80, 81 e 83.

SILVA, J. **Automatic Test Data Generation with Meta-heuristics for Concurrent Programs**. Phd Thesis (master's dissertation) — São Paulo University, 2014. In Portuguese. Citations on pages 58 e 60.

SILVA, R. A. **Mutation Testing applied to concurrent programs in MPI**. Master's Thesis (MSc Thesis) — ICMC, Instituto de Computação e Matemática Computacional, USP, 2013. In Portuguese. Citations on pages 25, 40, 44 e 47.

SILVA, R. A.; SOUZA, S. R. S.; SOUZA, P. S. L. Deterministic execution of concurrent programs during the mutation testing. In: **6th Brazilian Workshop on Systematic and Automated Software Testing**. [S.l.: s.n.], 2012. (SAST). In Portuguese. Citations on pages 25, 44 e 47.

_____. Mutation testing for concurrent programs in MPI. In: **13th Latin American Test Workshop**. [S.l.: s.n.], 2012. (LATW), p. 69–74. Citations on pages 24, 43, 79, 81 e 83.

_____. Search-based mutation testing for concurrent programs. In: **IV WORKSHOP DE TESES E DISSETAÇÕES DO CBSOFT**. [S.l.: s.n.], 2014. (WTDSOFT, v. 2), p. 109–116. In Portuguese. Citation on page 118.

_____. A systematic review on search based mutation testing. **Information and Software Technology**, v. 81, p. 19 – 35, 2017. Citations on pages 11, 26, 27, 55, 78, 114 e 118.

SIR. **Software-artifact Infrastructure Repository**. 2018. Available: <<http://sir.unl.edu>>. Citation on page 115.

SISTLA, A. P.; CLARKE, E. M. The complexity of propositional linear temporal logics. **J. ACM**, ACM, v. 32, n. 3, p. 733–749, Jul. 1985. Citation on page 49.

SOTTILE, M. J.; MATTSON, T. G.; RASMUSSEN, C. E. **Introduction to concurrency in programming languages**. [S.l.]: Boca Raton: Chapman & Hall/CRC Press, 2010. Citation on page 34.

SOUZA, P. S.; SOUZA, S. R.; ZALUSKA, E. Structural testing for message-passing concurrent programs: an extended test model. **Concurrency and Computation: Practice and Experience**, v. 26, n. 1, p. 21–50, 2014. Citations on pages 24, 42 e 83.

SOUZA, P. S.; SOUZA, S. S.; ROCHA, M. G.; PRADO, R. R.; BATISTA, R. N. Data flow testing in concurrent programs with message passing and shared memory paradigms. **Procedia Computer Science**, v. 18, p. 149 – 158, 2013. 2013 International Conference on Computational Science. Citations on pages 42 e 117.

SOUZA, S. R. S.; SOUZA, P. S. L.; BRITO, M. A. S.; SIMAO, A. S.; ZALUSKA, E. J. Empirical evaluation of a new composite approach to the coverage criteria and reachability testing of concurrent programs. **Softw. Test. Verif. Reliab.**, John Wiley and Sons Ltd., v. 25, n. 3, p. 310–332, 2015. Citation on page 47.

SOUZA, S. R. S.; SOUZA, P. S. L.; MELO, S. M.; SILVA, R. A.; VERGILIO, S. R. Teste de programas concorrentes. In: DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. (Ed.). **Introdução ao Teste de Software**. [S.l.]: Elsevier Editora Ltda, 2016. v. 2, p. –. Citations on pages [37](#) e [118](#).

SOUZA, S. R. S.; VERGILIO, S. R.; SOUZA, P. S. L.; SIMÃO, A. S.; HAUSEN, A. C. Structural testing criteria for message-passing parallel programs. **Concurr. Comput.: Pract. Exper.**, John Wiley and Sons Ltd., v. 20, p. 1893–1916, 2008. Citations on pages [11](#), [41](#) e [42](#).

STAUNTON, J.; CLARK, J. A. Searching for safety violations using estimation of distribution algorithms. In: **Third International Conference on Software Testing, Verification, and Validation Workshops**. [S.l.: s.n.], 2010. (ICSTW), p. 212–221. Citations on pages [58](#) e [59](#).

STEENBUCK, S.; FRASER, G. Generating unit tests for concurrent classes. In: **IEEE Sixth International Conference on Software Testing, Verification and Validation**. [S.l.]: IEEE Computer Society, 2013. (ICST), p. 144–153. Citations on pages [58](#) e [60](#).

TAI, K.; CARVER, R.; OBAID, E. Deterministic execution debugging of concurrent Ada programs. In: **Proceedings of the Computer Software and Applications Conference**. [S.l.: s.n.], 1989. (CSAC), p. 102–109. Citation on page [44](#).

TAI, K. C. On testing concurrent programs. In: **The IEEE Computer Software & Applications Conference**. [S.l.]: IEEE Computer Society, 1985. (COMPSAC), p. 510–517. Citation on page [44](#).

TAI, K.-C. An approach to testing concurrent Ada programs. In: **Proceedings of the third annual Washington Ada symposium on Ada: Ada use in focus : practical lessons in perspective**. [S.l.]: ACM, 1986. p. 253–264. Citation on page [44](#).

TALBI, E.-G. **Metaheuristics: From Design to Implementation**. [S.l.]: Wiley Publishing, 2009. Citation on page [51](#).

TANENBAUM, A. S. **Organização Estruturada de Computadores**. 5.. ed. [S.l.]: Prentice Hall, 2007. Citations on pages [30](#) e [33](#).

TAYLOR, R.; LEVINE, D.; KELLY, C. Structural testing of concurrent programs. **Software Engineering, IEEE Transactions on**, v. 18, n. 3, p. 206–215, March 1992. ISSN 0098-5589. Citations on pages [24](#), [40](#), [41](#) e [83](#).

TRACEY, N.; CLARK, J.; MANDER, K.; MCDERMID, J. An automated framework for structural test-data generation. In: **13th IEEE international Conference on Automated Software Engineering**. [S.l.]: IEEE Computer Society, 1998. (ICASE). Citation on page [55](#).

VASENIN, V. A.; KRIVCHIKOV, M. A. A model of dynamical concurrent program execution. **Program. Comput. Softw.**, Plenum Press, v. 39, n. 1, p. 1–9, Jan. 2013. Citation on page [24](#).

VERGILIO, S. R.; SOUZA, S. R. S.; SOUZA, P. S. L. Coverage testing criteria for message-passing parallel programs. In: **6th IEEE Latin American Test Workshop**. [S.l.]: IEEE Computer Society, 2005. (LATW), p. 161–166. Citation on page [41](#).

VETTER, J. S.; SUPINSKI, B. R. Dynamic software testing of mpi applications with umpire. In: **Proceedings of the 2000 ACM/IEEE conference on Supercomputing**. [S.l.]: IEEE Computer Society, 2000. Citation on page [38](#).

- VINCENZI, A. M. R.; SIMÃO A. S. NAND DELAMARO, M. E.; MALDONADO, J. C. Muta-pro: Towards the definition of a mutation testing process. **Journal of the Brazilian Computer Society**, v. 12, n. 2, p. 49–61, 2006. Citation on page [77](#).
- WALCOTT, K. R.; SOFFA, M. L.; KAPFHAMMER, G. M.; ROOS, R. S. Timeaware test suite prioritization. In: **International Symposium on Software Testing and Analysis**. [S.l.: s.n.], 2006. (ISSTA), p. 1–12. Citation on page [55](#).
- WEGENER, J.; GROCHTMANN, M. Verifying timing constraints of real-time systems by means of evolutionary testing. **Real-Time Syst.**, v. 15, p. 275–298, 1998. Citation on page [55](#).
- WEGENER, J.; STHAMER, H.; JONES, B. F.; EYRES, D. E. Testing real-time systems using genetic algorithms. **Software Quality Control**, v. 6, p. 127–135, 1997. Citation on page [55](#).
- WEISS, S. N. A formal framework for the study of concurrent program testing. In: **Second Workshop on Software Testing, Verification, and Analysis**. [S.l.: s.n.], 1988. (STVA), p. 106–113. Citation on page [24](#).
- WILKINSON, B.; ALLEN, M. **Parallel programming: techniques and applications using networked workstations and parallel computers**. [S.l.]: Prentice-Hall, Inc., 1999. Citation on page [30](#).
- WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in software engineering: an introduction**. [S.l.]: Kluwer Academic Publishers, 2000. Citations on pages [97](#), [103](#), [109](#) e [110](#).
- WOHLIN, C.; RUNESON, P.; HOST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLEN, A. **Experimentation in Software Engineering**. 1. ed. [S.l.]: Springer-Verlag Berlin Heidelberg, 2012. Citation on page [73](#).
- WU, G.; KAISER, L. Constructing subtle concurrency bugs using synchronization-centric second-order mutation operators. In: **SEKE**. [S.l.]: Knowledge Systems Institute Graduate School, 2011. p. 244–249. Citation on page [37](#).
- YANG, C.-S. D.; SOUTER, A. L.; POLLOCK, L. L. All-du-path coverage for parallel programs. In: **Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis**. [S.l.]: ACM, 1998. p. 153–162. ISBN 0-89791-971-8. Citations on pages [24](#), [40](#), [41](#) e [83](#).
- YANG, L. **MODEL CHECKING CONCURRENT AND REAL-TIME SYSTEMS: THE PAT APPROACH**. Phd Thesis (PhD Thesis) — DEPARTMENT OF COMPUTER SCIENCE NATIONAL UNIVERSITY OF SINGAPORE, 2009. Citation on page [48](#).
- YOO, S.; HARMAN, M. Pareto efficient multi-objective test case selection. In: **Proceedings of the 2007 International Symposium on Software Testing and Analysis**. [S.l.: s.n.], 2007. (ISSTA), p. 140–150. Citation on page [55](#).
- YUE, J.; HARMAN, M. Constructing subtle faults using higher order mutation testing. In: **Eighth IEEE International Working Conference on Source Code Analysis and Manipulation**. [S.l.]: IEEE Computer Society, 2008. (SCAM), p. 249–258. Citation on page [56](#).
- _____. An analysis and survey of the development of mutation testing. **Software Engineering, IEEE Transactions on**, v. 37, n. 5, p. 649–678, 2011. Citations on pages [24](#) e [42](#).