Anais da 2ª Escola Regional de Engenharia de Software (ERES 2018)
em Dois Vizinhos, PR, Brasil, 22 a 24 de outubro de 2018

Achievements, Challenges and Opportunities on Mutation Testing of Concurrent Programs

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

[1]Software Engineering Course Coordination – Federal University of Technology - PR
85660-000 – Dois Vizinhos – PR – Brazil

[2]Institute of Mathematics and Computer Sciences – University of São Paulo
13566-590 – São Carlos – SP – Brazil

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

Abstract. *With the increasing advances in the hardware technology and the massive presence of multicore processors in personal computers, concurrent programming has been becoming more popular. Therefore, new challenges have been emerged due to the communication and synchronization aspects inherent in concurrent programs. The testing activity in this scenario is considered essential for the delivery of reliable programs. Mutation testing is a fault-based testing criterion that uses mistakes done by developers during the software development. The application of this criterion in concurrent programs is challenging due to the non-determinism and concurrency problems, such as starvation and deadlock. Studies have been conducted for the application of the mutation test in concurrent programs, however, some aspects still need attention and improvements, leading researchers to face a collection of challenges. This paper presents a roadmap of this field identifying the achievements, challenges, and opportunities to study in the context of mutation testing of concurrent programs. As a result, researchers may find a guide for the proposal of new researches in the field.*

1. Introduction

Concurrent programming paradigm has become more popular nowadays. It happens because the hardware supports parallel programming, increasing the performance of concurrent programs. 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 Neumman's architecture is used that is composed of 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.

Mutation testing is a testing criterion that uses information about typical errors that can be made in the software development process to derive test cases. Hence, the typical errors in a domain or paradigm of development are characterized and implemented as mutation operators that during the test activity, generate modified versions (mutants) of the product under test. The objective of the mutation test is to create a test set that can identify the behavior difference between the original program and the mutated program that has undergone a minor modification. When this difference in behavior is found, then it can be said that the mutant is dead (killed). Consider a program P that runs with a set of T test cases. The creation of mutants is based on a model of faults F. Each fault f present in F it is introduced in P one by one producing a set of mutants M. Each element present in F is a mutation operator. When M is executed with T and has a behavior other than P, mutant M is killed by the test case t. The goal is to kill all M with at least one t. If a mutant cannot be killed, the tester needs to show that M is equivalent to P or to add new t that kills M. If in the attempt to kill a mutant, a new t is generated and when executed in P, makes it to fail, the program need to be corrected and the test must reinitialize.

In the context of concurrent programs, apply software testing becomes a challenge because new issues must be explored such as nondeterminism and race conditions. When a test input is executed in a sequential program, only one correct output is given. The statements in a sequential program are always executed in the same order and do not change from one execution to another. This is false for concurrent programs, in which different executions of the same input may exercise different statements and may result in different outputs depending on the order in which the processes were executed. This can be a problem in mutation testing. When a mutant M is executed by a test case t, it will be killed if it exhibits an incorrect output according to a given specification. In the case of concurrent programs, an alternative would be the comparison of the set of all possible results of M with t with all possible results of P with t and then kill M if any output is different.

This paper addresses a roadmap, defining the achievements, challenges, and opportunities for mutation testing of concurrent programs. The remainder of the paper is organized as follows: Section 2 presents an overview of the roadmap. Section 3 provides the achievements in mutation testing of concurrent programs; Section 4 addresses the challenges identified in the mutation testing procedure in concurrent programs; Section 5 describes some opportunities in the mutation testing of concurrent programs; the conclusions are summarized in Section 6.

2. Mutation testing research roadmap

A roadmap provides directions to reach the desired destination starting from the body of knowledge of the field and going until an ideal scenario. The mutation testing for concurrent programs is organized as follows:

- Achievements: The first section constitutes in the theoretical background of mutation testing of concurrent programs. The main research in the area is the definition of mutation operators. Each programming language, library, and API has its own set of mutation operators. Another point is how the mutants will be executed and evaluated. This characterizes the definition of testing strategies that basically uses deterministic and non-deterministic approaches. Finally, some tools have been defined to support the testing application.
- Challenges: In this section, we discuss the main problems faced in the definition of new techniques, approaches, and tools in the context of mutation testing of concurrent programs. Example of challenges is the program selection, mutant generation, and the definition of deterministic or non-deterministic execution. These challenges must guide researches to propose solutions to these problems.
- Opportunities: The last section presents the opportunities in the research field, with examples of researches that must be addressed to improve the application of the mutation test to concurrent programs.

Fig. 1 illustrates the roadmap. The main activities of the mutation testing process are at the top of the figure, namely initial execution, mutant generation, mutant execution, and equivalent mutant identification. In the horizontal, there are the roadmap sections, namely achievements, challenges, and opportunities. In the white boxes, there are the aspects identified.

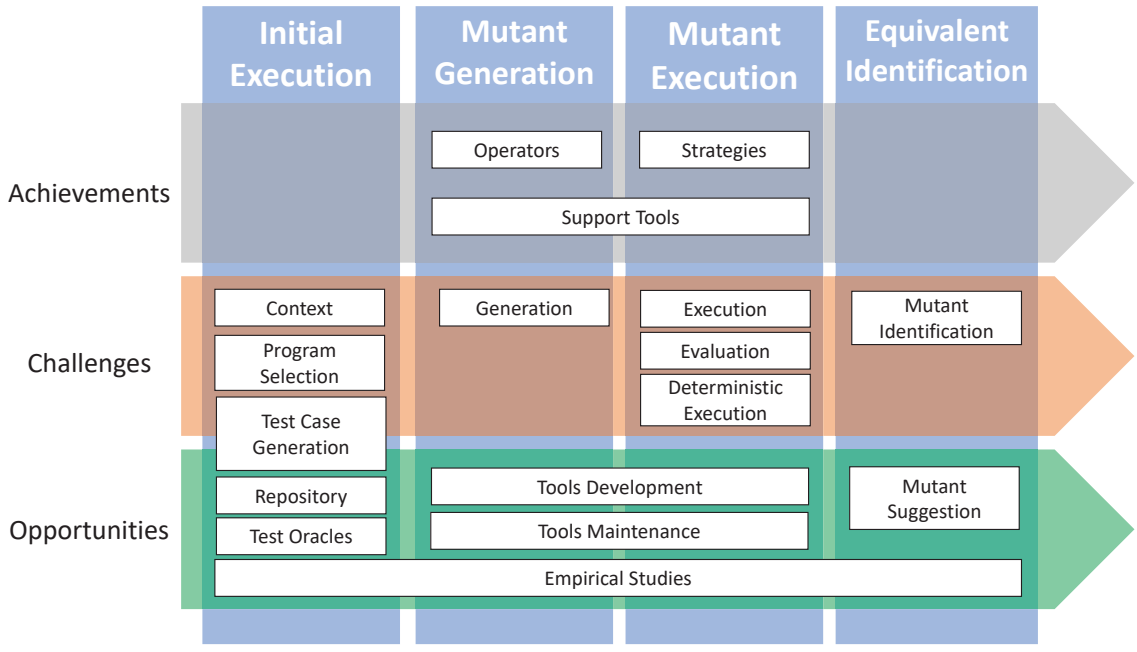

Figure 1. Mutation testing for concurrent programs roadmap.

3. Achievements

Besides the knowloge aquired during the years in the researching of mutation testing of concurrent programs, this paper is based on some empiral studies defined in the literature. Souza et al. (2011) presented a systematic review related to concurrent testing approaches, bug classification and testing tools. Melo et al. (2015) developed a catalog containing 116 tools for testing concurrent programs. **Mutation operators.** In the literature, several approaches have been defined for the definition of mutation operators for concurrent programs. Offutt et al. (1996) propose the definition of mutation operators for Ada programs. Considering the operators defined in Offutt et al. (1996), Silva-Barradas (1998) presents the application of the mutation test criterion for concurrent programs in Ada. The author classifies the mutation operators into Mutation operators for declarations and mutation operators for conditional expressions. M. 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 (generating a small number of mutants). Ghosh (2002) presents a proposal of mutation operators for concurrent object-oriented programs in Java. The focus of the approach is focused on cases where access to shared data needs to be protected, but it is not. The failure model assumes that programmers may not use the *synchronized* construct when needed.

Giacometti et al. (2003) present an initial proposal for the application of the mutation test for programs in message passing environment, considering the PVM (Parallel Virtual Machine) environment. Unlike the other works, this one presents the definition of mutation operators considering a predefined concurrent programming pattern, while the others define mutation operators only for some functions that allow concurrent programming in such languages. 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. Jagannath et al. (2010) define mutation operators for programs developed using the actor-oriented programming model. A program developed with this approach consists of a set of concurrent objects that communicate through message exchange.

Wu and Kaiser (2011) present an approach where second-order mutation operators are applied which generate competition errors not represented by first-order operators. The authors present 6 second-order mutation operators for Java. Silva et al. (2012b) defined 26 mutation operators for concurrent programs in MPI. The operators are classified into 3 categories organized according to the function-target where the mutation operators are applied. The “Collective” category presents mutation operators that are applied in collective-communication functions, the “Point-to-point” category presents mutation operators that are applied in point-to-point communication functions subdivided in operators applied on send, receive or others point-to-point functions and the “All” category presents operators that can be applied in both collective and point-to-point functions. 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.

Testing strategies. To deal with the non-determinism in the testing of concurrent programs, some strategies were defined. The MET (**Multiple Execution Testing**) ap-

proach 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 (**Deterministic Execution Testing**) approach presented in 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. Carver (1993) defined the **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 programs.

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. 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 DEMT, 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. M. E. 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.

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. Silva (2013) and Silva et al. (2012a) developed two approaches to support mutation testing. the **DEMT Adapted** approach runs the mutant deterministically, and the output is compared with the set of all possible outputs generated by reachability testing. The mutant is killed if it presents a different output or could not follow the synchronization sequence. The **MET Adapted** 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. 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 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.

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.

Support tools. For mutation testing, a few tools have been developed to support the mutation testing of concurrent programs. **Javalanche** (Schuler & Zeller, 2009) is an open source framework for mutation testing that applies a subset of method-level mutation operators. A characteristic of this tool is the possibility of ranking mutations by their impact on the behavior of program functions. Javalanche implements a subset of method-level mutant operators composed of 7 operators. **MutMut** (Gligoric, Jagannath, & Marinov, 2010) proposes an approach for an efficient execution of mutants in multithreaded programs. It uses a technique for the selection of mutants to be executed. When the original program is executed, the technique selects points in the code for mutation considering relevant aspects of the concurrent programs. The approach also enables the tester to select a *thread* to be executed, forcing the mutation introduced to be executed. **ConMan** (Bradbury et al., 2006) implements a set of mutation operators for concurrent programs in Java (J2SE 5.0). The mutation operators are classified into operators that modify critical regions, keywords, and calls for concurrent methods and operators that replace concurrent objects.

CCmutator (Kusano & Wang, 2013) implements those operators as well as new specific mutation operators for concurrent programs in *PThreads*. It utilizes the High Order Mutation technique, in which two or more mutations are inserted in the program for the creation of strong mutants and improvements in the quality of the testing case set. **Comutation** (Gligoric et al., 2013) uses selective mutation based on the mutation operators for concurrent Java programs. Selective mutation selects a subset of mutation operators in which test cases that have a high mutation score for this subset also feature for the other operators. The objective is to reduce the mutation testing cost. **ValiMut** (Silva, 2013) tool supports the use of mutation testing in competing programs developed in MPI. The tool allows the application of mutation operators defined for MPI in Silva et al. (2012b). This tool uses some modules of the ValiMPI tool (Hausen, Vergílio, Souza, Souza, & Simão, 2007) and applies the adapted MET approach (previously described) for the deterministic execution of the mutants. **BeMutation** (Behavior Mutation) (Silva, 2018) is a tool to support the application of mutation testing in Java multithread programs. BeMutation has implemented 27 mutation operators defined in Bradbury et al. (2006) and extended in Gligoric et al. (2013). The tool uses JPF-Inspector tool to generate synchronizations sequences.

4. Challenges

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. The challenge related to the context is the definition of a procedure to apply mutation testing for a specific programming language or libraries. For each language, a new procedure must be defined, considering the fault model and the mutation operators that are strictly related to the nuances of the programming language syntax.

Program selection. In the attempt to develop new supporting tools, researchers and developers must have a set of programs to evaluate characteristics of the tool, such as scalability and completeness related to the target programming language. Therefore, a challenge in this scenario is to find a set of programs classified using some software metrics as size (or Lines of Code), complexity, toy or real program, etc. The Software-artifact Infrastructure Repository (SIR, 2018) is a repository in which a set of programs can be found, however, when it comes to concurrent programs, the original version of the program is 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.

Mutant generation. The mutant generation is another issue in the testing of concurrent programs. Even with mutation operators defined for a given programming language, for some programming languages (e.g. PVM) there is not a tool to support the generation of the mutants. It is necessary a tool to support the generation of the mutants to guarantee the application of the mutation testing criterion. **Mutant execution.** As happens in the generation, it is crucial the use of a tool for the supporting of the mutant execution. In the execution task of mutation testing of concurrent programs, it is necessary to identify the synchronization sequence executed in the execution, especially when the Deterministic Testing approach is used. **Mutant evaluation.** After the execution of the mutant, it is necessary to evaluate whether output of the mutant is valid or not. It is necessary to evaluate the output, since different executions of a concurrent program with a single input may generate different and correct outputs. If the mutant presents an output that is different from the execution of the original program, it is necessary to evaluate if the different output was generate by the mutation or a different synchronization sequence. If the output would never be presented by the original program, the mutant is considered as dead. Therefore, to evaluate this, it is mandatory to know all possible outputs of the original program. This is impractical due to the size and complexity of concurrent programs. An alternative is to use the deterministic execution and use a synchronization sequence as support for this evaluation. The challenge is that there is no tool to support the evaluation of mutants. Some testing strategies were defined, but empirical evaluations are necessary to compare and identify the benefits of each one.

Test case generation. In the testing of programs, it is necessary to test different inputs and evaluate the outputs. In the testing of concurrent programs, besides the input, it is necessary to select synchronization sequences to execute and evaluate. A challenge in this context is the use of tools and/or techniques to support this activity. **Equivalent mutants identification.** The identification of equivalent is an indecisive problem for which there is no general purpose algorithm that support it. However, ignore them is not a good practice because it may lead to inaccurate mutation scores. The solution would be the generation of a good set of test cases that would kill almost all mutants so that the effort to analyze equivalent mutants would be low.

Deterministic vs. non-deterministic execution. 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*). 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. The Java PathFinder model checking tool is, so far, the most useful tool that saves the execution trace and allows the deterministic execution. 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. 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.

5. Opportunities

Catalog of programming languages. An important aspect to be considered is the creation of a catalog of programming languages and all the techniques, support tools and scientific papers available in the literature for each concurrent programming language, library, and API. **Benchmark and repository of programs.** Along with the catalog of the programming languages, an opportunity arises from the generation and categorization of concurrent programs in different programming languages. Those programs must be categorized using software metrics as size (or Lines Of Code), complexity, toy or real program, etc. This may help researchers and developers to evaluate supporting tools and be favorable in the conduction of empirical studies the field of mutation testing for concurrent programs. A research opportunity would be to catalog the benchmarks among the results with different test cases used, to reduce execution time and facilitate the comparison of the mutant score. **Support tools.** Supporting tools are required for an efficient application of the software testing activity. However, some tools are made just as a proof of concept of proposed approaches and sometimes are not available for download. Others need maintenance and adaption to work as it should be. An opportunity is to contact authors and ask for tools and available documentation to evolve and maintain those tools. Besides that, new tools can be developed to support the testing activity in environments where no tool is available.

Testing oracles. the results of test executions is a time-consuming and at times error-prone activity. In the context of concurrent programs, the problems with manual verification are exponential. A single test input in a concurrent program can generate different and correct outputs due to the different synchronization sequences. Depending on the testing strategy, a given input must be executed several times in an attempt to exercise different synchronization sequences, therefore generating different outputs. In addition, the inherent complexity of concurrent programs generally means that significantly larger test sets are required to achieve adequate test coverage, thus further increasing the cost of manual verification. An alternative to manual verification is to develop a test oracle that automatically verifies the results of test executions. However, the cost of developing an oracle by hand can approach that of implementing the system itself (Hunter & Strooper, 2001). The opportunity in this context is the creation of test oracles to support the application of mutation testing of concurrent programs.

Equivalent mutants suggestion. The identification of equivalent mutants is, so far, a manual activity. Therefore, an opportunity in this context is the use of heuristics to identify equivalent mutants and present the differences (looking for aspects of reachability, infection, and propagation of the execution) of the mutant in comparison with the original program. **Empirical studies.** Experimental studies in Software Engineering are important for the evaluation and evolution of existing techniques and tools. Through eval-

uation, you can identify gaps in approaches and limitations in tools. One opportunity is in planning and conducting experimental studies to verify the effectiveness in finding interesting synchronization sequences and evaluation of deterministic and non-deterministic execution of mutants.

6. Conclusion

The application of mutation testing for concurrent programs is a field in software engineering that has been received more attention. Therefore, several testing strategies and tools have been developed. The testing of concurrent programs has its challenges and the mutation testing has its own that must be investigated. In this paper, a roadmap in the field of mutation testing for concurrent programs is presented. It is known that covering into one article all ongoing and foreseen research directions is impossible, therefore, the main challenges and opportunities were presented. The contribution of this paper relies on the guide to new researches about trends and problems in the application of mutation testing of concurrent programs.

References

Almasi, G. S., & Gottlieb, A. (1989). *Highly parallel computing*. Benjamin-Cummings Publishing Co., Inc.

Andrews, G. (2001). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley.

Bradbury, J. S., Cordy, J. R., & Dingel, J. (2006). Mutation operators for concurrent Java (J2SE 5.0). In *Proceedings of the second workshop on mutation analysis* (pp. 11–20). IEEE.

Carver, R. (1993). Mutation-based testing of concurrent programs. In *Proceedings of test conference* (pp. 845–853).

Delamaro, M., Maldonado, J., Pezzè, M., & Vincenzi, A. (2001). Mutant operators for testing concurrent Java programs. In *Xv simpósio brasileiro de es.*

Delamaro, M. E. (2004). Using instrumentation to reproduce the execution of Java concurrent programs. In *Simpósio brasileiro de qualidade de software.*

Farchi, E., Nir, Y., & Ur, S. (2003). Concurrent bug patterns and how to test them. In *Proceedings of the 17th international symposium on parallel and distributed processing.* IEEE.

Ghosh, S. (2002). Towards measurement of testability of concurrent object-oriented programs using fault insertion: a preliminary investigation. In *Second ieee international workshop on source code analysis and manipulation* (pp. 17–25).

Giacometti, C., Souza, S. R. S., & Souza, P. S. L. (2003). Teste de mutação para a validação de aplicações concorrentes usando PVM. In *Revista eletrônica de iniciação científica* (Vol. 2).

Gligoric, M., Jagannath, V., & Marinov, D. (2010). Mutmut: Efficient exploration for mutation testing of multithreaded code. In *Proceedings of the 2010 third international conference on software testing, verification and validation* (pp. 55–64). Washington, DC, USA: IEEE.

Gligoric, M., Zhang, L., Pereira, C., & Pokam, G. (2013). Selective mutation testing for concurrent code. In *Proceedings of the 2013 international symposium on software testing and analysis* (pp. 224–234).

Hausen, A. C., Vergílio, S. R., Souza, S. R. S., Souza, P. S. L., & Simão, A. S. (2007). A tool for structural testing of MPI programs. In *Ieee latin-american testworkshop - latw.* IEEE.

Hunter, C., & Strooper, P. (2001). Systematically deriving partial oracles for testing concurrent programs. In *Proceedings 24th australian computer science conference. acsc 2001* (p. 83-91).

Jagannath, V., Gligoric, M., Lauterburg, S., Marinov, D., & Agha, G. (2010). Mutation operators for actor systems. In *Proceedings of the 2010 third international conference on software testing, verification, and validation workshops* (pp. 157–162).

Kusano, M., & Wang, C. (2013). Ccmutator: A mutation generator for concurrency constructs in multithreaded c/c++ applications. In *Ase* (p. 722-725).

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

Melo, S. M., Souza, S. R. S., Silva, R. A., & Souza, P. (2015). Concurrent software testing in practice: A catalog of tools. In *6th international workshop on automating test case design, selection and evaluation* (pp. 31–40).

Offutt, A. J., Voas, J., & Payne, J. (1996). *Mutation operators for Ada* (Tech. Rep.). George Mason University.

Schuler, D., & Zeller, A. (2009). Javalanche: Efficient mutation testing for java. In *Esec/fse '09* (pp. 297–298). New York, NY, USA: ACM.

Silva, R. A. (2013). *Mutation testing applied to concurrent programs in MPI* (MSc Thesis). ICMC, Instituto de Computação e Matemática Computacional, USP.

Silva, R. A. (2018). *Search based software testing for the generation of synchronization sequences for mutation testing of concurrent programs* (PhD Thesis). Institute of Mathematics and Computer Sciences University of São Paulo.

Silva, R. A., Souza, S. R. S., & Souza, P. S. L. (2012a). Deterministic execution of concurrent programs during the mutation testing. In *6th brazilian workshop on systematic and automated software testing.*

Silva, R. A., Souza, S. R. S., & Souza, P. S. L. (2012b). Mutation testing for concurrent programs in MPI. In *13th latin american test workshop* (pp. 69–74).

Silva-Barradas, S. (1998). *Mutation analysis of concurrent software* (PhD Dissertation). Dottorato di Ricerca in Ingegneria Informatica e Automatica, Poli. di Milano.

SIR. (2018). *Software-artifact infrastructure repository.* Retrieved 2018-02-20, from `http://sir.unl.edu`

Souza, S. R. S., Brito, M. A. S., Silva, R. A., Souza, P. S. L., & Zaluska, E. (2011). Research in concurrent software testing: A systematic review. In *Proceedings of the workshop on parallel and distributed systems: Testing, analysis, and debugging.*

Souza, S. R. S., Souza, P. S. L., Brito, M. A. S., Simao, A. S., & Zaluska, E. J. (2015). Empirical evaluation of a new composite approach to the coverage criteria and reachability testing of concurrent programs. *Softw. Test. Verif. Reliab.*, *25*(3), 310–332.

Tai, K., Carver, R., & Obaid, E. (1989). Deterministic execution debugging of concurrent Ada programs. In *Proceedings of the computer software and applications conference* (pp. 102–109).

Wu, G., & Kaiser, L. (2011). Constructing subtle concurrency bugs using synchronization-centric second-order mutation operators. In *Seke* (p. 244-249).