UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Study and definition of project attributes for selection of testing techniques for concurrent software

Italo de Oliveira Santos

Dissertação de Mestrado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CMC)

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Italo de Oliveira Santos

Study and definition of project attributes for selection of testing techniques for concurrent software

Master dissertation submitted to the Institute of Mathematics and Computer Sciences – ICMC-USP, in partial fulfillment of the requirements for the degree of the Master 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 2020

Italo de Oliveira Santos

Estudo e definição de atributos de projeto para seleção de técnicas de teste de software concorrente

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestre 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 2020

Thank God above all else without him I would be nothing.

To my family for being my base, everything I am today owes them.

To my professors who have awakened in me the desire to research and study ever more.

To the friends who walked along with me this walk.

ACKNOWLEDGEMENTS

I am indebted to many people, without whom this volume would never seen the daylight. First, I would like to start by thanking God for the gift of my life, without him nothing would be possible, I am deeply grateful for my health, strength to move on and face new challenges and for always being with me during my life journey.

I would like to express my deepest gratitude to my parents. My father, Francisco Jose, for all the teachings and lessons you taught me that shaped the person I became. My mother, Lucilene Alves, for the affection and constant love throughout my creation and for being always available to help me. There are no words to describe my feeling of gratitude for them, so I would like to dedicate the lyrics of a song that synthesizes my feelings: *"You raise me up, so I can stand on mountains, you raise me up to walk on stormy seas, I am strong when I am on your shoulders, you raise me up to more than I can be"*. Also, I am eternally indebted with my brother Iuri Santos, for the friendship and the moments we have shared since the beginning of our lives, thanks for being the best brother ever.

My sincerest gratitude to my extended family sent by God which are my friends: Daniel, Jocilene, Felipe, Nilvan, Natan, Natalia, Janaina, Thamires, Gaby, Hugo, Ana, Jonathan, Hareta, Filho, Gessica, Dinha, Maira, Gibson, Rafael, Lorena, Beth, Bete, Katya, Rodolpho, Paula, Kassia, Emanuel, Andrea, Luana, Janusia, Anderson, Joanna. They shared with me several moments of happiness and sadness, we have always been present helping each other in what was possible and it is really good to know that they are there to support me.

A special thanks to Priscila and her family. I was fortunate to meet her during my undergraduate course and had the pleasure of working on research projects. I am grateful for the encouragement to pursue the research path, I have no words to express how you changed my life. Thanks for helping me so I could begin my master's and achieve this dream.

My heartfelt gratitude to my advisor Simone for the friendship and kindness. She received me in São Carlos so well, thanks for making me feel at home even being far away. I could not possibly wish for a better or friendlier advisor. I will be forever grateful to have had such an amazing mentor like her. To me, you are more than a teacher, a friend, an inspiration and an example that I want to follow. Thanks for everything I learned from you, not only about technical concepts but also about life and how to be a better person and research. You are an inspiring person and always knew how to motivate me to learn more. You owe all my gratitude. Thank you for leading me on this master's journey, for the hours you spent guiding me in the development of this work, and the effort you put to make this work possible.

A special thanks to Paulo Sérgio for supporting the evolution of this project, always contributing with feedback and your knowledge. I thank Silvana for the help and dedication to this research, I appreciated your help. I also want to thank the undergraduate students that I had the opportunity to mentor, Julio Cesar and Andre Baconcello. It was an amazing experience to be able to work with them and I am grateful for the trust in me so we could do a great job together.

To the professors of the ICMC who make up the team of PPG-CCMC, which I had the opportunity to study and learn, I am very thankful in general for all the content that you have passed along with so much affection and disposition, always seeking to lapidate our skills.

I thank the authors of the many papers that I have read during my research who enlightened me with good ideas and taught me new things.

It was an absolute pleasure to work and learn with and from all of my colleagues from the Software Engineering Lab (LabES): Misael, Jorge, Pedro, Lucas, Laiza, Brauner, João Paulo, Livia, Katia, Ricardo, Raul, Anderson. A special thanks go out to: Leo for always being available to help me with great ideas and suggestions; Stevão for the countless times that we discussed possibilities to make this study better; Thanks to Claudinei for his hospitality and sense of humor. Thanks to Allan and João Choma for the time dedicated to reviewing this study.

My true appreciation to all the friends I have made while I was a member at CVU São Carlos: Sara, David, Enzo, Tata, Douglas, Bruna, Rodrigo, Brenda, Duda, Vic, Arielton, and João. You are very special and amazing people, I hope that we can meet at other moments and that our determination to help who needed never end.

I also would like to mentioned and thanks to all the friends I have made since the moment I arrived in São Carlos: Atila, Piva, Milagros, Guilherme, Silvinha, Franciele, Renato, Andrea, Douglas, Flavio. I want to wish success and thank you for the moments we spent together.

I would like to express my gratefulness to all the members of the São Carlos Choir, I am very grateful for the moments we spent together in rehearsals learning and singing new songs and in the performances.

A special thanks goes out to Patricia Leme (Pazu), for the yoga and meditation practices given at USP and for her dedication and care in helping to improve the quality of life of students.

Thanks to Angela Giampetro for the dedication and support in reviewing my english texts, for always welcoming me in your office, I learned many things from you.

Special thanks to the librarians at USP campus São Carlos for always promoting a welcoming environment in the library, which is favorable for studies and research.

Thanks to the professors that accepted to be in my defense committee: André Endo (UTFPR), Marcelo Eler (EACH - USP), Márcio Delamaro (ICMC-USP). I am grateful for their ideas and contributions to improving this work.

I acknowledge to the staff of the graduate program in the name of Bruno Mohamad,

Lycaena and Marcos Cola for the hard work in always trying to solve the needs of the students and the disposition to always receive us in coordination to solve problems.

I want to thank the University of São Paulo (USP) for providing a quality environment for research, study, daily living and extend my gratitude for all university staff.

I dedicate a special thanks to CNPq and FAPESP (São Paulo Research Foundation), for the financial support under process number 2018/10183-9, which made this research possible.

*“The good you do today, will often be forgotten. Do good anyway.
Give the best you have, and it will never be enough. Give your best anyway.
In the final analysis, it is between you and God. It was never between you and them anyway.”*
(Mother Teresa)

RESUMO

SANTOS, I. O. **Estudo e definição de atributos de projeto para seleção de técnicas de teste de software concorrente**. 2020. 134 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2020.

[Contexto:] A escolha de uma técnica de teste a ser adotada em um projeto de teste de software é baseada no conhecimento do testador e geralmente não considera todas as técnicas de teste disponíveis. A seleção da técnica correta em qualquer estágio é um dos problemas críticos no desenvolvimento de software. Em um projeto anterior deste grupo de pesquisa, uma estrutura foi definida para apoiar a seleção sistemática de técnicas de teste de software concorrente. Um esquema de caracterização foi proposto e implementado na ferramenta SeleCTT, composta por um conjunto de atributos que considera características de programas concorrentes. A estrutura calcula a adequação de atributos para orientar a seleção de técnicas de teste para um projeto de software. **[Objetivo:]** A seleção de uma técnica de teste em cada estágio do ciclo de vida de um software depende de muitos fatores. Este trabalho tem como objetivo contribuir para a atividade de selecionar técnicas de teste para programas concorrentes, investigar a caracterização dos atributos de projetos e propor uma abordagem de combinação de técnicas de teste para melhorar o processo de seleção na SeleCTT. **[Metodologia:]** Foi elaborado um catálogo com 61 atributos de seleção usados em abordagens de seleção de técnicas de teste, e a maioria dos atributos usados está concentrada na categoria “características gerais de teste” e em “projeto de software”, o que destaca a importância de tais categorias para apoiar a seleção. Além disso, foi proposta uma combinação dos atributos de seleção incorporados na SeleCTT. Após o cálculo da adequação da técnica de teste de acordo com os dados de um projeto de software informado como entrada, a SeleCTT recomenda mais de uma técnica, considerando informações relacionadas aos atributos *erros concorrentes* e *nível de teste*. Orientando o usuário na aplicação de técnicas de teste complementares que podem encontrar diferentes falhas. **[Resultados e Conclusões:]** Os resultados sugerem que nossa abordagem de seleção combinada alcançou maior eficácia na seleção de técnicas de teste para programas concorrentes. A escolha de uma boa técnica de teste pode melhorar a qualidade de um projeto de software e até reduzir o custo de desenvolvimento. Os resultados obtidos com as contribuições feitas no processo de seleção das técnicas de teste são promissores e contribuem para a qualidade da atividade de teste.

Palavras-chave: Teste de Software Concorrente, Atributos de Projeto, Seleção de Técnicas.

ABSTRACT

SANTOS, I. O. **Study and definition of project attributes for selection of testing techniques for concurrent software**. 2020. 134 p. Dissertação (Mestrado em Ciências – Ciências de Computação e Matemática Computacional) – Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos – SP, 2020.

[Context:] The choice of a testing technique to be adopted in a software testing project is based on the tester's knowledge and often does not consider all testing techniques available. The selection of the right technique at any stage is one of the critical problems in software development. In a previous project of this research group, a framework was defined for supporting the systematic selection of concurrent software testing techniques. A characterization scheme was proposed and implemented in SeleCTT tool, which is composed of a set of attributes that considers characteristics of concurrent programs. The framework calculates suitable attributes for guiding the selection of testing techniques for a particular software project. **[Objective:]** The selection of a testing technique at each stage of a software's life cycle depends on many factors. This work aims to contribute to the activity of selecting testing techniques for concurrent programs, investigating the characterization of project attributes, and proposing an approach of combining testing techniques to improve the selection process of SeleCTT. **[Methodology:]** A catalog with 61 selection attributes used for testing techniques selection was elaborated, and most attributes used are concentrated in the "general testing characteristics" category and, then, in "software project", which highlights the importance of such categories for supporting the selection. Moreover, a combination of the selection attributes incorporated in SeleCTT was proposed. After the calculation of the testing technique suitability according to data of a software project informed as input, SeleCTT recommends more than one technique, considering information related to *concurrent bugs* and *testing level* attributes. It, therefore, guides the user in the application of complementary testing techniques that can find different faults. **[Results and Conclusions:]** The results suggest our combined selection approach has achieved higher effectiveness in the selection of testing techniques for concurrent programs. Choosing a good testing technique can improve the quality of a software project and even reduce the development cost. The results obtained from the contributions made in the process of selecting testing techniques are promising and contribute to the quality of the testing activity.

Keywords: Concurrent Software Testing, Project Attributes, Testing Techniques Selection.

LIST OF FIGURES

Figure 1 – Main criteria adopted to support decision-making on the selection of a testing technique.	25
Figure 2 – Overview of the research stages.	28
Figure 3 – Terminology of testing terms.	32
Figure 4 – Test activity.	33
Figure 5 – Test phases.	34
Figure 6 – Example of non-determinism between processes.	40
Figure 7 – Overview of the experiment process (WOHLIN <i>et al.</i> , 2012).	41
Figure 8 – Methodology for the design of the catalog.	46
Figure 9 – Catalog of selection attributes - Features.	48
Figure 10 – Initial interface of the catalog.	48
Figure 11 – Description interface of the catalog.	49
Figure 12 – Categories interface of the catalog.	49
Figure 13 – Suggest works interface of the catalog.	50
Figure 14 – Distribution of attributes extracted from selected studies classified by categories.	56
Figure 15 – SeleCTT-v1 process overview.	58
Figure 16 – SeleCTT-v2 recommendation of results page.	62
Figure 17 – Plus icon to compare information.	63
Figure 18 – Selection of a testing technique.	64
Figure 19 – SeleCTT-v2 feedback page.	64
Figure 20 – Overview of the body of knowledge of software projects.	65
Figure 21 – Body of knowledge - project details.	65
Figure 22 – Characterization of the projects.	66
Figure 23 – Forward snowballing overview.	70
Figure 24 – SeleCTT-v2 process overview.	79
Figure 25 – Overview of case study activities.	84
Figure 26 – Confidence interval for SeleCTT (v1 and v2) results.	90
Figure 27 – Characterization of the participants.	91

LIST OF TABLES

Table 1 – Testing techniques selection approaches (SANTOS <i>et al.</i> , 2019)	27
Table 2 – Equivalence partitioning criterion	35
Table 3 – Errors in concurrent programs (SOUZA <i>et al.</i> , 2017)	38
Table 4 – List of selected studies that compose the catalog	47
Table 5 – General testing characteristics category	52
Table 6 – Tools category	52
Table 7 – Test team category	53
Table 8 – Software project category	53
Table 9 – ODC faults category	54
Table 10 – Web applications category	54
Table 11 – Concurrent software category	54
Table 12 – Model-based testing category	55
Table 13 – Attributes of concurrent software testing techniques implemented in SeleCTT (MELO, 2018)	60
Table 14 – Template of project characterization	68
Table 15 – Forward snowballing inclusion and exclusion criteria (MELO <i>et al.</i> , 2019a) .	69
Table 16 – Iteration forward snowballing results	70
Table 17 – List of studies selected	72
Table 18 – Information of studies - Part I	74
Table 19 – Information of studies - Part II	76
Table 20 – Overview of characterized software projects	85
Table 21 – Software testing techniques for concurrent programs selected for our case study	86
Table 22 – Oracle of testing techniques	87
Table 23 – Results from SeleCTT evaluation	88
Table 24 – Effectiveness values of SeleCTT recommendations (v1 and v2) for software projects	89
Table 25 – Results of participants' recommendations	92
Table 26 – Effectiveness values of participants' recommendations for software projects .	92

CONTENTS

1	INTRODUCTION	23
1.1	Context	23
1.2	Motivation	26
1.3	Related work	26
1.4	Research objectives	27
1.5	Methodology	28
1.6	Work structure	30
2	THEORETICAL BACKGROUND	31
2.1	Software testing	31
2.1.1	<i>Functional testing technique</i>	34
2.1.2	<i>Structural testing technique</i>	36
2.2	Concurrent programming	36
2.3	Concurrent software testing	37
2.4	Empirical software engineering	39
2.4.1	<i>Overview of empirical strategies</i>	40
2.4.2	<i>Experiment process</i>	41
2.5	Final remarks	43
3	CATALOG OF SELECTION ATTRIBUTES	45
3.1	Catalog design	45
3.2	Catalog of selection attributes	47
3.3	Discussion	51
3.4	Final remarks	55
4	SELECTT-V2	57
4.1	Overview of SeleCTT-v1	57
4.2	Combined selection	61
4.2.1	<i>Novel features</i>	62
4.3	Concurrent project characterization	65
4.4	Forward snowballing	67
4.4.1	<i>Research question</i>	68
4.4.2	<i>Inclusion and exclusion criteria</i>	68
4.4.3	<i>Identification of seed set</i>	69

4.4.4	<i>Iterations</i>	69
4.4.5	<i>Overview of results</i>	71
4.4.6	<i>Analysis and discussion</i>	71
4.4.7	<i>Threats to validity</i>	78
4.5	Final remarks	78
5	COMBINED SELECTION EVALUATION	81
5.1	Case study design	81
5.1.1	<i>Rationale for the study</i>	81
5.1.2	<i>Study goals</i>	82
5.1.3	<i>Research questions</i>	82
5.2	Data collection	83
5.2.1	<i>Instruments</i>	83
5.2.2	<i>Procedures</i>	84
5.2.3	<i>Data analysis</i>	86
5.3	Results and interpretation	87
5.3.1	<i>Scenario 1: Quantitative analysis</i>	87
5.3.2	<i>Scenario 2: Quantitative analysis</i>	91
5.3.3	<i>Comparison between Scenarios</i>	94
5.4	Threats to validity	95
5.5	Final remarks	95
6	CONCLUSIONS	97
6.1	Results	97
6.2	Revisiting the contributions	98
6.3	Limitations	99
6.4	Future work	99
6.5	Publications	100
	BIBLIOGRAPHY	103
APPENDIX A	PROJECT CHARACTERIZATION	111
APPENDIX B	CHARACTERIZED PROJECTS LIST	119
APPENDIX C	FORWARD SNOWBALLING SEED SET LIST	123
APPENDIX D	PARTICIPANT CHARACTERIZATION FORM	129
APPENDIX E	PARTICIPANT FEEDBACK FORM	133

INTRODUCTION

1.1 Context

Software testing is one of the Verification, Validation, and Testing (VV& T) activities used to increase software quality. Such activities reveal the presence of defects and check whether the software matches its specification ([MYERS; SANDLER; BADGETT, 2011](#)). They have become increasingly difficult, due to the evolution of computing, which includes a wide variety of application domains, programming languages, operating systems, hardware platforms, and electronic devices. According to [Myers, Sandler and Badgett \(2011\)](#), although relatively few people used computers in the 1970s, presently, no one can complete a day's work without a computer, and the machines are hundreds of times more potent than the first computing devices.

[Ammann and Offutt \(2016\)](#) highlighted that testing is one of the most expensive activities in a software development process, since it consumes a large part of the total cost of a project. However, it cannot be overlooked when the aim is the development of a high-quality software. The ideal scenario would be the testing of all possible input permutations of a program, which is not possible in most programs ([MYERS; SANDLER; BADGETT, 2011](#)). Even a seemingly simple program may have hundreds or thousands of possible combinations of input and output, which hampers the creation of test cases for all possibilities. The complete testing of a complex system would require considerable and many human and computational resources to be feasible.

Computational devices are continually evolving towards achieving higher computational performance. Parallel machines with multiple processing elements working simultaneously to solve the same problem in less are usual nowadays. This computing environment enables the development of parallel (or concurrent) programs to solve more complex problems. These programs bring new challenges to software testing activity. According to [Melo \(2018\)](#), the main challenges are (i) the high number of synchronizations between processes or *threads* to be exercised during a test, (ii) detection and correction of typical errors of this type of software (e.g.,

deadlock, atomicity violation, and race conditions), and (iii) development of efficient methods of automatic generation of test data that facilitate and reduce the test activity costs. Several works have been developed concerning these challenges (LU; JIANG; ZHOU, 2007; SARMANHO *et al.*, 2008; SOUZA *et al.*, 2008; KOJIMA *et al.*, 2009).

The selection of a testing technique to be implemented in a software testing project is based on the knowledge of the tester and often does not consider all testing techniques available in the industry or academia. According to Vegas and Basili (2005), this is due to the lack of both evidence about their use in software projects and a body of knowledge that gathers empirical evidence of their applicability.

Melo (2018) proposed a body of knowledge that integrates information relevant to the decision-making process on the selection of testing techniques for a particular concurrent software project. Moreover, an experiment design that acts as a guide for the empirical studies was defined and can be used for the feedback, update, and evolution of the body of knowledge. The testing technique selection was systematized by a characterization scheme that considers the main characteristics of the concurrent programming that influence the software testing activity and calculates their adequacy to the attributes of the project in development. The characterization scheme is composed of 21 attributes. They are classified into four categories, according to the type of information they offer:

- **Programming model:** the programming model used by the software under test (execution platform, development paradigm, programming language);
- **General testing characteristics:** the type of technique, approach for test data generation, testing level, input and output of the test case, quality assessment attributes, and empirical study used for validation (VEGAS; BASILI, 2005; DIAS-NETO; TRAVASSOS, 2009a);
- **Tool support:** tool identification, language and operating platform (VEGAS; BASILI, 2005);
- **Concurrent testing characteristics:** aspects of specific testing technologies for concurrent programming.

Melo *et al.* (2019b) developed SeleCTT (An infrastructure for the Selection of Concurrent Software Testing Techniques), a tool that provides access to the body of knowledge and automates the selection process for the interaction between the community and the proposed framework. Based on the characteristics of the software project under test and the attributes of the testing techniques presented in the body of knowledge, the approach suggests the best testing technique. However, only one is suggested, whereas the ideal solution would be to check the possibility of combining more than one technique for improving results. Besides, the choice does not take into account information from previous projects.

In this sense, this project contributes to the proposition of a combined selection approach for software testing techniques. This project extends the previous work developed by [Melo *et al.* \(2019b\)](#), [Melo \(2018\)](#), [Moura, Melo and Souza \(2018\)](#), defining attributes that can guide this combination of techniques, aiming to cover the complementary aspect of the techniques. We have looked for other approaches defined in the literature for the selection of testing techniques, mapping their state-of-the-art, and investigating the attributes that support this selection.

A survey was conducted with testers from the industry to understand the way they select testing techniques. The results indicate the factors considered essential for the software testing selection. According to Figure 1, the scope of the project is the most adopted criterion for decision-making about which testing technique to adopt (124 responses). Another criterion that supports decision-making was time (71 responses). The tester chooses a technique according to the time specified in the schedule. Risk, which regards the implementation of the technique, and cost were given the same number of responses (26). Therefore, we can infer testers are more concerned about ensuring the best software technique is applied without considering cost as the most important criterion. Moreover, the complexity of the technique (20 responses) also supports decision-making, and the criterion with fewer answers was previous experiences (12 responses) of testers. Other criteria (e.g., literature search, analysis of technical limitations of each technique, among others) were indicated, however, since they were cited less frequently, they are not directly shown in the chart.

Figure 1 – Main criteria adopted to support decision-making on the selection of a testing technique.

We also analyzed the selection attributes used in the literature to support testing technique selection approaches, through this analysis, we developed a catalog of selection attributes to aid testers in performing the testing techniques selection. Another contribution of our research is the proposal of new features and their implementation in SeleCTT. Among such features are a body of knowledge of projects that stores information on projects registered by users, thus benefitting others, who can visualize the techniques recommended and applied in a testing activity, and possibility for them to give feedback on the application of such techniques.

The selection process conducted by SeleCTT only recommends the most suitable testing technique for a software project. Towards improving it, this research proposes a way to help testers to apply complementary testing techniques stored in the body of knowledge. The combined selection consists in grouping two selection attributes from SeleCTT, namely *type of concurrent bugs* and *testing level*, and showing users their correlation for supporting their decision-making by prioritizing this knowledge.

1.2 Motivation

The appropriate use of an efficient testing technique at any stage of the software development life cycle improves the quality of the product tested. The several testing techniques available display different and often complementary characteristics, which include cost of application, effectiveness in revealing defects, types of defects, among others. The choice of a testing technique should not only be based on subjective knowledge, but incorporate objective knowledge, guided by elements that favor a good choice (VICTOR; UPADHYAY, 2011).

Although several studies have compared different testing techniques, few have analyzed their applicability, or evaluated the relevant parameters for their applicability conditions (VEGAS; BASILI, 2005). The available information on testing techniques is usually distributed among different sources (books, papers, and even people), i.e., developers do not have a general idea of the techniques and the relevant information on them. Moreover, they do not have access to practical information on each testing technique, unless they have used it.

Developers do not tend to share the knowledge acquired from the use of the testing techniques (VEGAS; BASILI, 2005); consequently, they do not learn from other experiences. The challenge is the identification of relevant information for the selection of techniques, therefore, the aim is to help testers choose those most appropriate for each project during the testing technique selection process.

1.3 Related work

A systematic mapping conducted by Santos *et al.* (2019) identified some relevant approaches for the selection of testing techniques. Their classification is shown in Table 1.

The characterization schema is a strategy of selection that consists of gathering relevant information about software testing techniques for the construction of a repository that stores the description of each technique according to a defined pattern. Therefore, a decision can be made on whether or not to use a technique (VEGAS; BASILI, 2005; ISABEL; TRAVASSOS, 2011; MELO *et al.*, 2017; VEGAS, 2004; VEGAS; JURISTO; BASILI, 2006; COTRONEO; PIETRANTUONO; RUSSO, 2013b).

Table 1 – Testing techniques selection approaches (SANTOS *et al.*, 2019)

Approach Type	Description
Characterization Schema	Gathers relevant information about software testing techniques
Framework for testing selection and combination techniques	Selects, compares and evaluates testing techniques
Machine learning	Uses offline and online strategies to take historical information about techniques and adapts the selection.
Multi-Criteria Decision Making Selection	Considers the selection of the best testing technique at any phase of the software development life cycle
Porantim	Supports the unique or combined selection of MBT techniques for a given software project considering adequacy level, project characteristics and impact of more than one technique

Porantim is another selection approach proposed and has its origin in the characterization scheme, which uses a body of knowledge of model-based testing (MBT) techniques that works as a repository of techniques that might be used in a given software project (NETO; TRAVASSOS, 2009; NETO; RODRIGUES; TRAVASSOS, 2011; GRANDE; NETO; RODRIGUES, 2012; DIAS-NETO; TRAVASSOS, 2014).

Two different studies proposed a framework approach that selects, compares and evaluates the selection of testing techniques. Wojcicki and Strooper (2007) presented a framework that applies a systematic strategy focused on maximizing completeness and minimizing efforts using metrics, such as effort and effectiveness. The framework proposed in Rothermel and Harrold (1996) analyzes and evaluates regression testing selection techniques, and consists of four categories, namely inclusiveness, precision, efficiency, and generality.

Cotroneo, Pietrantuono and Russo (2013a) designed an approach that uses machine learning for the classification and prediction of the techniques' behavior, and Victor and Upadhyay (2011) developed a multi-criteria decision-making selection strategy that considers the selection of the best testing technique at any phase of the software development life cycle.

1.4 Research objectives

The main objective of this project is to answer the following question: *"What project attributes can guide the combined selection of testing techniques for concurrent software?"*, towards extending and improving the attributes defined by Melo (2018), and taking into account other characteristics (e.g., information from previous similar projects and combination of testing techniques for improving the results).

The specific objectives are:

- Identification of relevant attributes of concurrent software projects;
- Definition and implementation of the combined selection in SeleCTT tool; and
- Evaluation of the combined selection approach.

1.5 Methodology

The following methodology was defined (Figure 2) for the achievement of the goals, and seven activities were performed (below is a short description of them).

Figure 2 – Overview of the research stages.

1. **Systematic mapping study:** we identified and analyzed papers that represent the current state-of-the-art of the testing techniques selection. The results led to a discussion on (i) testing techniques selection approaches; (ii) a classification schema of approaches for testing techniques selection; (iii) types of empirical methods that evaluate the proposals; (iv) a complete list of attributes that supports the choice of the best technique for a software project; and (v) approaches that combine testing techniques. The combination of selection approaches improved the development process ([SANTOS *et al.*, 2019](#));
2. **Brazilian companies survey:** a survey on software testing practices conducted in Brazilian software companies provided an overview of the latest testing techniques, tools, and metrics used, challenges faced by testers, and the process of testing technique selection,

which is the central concern of this research. Furthermore, it identified the software testing practices most used by testers. The challenges faced by testers can provide interesting topics to be explored more deeply by researchers;

3. **Study of selection attributes:** one of the aims of the systematic mapping was the identification of attributes used by the testing techniques selection approaches. 10 out of 15 papers cited attributes that support the selection process. 178 attributes were extracted, and a careful analysis conducted excluded duplicated ones. Some attributes had the same purpose, despite their different nomenclatures, and were gathered in one attribute. In the end, 61 attributes were selected. The study of the selection attributes was essential for the verification of possible attributes related to concurrent software to be incorporated in SeleCTT. The analysis revealed no new attributes that might be implemented or influence our combined selection approach;
4. **Catalog of selection attributes:** the study conducted in the previous stage enabled the development of a catalog of selection attributes for assisting testing practitioners in the choice of the most suitable testing techniques for a particular software project. The catalog was classified into 8 categories, namely (i) general testing characteristics, (ii) tools, (iii) test team, (iv) software project, (v) orthogonal defect classification (ODC) faults, (vi) web applications, (vii) concurrent software, and (viii) model-based testing. It is expected to foster collaboration between industry and academia for improvements in the quality of software developed;
5. **Definition of a combined selection approach:** a combination of the selection attributes incorporated in SeleCTT was proposed. In the selection process, after the calculation of the testing technique suitability according to data of a software project informed as input, SeleCTT recommends more than one testing technique, considering information related to *concurrent bugs* and *testing level* attributes. It, therefore, guides the user in the application of complementary testing techniques that can find different faults, recognizing the testing level of the users interest and the technique to be applied;
6. **Evolution of SeleCTT tool:** this project has added novel features to SeleCTT towards a combined selection of testing techniques, thus evolving its scope proposed in [Melo et al. \(2019b\)](#), [Melo \(2018\)](#), [Moura, Melo and Souza \(2018\)](#).
7. **Evaluation of the combined selection approach:** the last step consists in the evaluation of the proposed combined selection approach, which was validated through a case study divided into two scenarios to compare the effectiveness of the recommendations made by SeleCTT (v1 and v2) in scenario 1 and the effectiveness of the recommendations made by the testers in scenario 2.

1.6 Work structure

The remaining chapters are organized as follows:

- Chapter 2 introduces some concepts of software testing and concurrent programming, discusses concurrent software testing, and highlights its importance to empirical software studies in the field of software engineering.
- Chapter 3 presents the proposed unified catalog of selection attributes highlighting the features designed to assist software testing practitioners who select testing techniques to be applied in a software project.
- Chapter 4 describes the activities conducted towards improving SeleCTT features (e.g., forward snowballing for updating the body of knowledge of concurrent testing techniques and characterization of 20 software projects).
- Chapter 5 reports the case study conducted to evaluate the combined selection approach and the analysis and discussion of the results.
- Chapter 6 summarizes the research, showing an overview of the results, contributions, limitations faced, and possibilities for future research.

THEORETICAL BACKGROUND

This chapter addresses concepts related to software testing and concurrent programming and describes more about concurrent software testing and their importance to empirical software studies in the field of software engineering.

Section 2.1 discusses the importance of testing activity in software development process and describes some terminology commonly used in the literature. It also introduces the testing phases and activities involved and explains some testing techniques and criteria.

Section 2.2 presents general concepts related to concurrent programming, and Section 2.3 discusses the main errors and challenges in concurrent software testing. It also describes the family of structural testing criteria for concurrent software testing related to the message-passing and shared memory paradigm, and address non-deterministic behavior in concurrent testing.

Section 2.4 highlights the importance of empirical software engineering. It provides an overview of the most commonly used empirical strategies and details the activities of the experiment process.

2.1 Software testing

The view of software testing has matured into a constructive one in recent years. Testing is no longer considered an activity that starts only after the coding phase has been completed and is limited to the detection of failures (BOURQUE; FAIRLEY *et al.*, 2014). It has been widely used in the industry for quality assurance, since, by directly scrutinizing the software in execution, it provides realistic feedback of the software behavior, and remains the essential complement to other analysis techniques. Testing embraces a variety of activities, techniques, and actors, but also poses many complex challenges (BERTOLINO, 2007).

The literature has established specific meanings for terms related to software testing (e.g., failure, fault, error, and mistake) (DELAMARO; JINO; MALDONADO, 2017). According

to [IEEE \(1990\)](#) a **fault** is a definition of incorrect data, a **mistake** is a human action that leads to an incorrect result, an **error** is the difference between the value obtained and the expected value, and a **failure** is an unexpected software behavior different from that expected by the user, such terms describe an incorrect function of the system. Figure 3 shows the terminology that distinguishes them.

Figure 3 – Terminology of testing terms.

According to [Myers, Sandler and Badgett \(2011\)](#), the test activity must be performed in the following four distinct steps in each stage of the test process: (i) **planning**: the test project is planned and includes activities (e.g., definition of the software to be tested, the establishment of the time to be spent on the test and time limit for its purpose, preparation of the test environment structure, selection of both the team responsible for the execution and the automation tools, among other necessary resources); (ii) **test case design**: test cases are defined and test scenarios and scripts are prepared and reviewed; (iii) **execution**: describes the time that the previously planned test cases are executed and the results are recorded; and (iv) **analysis**: after the running of the test cases, the results are analyzed regarding the software behaving under its specification. Figure 4 shows the steps of the test activity.

Among the steps presented, the design of test cases deserves careful attention, since it includes the selection of test cases of a higher chance of identifying defects, which avoids additional efforts in the execution of inefficient test cases. Techniques and testing criteria are developed towards assisting this activity. Testing techniques provide standards for a systematic design of test cases that exercise both the internal logic of software components and their input and output domains ([DELAMARO; JINO; MALDONADO, 2017](#)).

The quality of a test case is directly associated with the strength of the testing criterion used and defines the properties or requirements to be considered in the evaluation of the tests. According to [Melo \(2018\)](#), the testing criteria can be defined according to their function performed. Test data selection criteria are procedures for the choice of test cases, and adequacy criteria of

Figure 4 – Test activity.

test data are used in evaluations of test data. According to [Ammann and Offutt \(2016\)](#), testing techniques and criteria are fundamental for the selection of test cases, since they minimize their number, thus ensuring the tests cover specific parts of the code.

The test case design must reveal faults that occur in software. Such faults can be identified in all phases of the development process, consequently, the test activity must be performed throughout the software development lifecycle. The test procedure involves the following distinct phases ([BOURQUE; FAIRLEY *et al.*, 2014](#); [DELAMARO; JINO; MALDONADO, 2017](#)):

- **Unit testing:** verifies the isolated functioning of separately testable software elements. Depending on the context, they can be individual subprograms or a more significant component made of highly cohesive units;
- **Integration testing:** applied after the units have been tested individually, it ensures interactions among software components are working correctly;
- **System testing:** tests the behavior of an entire system. It is usually considered appropriate for assessing nonfunctional system requirements (e.g., security, accuracy, and reliability).

Such phases are applied during the software development process, unit testing can be applied while the units are implemented, i.e., before the system is finished. Integration testing must be performed after the units have been tested individually, since the various parts of the software work together. The appropriate functioning of the interaction between them must be checked to not lead to errors. The system testing is conducted after the systems have been completed and all their parts have been integrated ([DELAMARO; JINO; MALDONADO, 2017](#)).

Regression testing must also be highlighted ([DELAMARO; JINO; MALDONADO, 2017](#)), it is applied when changes have been made in the software, and assures newly introduced changes do not obstruct the behavior of the unchanged part of the software ([YOO; HARMAN, 2012](#)). In practice, the approach should verify whether the software still supports the previous

tests in a test suite. During incremental development, regression testing aims to show the software behavior is unchanged by incremental changes to the software (BOURQUE; FAIRLEY *et al.*, 2014). Figure 5 shows such test phases. Regression test is at the intersection of the phases. Since the software under test is updated and must apply regression test, test cases developed in the three phases can be employed to show that the new modifications do not change the behavior of the software.

Figure 5 – Test phases.

According to Delamaro, Jino and Maldonado (2017), testing techniques differ from each other by the source of information used for the selection of test data. The functional technique is based on the specification of the software product, whereas the structural technique uses the internal structure of the program. The defect-based technique employs typical defects occurred during the software development process, and the model-based technique requires the use of a formal model or a formal language to describe the systems' specifications. The next sections detail the functional and structural testing techniques and their testing criteria.

2.1.1 Functional testing technique

The program or system under test is observed as a “black box”, and the choice of test cases for functional testing is based on the requirement or design specification of the software under test (NIDHRA; DONDETI, 2012). Functional testing focuses mainly on the external behavior of the software entity (MITRA; CHATTERJEE; ALI, 2011), and its criteria are (MYERS; SANDLER; BADGETT, 2011): (i) equivalence partitioning, (ii) boundary value analysis, (iii) cause-effect graphing, and (iv) error guessing. We will present the first and second below.

- Equivalence partitioning: reduces the number of test cases to a manageable level, while still maintaining reasonable software functionality coverage. This technique is used

intuitively by almost all testers, even if they are not aware it is a formal test design method (COPELAND, 2004).

Let us consider the following situation: we are developing a program to sell tickets to concerts and must establish rules to give discounts according to the age range of the public. The partitioning criterion of equivalence is shown in Table 2.

Table 2 – Equivalence partitioning criterion

Age	Output
<7 & 60>	Do not pay;
8 - 20	Receive 50% discount;
21 - 59	No discount;

Run all possible inputs from a specific range is not required during test. Only one value of each class or each range must be tested. An assumption is that any test within that range is just as good as any other; the same applies to the other ranges, called equivalence classes. It consists of a set of data treated the same by the module or that should produce the same result regarding testing and any data value within a class is equivalent to any other value (COPELAND, 2004).

The equivalence partitioning criteria expected that: (i) if a test case in an equivalence class detects a defect, all other test cases in the same equivalence class should detect the same defect, and (ii) if a test case in an equivalence class does not detect a defect, no other test cases in the same equivalence class should detect it.

- **Boundary value analysis:** focuses on boundaries where defects are hidden. Experienced testers have frequently faced this situation, whereas inexperienced ones may have an intuitive feel that mistakes will occur most often at the boundaries (COPELAND, 2004). Experience has shown test cases that explore boundary conditions have a higher payoff. Boundary conditions are found directly on, above, and beneath the edges of both input and output equivalence classes. The following guidelines are used to apply the boundary value (MYERS; SANDLER; BADGETT, 2011):

1. In an input condition that specify a range of values, test cases must be written for the ends of the range, and invalid-input test cases for situations beyond the ends;
2. In an input condition that specifies the number of values, test cases must be written for the minimum, maximum numbers of values, one beneath and beyond these values;
3. Use guideline 1 for each output condition;
4. Use guideline 2 for each output condition;
5. If the input or output of a program is an ordered set, attention must be devoted to the first and last elements of the set; and
6. Testers should use their ability to search for other boundary conditions.

2.1.2 Structural testing technique

In structural testing, also known as white-box testing, the internal control structure of a program guides the selection of test data (HOWDEN, 1980). This technique enables analyses of the internal structure of a program and derivation of test data from an examination of the program's logics (MYERS; SANDLER; BADGETT, 2011). White-box is applied at a low-level design and all levels of system development, especially unit, integration, and system testing.

In general, most criteria of structural testing use a program representation, known as a control flow graph. A program P can be decomposed into a set of disjoint blocks of commands. The execution of the first command promotes the execution of all other commands of that block, in a given order. All commands in a block, possibly except the first, have a single predecessor and precisely one successor, possibly except the last command.

According to Delamaro, Jino and Maldonado (2017), structural testing imposes the following limitations and disadvantages due to its inherent limitations as a validation strategy: (i) no general-purpose test procedure can prove the correctness of a program, (ii) given two programs, it is undecidable if they compute the same function, (iii) in general, is undecidable, if two paths of a program, or different programs, compute the same function, (iv) it is undecidable if a given path is executable, that is, if there is an input dataset leading to the execution of that path, (v) missing paths: if the program does not implement some functions, there will be no path that matches that function, and (vi) coincident correction: the program may present a correct result for a particular input, satisfying a test requirement and not revealing the presence of a defect; however if another input is chosen, the result obtained could be incorrect.

The following criteria were established in structural testing (DELAMARO; JINO; MALDONADO, 2017): (i) complexity, (ii) control flow, and (iii) data flow.

2.2 Concurrent programming

Although computers keep getting faster, the demands placed on them are increasing at least as fast. Many areas would benefit from higher computational power. Plenty of computing power is available, especially in science, engineering, and industry, however, it is never enough. Therefore, to be able to handle larger problems, computer scientists are turning more to parallel computers (TANENBAUM, 2016). The development of concurrent programming was originally motivated by the need for reliable operating systems. Principles of concurrent programming have shown highly useful, since they apply to any form of parallel computing (HANSEN, 2001).

The literature reports no full agreement on the distinction among terms parallel, distributed, and concurrent computing and, according to several authors, (i) in concurrent, a program is one in which multiple tasks can be in progress at any instant; (ii) in parallel, a program is one in which multiple tasks cooperate closely to solve a problem; and (iii) in distributed, a program

may need to cooperate with other programs to solve a problem. Although, parallel and distributed programs are concurrent, a program, such as a multitasking operating system is also concurrent, even when it is run on a machine with only one core, since multiple tasks can be in progress at any instant ([PACHECO, 2011](#)).

When a user runs a program, the operating system creates a process (an instance of a computer program being executed) that involves several entities ([PACHECO, 2011](#)), namely (i) executable machine language program, (ii) block of memory that includes the executable code, a call stack that keeps track of active functions, a heap, and some other memory locations, (iii) descriptors of resources allocated by the operating system for the process, (iv) security information, and (v) information about the state of the process.

A concurrent program has two or more processes or threads that work together towards accomplishing a task. A process is a running program composed of a program, a data set, and a descriptor record (the area of memory that stores the contents of the registers when the process is not running). A thread (or light process) is an independent control line within the process ([TANENBAUM; STEEN, 2007](#)). Threading provides a mechanism for programmers to divide their programs into more or less independent tasks when a thread is blocked, another one can be run. Furthermore, in most systems, the switching between threads is much faster than the switching between processes. Threads are contained in processes, so that they can use the same executable, and usually share the same memory and input/output devices. Two threads belonging to one process can share most of its resources ([PACHECO, 2011](#)).

According to [Grama \(2003\)](#), different paradigms can be used in the development of concurrent programs. Processes interact regarding communication and synchronization, and the development paradigms can be (i) shared address space, which enables a uniform access to variables, while this seems to be a benefit, many agents will access this shared resource, therefore mechanisms must ensure the correctness of the results, and (ii) message-passing, which is one of the oldest and most widely used approaches for the programming of parallel computers. Two key attributes characterize this paradigm, i.e., it assumes a partitioned address space and supports only explicit parallelization.

2.3 Concurrent software testing

According to [Yang and Pollock \(1997\)](#), testing concurrent programs is a more complex activity due to the following factors: (i) development of specific static analysis techniques, (ii) detection of unwanted situations (e.g., synchronization errors, communication, data flow errors and deadlock), (iii) reproduction of execution with the same test entry and force of the execution of a path in the presence of non-determinism; (iv) generation of a representation of the concurrent program that captures information pertinent to the test; (v) analysis of the application of sequential testing criteria in the context of concurrent programs; and (vi) design of

data flow criteria for concurrent programs, considering the exchange of messages and shared variables. Melo (2018) also reported the following new challenges to concurrent software testing: (i) reduction in the cost of testing concurrent programs; (ii) automatic generation of test data; and (iii) efficient exploration of the interleaving space of events.

Software testing applied to concurrent programs aims to identify errors related to communication, parallelism, and synchronization. Krawczyk, Wiszniewski and Mork (1994) described two types of errors related to concurrent programs:

- **Observability error:** which is inherent to a testing environment and occurs in product machine (program) P if for some path $p_i \in P$, there exists another path $p_j \in P$, such that $D(p_i) = D(p_j)$ and $C(p_i) \neq C(p_j)$;
- **Locking error:** which occurs in P for some program path $p \in P$, where $p = q_0, q_1, \dots, q_j$ if there exists node q_i in p , such that subpath $p_s = q_0, q_1, \dots, q_i$ has nonempty path domain $D(p_s)$, all its outgoing arcs are labeled with predicates, and all these predicates evaluate to false for some element of $D(p_s)$.

The observability error occurs when the tester cannot control the behavior of the program running in parallel and different sequences of commands are executed and produce an expected result. In this case, an input can produce an output rated as correct, which is not necessarily correct, because of the synchronization sequences executed. Another input value that exercised the same sync sequences might reveal the error. The locking error occurs when a process does not communicate with the correct process, thus causing locking from a certain point in the execution of the program (SOUZA *et al.*, 2017). Table 3 shows the main types of errors in concurrent programs.

Table 3 – Errors in concurrent programs (SOUZA *et al.*, 2017)

Type of Error	Description
Deadlock	occurs when two or more threads block each other
Livelock	occurs when a thread cannot finish the execution or enter a critical region due to overwork or lack of speed
Starvation	occurs when a thread is never staggered to run.
Incorrect Threads Count	occurs due to an incorrect initialization of the number of threads required for the completion of an action
Atomicity Violation	is caused by the concurrent execution of multiple threads, which violates the atomicity of a code region
Unprotected Code	occurs when a critical region of the code is unprotected, which enables multiple threads to change the same data
Data Inconsistency	occurs when different threads have different views of the same variable
Sync Error	is caused by the occurrence of an unexpected synchronization between a thread or process
Decomposition Error	is related to the decomposition of data and functions for the generation of concurrent processes
Message Passing Error	is caused by the use of an erroneous function, which generates an unexpected behavior
Lost Messages	occurs when a message sent to a process is never received
Communication Error	occurs when a wrong message is sent or when it is sent by a wrong process

Structural testing criteria are also used in concurrent software testing, it is possible to establish a family of structural testing criteria for concurrent programs concerning the paradigm

of a message passing (SOUZA *et al.*, 2008) and shared memory (SARMANHO *et al.*, 2008). Below are the definitions of the criteria based on control flow testing:

- **All-Nodes:** all nodes of set N must be exercised at least once by the set of test cases;
- **All-Edges:** all edges of set E must be exercised at least once by the set of test cases;
- **All-Nodes-Sync:** all nodes of set N_{sync} must be exercised at least once by the set of test cases;
- **All-Edges-Sync:** all edges of sets E_{sync} and E_{sync}^p must be exercised at least once by the set of test cases.

The following definitions refer to the criteria based on data flow testing:

- **All-Uses:** all associations c-use, p-use, m-use, s-c-use, and s-p-use must be exercised by the set of test cases;
- **All-m-Uses:** all m-use associations must be exercised by the set of test cases;
- **All-s-Uses:** all s-c-use and s-p-use associations must be exercised by the set of test cases;
- **All-intra-Uses:** all intra-m-use associations must be exercised by the set of test cases;
- **All-inter-Uses:** all inter-m-use associations must be exercised by the set of test cases.

Unlike sequential programs, concurrent programs exhibit a non-deterministic behavior, which hampers the test activity. Non-determinism promotes different executions of a concurrent program with the same input value for the production of different correct outputs (SOUZA *et al.*, 2017). Figure 6 illustrates a scenario of non-determinism with a concurrent program and three parallel processes. The program is divided into a set of processes (Process 1, Process 2 and Process 3) that communicate through the exchange of messages using send (S1, S2, and S3) and receive (R1, R2, and R3) primitives. A race condition is observed among S1, S2, and S3 related to R1, R2, and R3. Each execution (Execution 1, Execution 2 and Execution 3) represents a synchronization sequence that can occur in the program. The testing activity aims to identify all possible synchronization sequences and analyze the outputs generated.

2.4 Empirical software engineering

Software engineering is formally defined by IEEE (1990) as “the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software”. Three aspects in such a definition are of particular importance here. Firstly, it implies a software process through pointing at different life cycle phases; secondly, it stresses the need for a systematic and disciplined approach; finally, it highlights the importance of quantification.

Figure 6 – Example of non-determinism between processes.

Experimentation provides a systematic, disciplined, quantifiable and controlled way of evaluating human-based activities. Empirical studies and experiments, in particular, are also important for researchers in software engineering. New methods, techniques, languages, and tools should not be only suggested, published and marketed, and new inventions and proposals must be compared with current ones. Experimentation provides this opportunity and should be used accordingly (WOHLIN *et al.*, 2012). The objective in this section is to address the way empirical studies and experiments, in particular, fit a software engineering context.

2.4.1 Overview of empirical strategies

According to Robson (2002), depending on the purpose of evaluation and conditions for the empirical investigation, the following three major different types/strategies of investigations can be conducted:

1. **Survey:** a system that collects information from or about people for the description, comparison or explanation of their knowledge, attitudes and behavior (FINK, 2003). The primary means of gathering qualitative or quantitative data are interviews or questionnaires. A sample representative of a population to be studied is taken, and the results of the survey are analyzed towards descriptive and explanatory conclusions.
2. **Case study:** an empirical analysis that draws on multiple sources of evidence for the investigation of an instance of a contemporary software engineering phenomenon within its real-life context, especially when the boundary between phenomenon and context cannot be specified (RUNESON *et al.*, 2012). It is used in research projects, activities or assignments. Data are collected for a specific purpose throughout the study, and statistical analyses are performed. The case study is usually aimed at tracking a specific attribute or establishing relationships among different attributes. The level of control is lower in a case study than in an experiment (WOHLIN *et al.*, 2012).
3. **Experiment:** an empirical inquiry that manipulates a factor or variable of the studied setting. Different treatments based on randomization are applied to or by different subjects;

other variables are kept constant and the effects on outcome variables are measured. In human-oriented experiments, humans apply different treatments to objects, while in technology-oriented experiments, different technical treatments are applied to different objects (WOHLIN *et al.*, 2012).

2.4.2 Experiment process

Experimentation is not simple, since experiments must be properly prepared, performed and analyzed. The main advantages of an experiment are the control of, for example, subjects, objects, and instrumentation, which ensures more general conclusions can be drawn, and statistical analyses with the use of hypothesis testing methods and opportunities for replication. To ensure that we make use of the advantages, we need a process supporting us in our objectives in doing experiments correctly (WOHLIN *et al.*, 2012).

The starting point for an experiment is insight and the idea it would be a possible way of evaluating whatever we are interested in, we have to recognize if an experiment is appropriate for the question we are going to investigate (WOHLIN *et al.*, 2012). The experiment process can be divided into the activities illustrated in Figure 7.

Figure 7 – Overview of the experiment process (WOHLIN *et al.*, 2012).

Scoping: a hypothesis must be clearly stated in this step. The objective and goals of the experiment must also be defined, and a goal is formulated. Basili and Rombach (1988) proposed a framework composed of the following constituents to capture the scope:

- Object of study (what is studied?);
- Purpose (what is the intention?);
- Quality focus (what effect is studied?);
- Perspective (whose view?);
- Context (where is the study conducted?).

The goal template by Basili and Rombach (1988) is: Analyze *<Object(s) of study>* for the purpose of *<Purpose>* with respect to the *<Quality focus>* from the point of view of the *<Perspective>* in the context of *<Context>*.

Planning: the foundation for the experiment is designed, and the context of the experiment is detailed. The hypothesis of the experiment is formally stated and includes a null hypothesis and an alternative one. The next step is the determination of independent variables (inputs) and dependent variables (outputs).

The experiment is then designed, and includes, for example, randomization of subjects. An issue closely related to the design is the preparation of instrumentation. As part of the planning, the validity of results is considered, and can be divided into four major classes, namely internal, external, construct and conclusion. Planning is a crucial step and must ensure the results are useful, since a poor planning may ruin any well-intended study (WOHLIN *et al.*, 2012).

Operation: in principle, this activity consists of three steps, namely (i) preparation, (ii) execution, and (iii) data validation. The former refers to the preparation of both subjects and necessary material (e.g., data collection forms). The participants must be informed on the intention, give their consent, and be committed. The actual execution is normally not a major problem, since the main concern is to ensure the experiment is conducted according to its plan and design, which include data collection. Finally, the collected data must be correct and provide a valid picture of the experiment (WOHLIN *et al.*, 2012).

Analysis and Interpretation: the data collected during operation provide the input to the activity, and are now analyzed and interpreted. The first step is the attempt to understand the data using descriptive statistics, and the next step is to consider whether the data set should be reduced, by either the removal of data points or reduction in the number of variables.

After the removal of data points or reduction in the data set, a hypothesis test is applied. The actual test is based on measurement scales, values of the input data and type of results expected. A crucial aspect is interpretation, i.e., we have to determine from the analysis whether the hypothesis was possible to reject (WOHLIN *et al.*, 2012). The analysis is essential for decision-making and conclusions concerning the use of the results, which includes motivation for further studies.

Presentation and Package: this last activity presents and packs the findings. It includes documentation of the results, either through a research paper for publication, a lab package for replication purposes, or as part of a company's experience base. An experiment will never provide the final answer to a question, hence, its replication must be facilitated. Comprehensive and thorough documentation is a prerequisite for the achievement of this objective (WOHLIN *et al.*, 2012).

Melo, Souza and Souza (2016) defined an experimental design in the form of a structured scheme towards reducing the efforts required in new experiments and facilitating the comparison

of results for similar experiments. It provides a guide that assists in the development of more controlled and replicable studies in the concurrent software testing area. To illustrate a possible objective for an experimental study in the context of concurrent software can be defined in [Melo \(2018\)](#) as: Analyzing the concurrent software testing technique (**object of study**) for the purpose of evaluating (**purpose**) its effectiveness (**quality focus**) from the researchers perspective (**perspective**) in the context of test specialists finding errors in their concurrent software (**context**).

2.5 Final remarks

This chapter introduced some concepts related to this research. It addressed the importance of the test activity in the software development process and described the test phases and test activities that must be performed. The functional and structural testing technique was also discussed, and fundamental concepts of concurrent programming were provided.

The challenges faced in concurrent software testing were presented and the main errors that occur in this field were discussed. The chapter also highlighted the non-deterministic behavior in concurrent software testing for emphasizing its complexity. Finally, an overview of empirical strategies used as a survey, case studies and experiments were provided and the activities of the experiment process were detailed.

The next chapter presents a unified catalog of selection attributes developed towards guiding testers in the testing technique selection process.

CATALOG OF SELECTION ATTRIBUTES

This chapter describes a catalog of selection attributes and addresses their classification into 8 categories. The contributions offered include (i) design of a catalog of selection attributes available online for aiding consultation by testing practitioners, and (ii) possibility of collaboration to its improvement.

The chapter is organized as follows: Section 3.1 describes the methodology applied for the design of the catalog. Section 3.2 presents the catalog and highlights the features that can assist testers in selecting a testing technique to be applied to a software project. Section 3.3 discusses the classification of the categories.

3.1 Catalog design

Software testing practitioners must make decisions on techniques to be used in a specific situation and estimate the time and efforts necessary for their application. However, the literature reports no clear guidelines for such a task (VOS *et al.*, 2012). The choice of a testing technique to be adopted in a software project remains based on the knowledge of the test designer and often does not consider all techniques available in the industry or academia (VEGAS; JURISTO; BASILI, 2006). Its correct selection influences the quality of both process and product. Several testing techniques are available, and each of them has different and often complementary features that test several aspects of the software. Their choice should not be based only on subjective knowledge, but incorporate objective knowledge, guided by elements that favor good choices (VICTOR; UPADHYAY, 2011).

A mapping study conducted by Santos *et al.* (2019) identified studies on testing techniques selection and the attributes that support the selection process. The literature reports several approaches that aid such a selection (VEGAS; BASILI, 2005; VICTOR; UPADHYAY, 2011; MELO *et al.*, 2017; NETO; RODRIGUES; TRAVASSOS, 2011).

The motivation for this study was the construction of a catalog of selection attributes that aids software testing practitioners in the selection of the best testing techniques to be applied to software under test. Towards avoiding misinterpretations, term **attribute**, in this context, is related to any property of a software system/project/process that might influence the selection of testing techniques. Figure 8 shows the methodology applied for the production of the catalog.

Figure 8 – Methodology for the design of the catalog.

The mapping focused on the state-of-the-art strategies on testing techniques selection, and among the studies selected, 10 addressed the attributes that support the selection process (Table 4) as a result, we developed an interactive catalog of selection attributes available online¹, to making this knowledge more accessible to testers and other researchers. 178 attributes were extracted from the 10 studies selected, and a careful twofold analysis excluded duplicated ones. The first analysis led to the removal of 75 attributes with the same name, and revealed some attributes had the same purpose, despite different nomenclatures, which required a second round for the elimination of such duplicate attributes, thus resulting in 61 attributes.

In the analysis and classification phase, meetings were held for discussions and categorization of the attributes. Some approaches were developed for specific domains (e.g., web,

¹ Catalog of selection attributes website: <https://italo-07.github.io/catalog_attributes/index.html>

Table 4 – List of selected studies that compose the catalog

ID	Reference	Title	Public. Year
S01	Vegas and Basili (VEGAS; BASILI, 2005)	A Characterization Schema for Software Testing Techniques	2005
S02	Isabel and Travassos (ISABEL; TRAVASSOS, 2011)	Features of software testing techniques for use in projects web	2011
S03	Melo et al. (MELO et al., 2017)	How to Test Your Concurrent Software: An Approach for the Selection of Testing Techniques	2017
S04	Vegas (VEGAS, 2004)	Identifying the Relevant Information for Software Testing Technique Selection	2004
S05	Vegas et al. (VEGAS; JURISTO; BASILI, 2006)	Packaging experiences for improving testing technique selection	2006
S06	Neto and Travassos (NETO; TRAVASSOS, 2009)	Porantim: An Approach to Support the Combination and Selection of Model-Based Testing Techniques	2009
S07	Neto et al. (NETO; RODRIGUES; TRAVASSOS, 2011)	Porantim-Opt: Optimizing the Combined Selection of Model-based Testing Techniques	2011
S08	Victor and Upadhyay (VICTOR; UPADHYAY, 2011)	Selection of Software Testing Technique: A Multi Criteria Decision Making Approach	2011
S09	Neto and Travassos (DIAS-NETO; TRAVASSOS, 2014)	Supporting the Combined Selection of Model-Based Testing Techniques	2014
S10	Cotroneo et al. (COTRONEO; PIETRANTUONO; RUSSO, 2013b)	Testing techniques selection based on ODC fault types and software metrics	2013

concurrent, model-based, ODC faults) and others for the selection of techniques that use basic information and can be generalized for several testing techniques (e.g., general testing characteristics, software project, test team, tools).

3.2 Catalog of selection attributes

Based on the information available in the catalog, tester practitioners can identify selection attributes of the testing techniques that can support decision-making in the selection process. The tester can query all registered attributes and access their definitions (Figure 9). The attributes were classified into 8 categories to be searched by testers who, in collaboration with researchers, can update the catalog content filling a form with suggestions for future improvements. The studies used as references are described for those interested in reading the full paper, and some basic information on the authors who developed the catalog is also provided, so that interested users can contact them.

In the catalog initial interface, testers access the general information of the catalog (Figure 10) clicking on a menu on the left for navigating among the other pages (click 1). The purpose for its development is described and the identification of the categories into which the attributes were classified and the quantity of each category is described (click 2).

The catalog provides users with an interface for consulting the definition (click 1) of each attribute (Figure 11). This information (click 2) can be used for supporting the selection process.

The catalog was classified into the following 8 categories:

Figure 9 – Catalog of selection attributes - Features.

Figure 10 – Initial interface of the catalog.

- **General testing characteristics:** includes attributes of testing techniques, such as application cost and easiness of the understanding of the technique;
- **Tools:** contains information on the tools available, which supports the application of a testing technique;
- **Test team:** includes information on the test teams characteristics (e.g., experience and knowledge regarding the application of the technique);
- **Software project:** is composed of attributes related to information on earlier projects that applied the testing technique;

Figure 11 – Description interface of the catalog.

- **ODC faults:** contains attributes that identify orthogonal defect classification faults (e.g., incorrect or missing implementation of an algorithm, and errors in the interaction among components in the software interface) (COTRONEO; PIETRANTUONO; RUSSO, 2013b);
- **Web applications:** is composed of attributes used in the web application domain;
- **Concurrent software:** comprises attributes used in the concurrent software domain;
- **Model-based testing:** is composed of attributes used in model-based testing.

Users can consult the attributes of each category (Figure 12) through the left side menu (click 1) and search for some attributes in the search field (click 2). Concerning each category, the attributes present their respective definitions, as well as example values that they can assume and the reference of the studies that used them.

Figure 12 – Categories interface of the catalog.

If other users are interested in contributing (click 1) towards keeping the catalog updated (Figure 13), they can indicate other studies that describe selection attributes by filling in a form (click 2). The information will be sent to the authors, who will upload it in the catalog. The catalog code is available on GitHub², and enables open access to other users interested

² <https://github.com/italo-07/catalog_attributes>

in contributing. The suggestions are reviewed by the authors and incorporated in the original version.

Figure 13 – Suggest works interface of the catalog.

Testers can use the catalog for identifying attributes to be applied for the selection of a testing technique as follows:

1. The tester receives the documentation of a software project to be tested;
2. The tester extracts relevant information from the project documentation listed in the software project category (e.g., programming language, software quality characteristics, environment);
3. The tester extracts relevant information from the testing techniques already known, or from a new one, looking in the attributes listed in the general testing characteristics category (e.g., inputs and outputs required for the application of the technique, testing level at which the technique can be applied, and type of faults the technique can reveal);
4. The tester should consider some attributes related to the test team that can influence the choice of a testing technique (e.g., personnel who worked on earlier projects, experience required for the application of the technique, and efforts related to time and cost);
5. Based on the previous steps, the tester may have selected a testing technique to be applied in a software project. After relevant information about the project, general testing characteristics, and information on the test team have been outlined, the tester can use the attributes in the tool category to check if the technique can be applied with the support of some tool and analyze the costs of its use;
6. If the tester has received a specific software project with characteristics of web, concurrent, model-based or ODC faults, he can refer to those specific categories in the catalog;

7. The tester can identify other attributes that help the selection process of testing techniques not listed in the catalog, and therefore, share this knowledge suggesting the inclusion of attributes.

The catalog also provides information on testing tools that can support the testing technique selection. An example is SeleCTT, which is based on the characteristics of a software project. It suggests a list of best testing techniques and selects one in the context of concurrent software testing (MELO *et al.*, 2019b). The catalog is more general, and can support the construction of other tools similar to SeleCTT, for different application domains.

3.3 Discussion

This section details the classification of the selection attributes that compose the catalog.

General testing characteristics: this category comprehends selection attributes that can be generalized for all testing techniques, rather than specific characteristics of a particular technique. It is composed of 24 attributes (Table 5) used in either only one study, or up to eight distinct studies. They provide information on criteria to be considered for test case generation, number of test cases generated, possibility of automation, cost of application, existence of dependency relationships, information of possible problems with the use of the technique, and testing level at which the technique works. Types of faults identified by the technique were the most used attribute cited in eight papers, which has proven the importance of its adoption in the testing technique selection process.

Tools: this category comprises 4 attributes related to the identification of tools characteristics (e.g., identifier, support for its development and maintenance, costs of its use, and interest in the records of tools already used by the test team). Such factors must be considered in the choice of a testing technique (Table 6).

Test team: some attributes describe characteristics of the test team (Table 7). This category comprises 6 attributes and considers aspects, such as testers' knowledge for applying a testing technique, personnel involved in the testing process, experience required for the application of the technique, testers' opinions after its application, advantages and disadvantages, and efforts required.

Software project: this category is composed of 9 attributes (Table 8) with critical information of generic software projects (e.g., methodology applied for the project development, testing records of previous projects, programming language of the software, architecture, and quality characteristics considered by the project).

ODC faults: all attributes that compose this category (Table 9) were taken from Cotroneo *et al.* (COTRONEO; PIETRANTUONO; RUSSO, 2013b), who used orthogonal defect classification (ODC) faults, i.e., a method for process improvement based on defect analysis. The author's

Table 5 – General testing characteristics category

Reference	Attribute	Description	Example Values
S01, S02, S03, S04, S05, S08	Adequacy criterion	Test case generation and stopping rule	Execution trace, reachability testing, threading schedules - family (e.g., data flow, flow control) and technique (e.g., sentence coverage)
S01, S02, S04, S05, S08	Automation	Part of technique automated by the tool	(e.g., flow chart, mutant generation)
S01, S02, S04, S05	Benefits	Benefits of the technique	[Description of benefits]
S01, S02, S04	Completeness	Coverage provided by the test cases	Percentage
S01, S04, S05, S08	Comprehensibility	Ease of understanding the technique	(high, medium, low)
S01, S04, S05	Cost of application	Effort for the application of the technique	(high, medium, low)
S01, S04, S05	Dependencies	Relationships among techniques	Two values: [technique] and dependency type
S01, S04, S05	Effectiveness	Capability of a set of cases for detecting faults	Percentage
S01, S04, S05	Element	Elements of the system on which the test acts	(e.g., function, system, subsystem)
S02, S03	Experimental evaluation	Indication of the experimental strategy and evidence obtained in the evaluation	Case study, controlled experiment
S01, S02, S03, S04, S05	Inputs	Inputs required for the application of the technique	Code, requirements, models, bytecode
S01, S04, S05	Number of cases generated	Number of cases generated per software size unit	Formula
S03	Outputs	Information on outputs (artifacts)	Percentage of coverage, execution time, bugs revealed
S01, S04	Precision	Number of repeated test cases generated by the technique	Percentage
S01, S02, S04, S05, S08	Problems	Problems with the use of the technique	[Description of technique drawbacks]
S04	Purpose	Type of evaluation and quality attribute to be tested	Not informed in the original study
S08	Reusability of test cases	Indication of reusability of test cases	Not informed in the original study
S01, S04, S05	Sources of information	Location of information on the technique	(e.g., a person, a book, an article, an experiment)
S01, S04, S05	Test data cost	Cost of test data identification	(high, medium, low)
S02, S03, S06, S07, S09	Testing level	Test levels explored in the testing of the software behavior/structure	Unit, integration, system, regression
S08	Time at which functionality is exposed	Indication on the time at which the functionality is exposed	Not informed in the original study
S02, S03	Type of analysis	Indication on the type of test analysis supported by the approach: static or dynamic behavior	Static, dynamic
S01, S02, S03, S04, S05, S06, S08, S09	Type of faults identified	Type of faults revealed	Data race, atomicity-violation, deadlock, livelock - Navigational Error, Interface Error, Data type wrong
S02, S03, S06, S07, S09	Type of testing technique	Indication of the test technique supported by the approach	Structural, functional, fault-based, model-based

Table 6 – Tools category

Reference	Attribute	Description	Example Values
S01, S03, S04, S05	Cost	Cost of tool purchase and maintenance	Academic, open source - Values: (purchase cost) and (maintenance)
S01, S03, S04, S05	Identifier	Name of the tool and the manufacturer	Values: [tool name] and [company name]
S01, S03, S04, S05	Support	Support provided by the tool developer	(24-hour hotline, technical assistance)
S01, S03, S04, S05, S06, S07, S09	Tools used	Tools used in earlier projects	Tool name (e.g., Contest, CHESS, JavaPathFinder)

Table 7 – Test team category

Reference	Attribute	Description	Example Values
S02, S08	Effort required	Indication of the effort, time and cost of execution of the tasks	Not informed in the original study
S01, S04, S05, S07, S09	Experience	Experience required for using the technique	(tool knowledge, etc.)
S01, S04, S05	Knowledge	Knowledge required for using the technique	(e.g., flow charts, cyclomatic complexity)
S01, S04, S05, S08	Opinion	General opinion on the technique used	[Description of the opinion]
S01, S04, S05	Personnel	People who worked on earlier projects	[peoples' names]
S01, S04, S05	Repeatability	Possibility of two people generating the same test cases	(yes, no)

Table 8 – Software project category

Reference	Attribute	Description	Example Values
S01, S04, S05	Aspect	Functionality of the system to be tested	(communications, database, GUIs, etc.)
S01, S04	Development method	Development method or life cycle to which it is linked	(prototyping, reuse, waterfall etc.)
S01, S03, S04, S05, S06, S07, S09	Environment	Platform (sw and hw) and programming language which the tool operates	Linux, Windows, Android
S01, S03, S04, S05, S06, S07, S09	Programming language	Programming language used	Java, C/MPI, C/Phthreads, Fortran, Ada, Erlang
S01, S04, S05	Reference projects	Earlier projects in which the technique is used	[project name]
S01, S04, S05	Size	Size of the software	(number in KLOC)
S04	Software architecture	Development paradigm to which it is linked	Not informed in the original study
S02, S03, S06, S07, S09	Software quality characteristics	Indication of the set of quality characteristics evaluated by the approach	Effectiveness, performance, efficiency
S01, S04, S05	Software type	Type of software to be tested by the technique	(real time, batch, interactive, etc.)

choice for ODC was ultimately due to its orthogonality feature, closeness to actual mistakes made by programmers, and wide adoption in real large projects. [Cotroneo, Pietrantuono and Russo \(2013b\)](#) proposed a pragmatic method according to which testing engineers characterize some software under test in terms of estimated number of faults of different types it contains, and select the techniques most prone to detecting the highest number of faults of the various types for a given application. The attributes of this category comprehend aspects, such as possible errors in the software interface and faults in the software documentation, thus affecting the specification and design phases of the software development. The original study does not provide example values of such attributes.

Web applications: among the studies that compose the body of knowledge of the catalog, only one focused on the identification of selection attributes in web applications (Table 10). [Isabel and Travassos \(2011\)](#) presented a set of characteristics of web software testing techniques synthesized from a literature review and evaluated through a survey with specialists. It represents an initial result towards the organization of a characterization schema of web application testing techniques for supporting the choice of suitable software testing techniques for a web software project. Out of the 15 general attributes presented, only 2 are directly related to the category of web applications, and identify the type and unit of application of the approach (e.g., web pages, web page components, among others).

Table 9 – ODC faults category

Reference	Attribute	Description	Example Values
S10	Algorithm	Incorrect or missing implementation that can be fixed through the reimplementation of an algorithm or data structures	Not informed in the original study
S10	Assignment	Values incorrectly assigned or not assigned	Not informed in the original study
S10	Build/package/merge	Mistakes in library systems, management of changes, or version control	Not informed in the original study
S10	Checking	Missing data or their incorrect validation incorrect loop or conditional statements	Not informed in the original study
S10	Documentations	Problems in publication and maintenance notes and a significant effect in the early stages of the software life cycle (specification and high level design)	Not informed in the original study
S10	Function	Problems in a sizable amount of code and capability that is either incorrectly implemented, or not implemented	Not informed in the original study
S10	Interface	Errors in the interaction among components, modules, call statements, and parameters list	Not informed in the original study
S10	Timing/serialization	Missing or incorrect serialization of shared resources	Not informed in the original study

Table 10 – Web applications category

Reference	Attribute	Description	Example Values
S02	Application category	Information on the type of application of the approach	Rich Internet Application (RIAS), Web Services, Workflow, ubiquitous, e-commerce, e-learning
S02	Unity	Information on the unit of application of the approach, (e.g. web pages, web components, links, and others)	[optional text]

Concurrent software: Melo *et al.* (2017) (Table 11) proposed a testing technique selection approach that uses 21 attributes to aid the selection process. Six are specific of concurrent programs and describe the context and purpose of the software, mechanisms for different synchronization sequences, program representation, and the development paradigm used.

Table 11 – Concurrent software category

Reference	Attribute	Description	Example Values
S03	Context/Objective of the concurrent software	Description of the context/objective of the software	HPC, distributed applications, mobile, embedded
S03, S06, S09	Development paradigm	Indication of the development paradigm used	Shared memory, message passing, hybrid – Object oriented
S03	Mechanism that executes different synchronization sequences	Approach that forces the execution of different synchronization sequences	Reachability testing, noise injection
S03	Program representation	Strategy for the capture of relevant information to be tested	State space graph, petri nets, prediction model
S03	Technique that treats the state explosion problem	Indication of the technique that treats the problem of state explosion	Dynamic partial order reduction, fitness function, preemption bounding
S03	Type of replay mechanism	Indication of type of replay mechanism for the re-execution of the program	Monitoring, regression scheduler, record-replay, SYN-sequences

Model-based testing: the specific context of model-based testing (MBT) lacks scientific knowledge regarding already-developed MBT techniques, which hampers the transfer of such technologies from academic to industrial environments (NETO; TRAVASSOS, 2009). Regarding the MBT category, some studies have proposed approaches for the selection of testing techniques (Table 12). The category is composed of 2 attributes that specify the models used in the scientific literature and technologies that support test case modeling.

Table 12 – Model-based testing category

Reference	Attribute	Description	Example Values
S06, S07, S09	Behavioural/Structural Model provided	Indication of models proposed in scientific research	Use Case, Activity, Statechart, Classes Diagram, Collaboration Diagram
S06, S07, S09	Modelling Technology	Indication of the technology used for modelling	ALTS, ASM, CO-OPN/2, UML, XML, amongst others

Overview of attributes classification: The bubble chart in Figure 14 shows a summary of the attributes classification per category. Among the selected studies, the largest number of attributes used in the selection approaches are concentrated in the *general testing characteristics* and *software project* categories. The selection approach in study S08 suggests the combination of only those two categories to support their selection approach. The attributes extracted from study S10 were concentrated in only one category, since the authors used specific attributes of ODC faults in their proposal.

3.4 Final remarks

This chapter presented a catalog of 61 selection attributes used in approaches for testing techniques selection. The attributes are classified into 8 different categories, according to their characteristics. Most attributes used in the selection approaches are concentrated in the “general testing characteristics” category and, then, in “software project”.

The catalog is aimed at guiding testers on the selection of testing techniques to be applied to software projects in industry. Therefore, this research is expected to foster collaboration between industry and academia towards improving the quality of software developed from the choice of the best testing technique and its validation in industry cases. The results are also expected to help testers to identify attributes to be applied for the selection of a testing technique.

The next chapter presents SeleCTT-v2, the novel features implemented to support the combined selection approach proposed in this work. We describe how was characterized the software projects and the forward snowballing carried to update the body of knowledge of testing techniques used by SeleCTT.

Figure 14 – Distribution of attributes extracted from selected studies classified by categories.

SELECTT-V2

This chapter addresses some activities towards the evolution of SeleCTT tool. To facilitate identification, we will use the terminology SeleCTT-v1 (initial version of the tool) and SeleCTT-v2 (updated version of the tool). Section 4.1 provides an overview of SeleCTT including its development. Section 4.2 describes the new features implemented, to support testers making the combined selection of testing techniques.

Section 4.3 describes the characterization of 20 software projects providing a documentation that outlines relevant information for the selection process of a testing technique. It is divided into three categories, namely (i) general information, (ii) software features, and (iii) testing characteristics to be performed.

Section 4.4 focuses on the application of a forward snowballing for updating the body of knowledge of concurrent testing techniques. The update was essential for the inclusion of new information on concurrent testing techniques developed between 2016 and 2019.

4.1 Overview of SeleCTT-v1

SeleCTT tool¹ was developed for supporting decision-making on the selection of suitable testing techniques for a concurrent software project. It works as a cross-browser web page that uses information extracted from the body of knowledge proposed in Melo *et al.* (2019b), and calculates and provides the user with the best options of testing techniques, according to the characteristics of the software project under development. The aim is to offer both researcher and tester an automatic way of selecting testing techniques through a list of recommendations that can be used during the software testing activity. Figure 15 shows an overview of the process. The user submits information on the software project and, based on this information and analysis of the database of testing techniques (body of knowledge), the tool returns an ordered list of the

¹ <<http://www.labes.icmc.usp.br/~selectt/home>>

most appropriate techniques to be consulted by the user (tester).

Figure 15 – SeleCTT-v1 process overview.

SeleCTT is a web tool composed of four main modules, described below:

- **Insertion module:** receives information about the project to be developed as input. It offers two forms of insertion, i.e., one related to the information input of the attributes of a new concurrent software testing technique to be inserted into the body of knowledge - insertion appears as pending, and remains in the temporary body of knowledge for the approval of an administrator to be considered in the calculation of adequacy, and another associated with the input of information on the software project to be developed. The results of the most suitable testing techniques refer to the input information in the insertion module, since this information will be used in the selection algorithm to be cross-referenced with the attributes of the techniques present in the body of knowledge.
- **Administration module:** manages the body of knowledge content and can be accessed only by users of the "System Administrator" class. Among the activities it undertakes are (i) body of knowledge management, which controls the insertion of new techniques through the removal of those that do not provide adequate information, and edits new attributes, updating the existing ones and information of the techniques already registered, (ii) user management, which controls the insertion, editing, removal and approval of users, as well as permissions, and enables changes in the user's class, and (iii) management of the content of the tool, which edits the content present in other interfaces.
- **Control module:** controls the attributes and criteria adopted in the characterization of the testing techniques, and manages the weights for the selection of the most suitable testing

technique for a particular concurrent software project. The weights for each attribute are global in the tool, and their management in each of the twenty-one attributes consists in the update of the values for all techniques of the body of knowledge. Such weights were obtained in a survey conducted by [Melo et al. \(2019b\)](#), [Melo et al. \(2017\)](#), and are the basis for the calculation of the adequacy of the techniques to the software project.

- **Results module:** compares information extracted from the insertion module and uses adequacy criteria defined in the control module to recommend a list of techniques ranked according to their level of relevance to the project. The activity flow of the testing technique selection starts from the characterization of the project by the test team, and provides information about each characterization attribute defined. Information on the testing techniques of the body of knowledge is retrieved by SeleCTT and used in the selection process for the calculation of the adequacy through comparisons of the project attributes and those of the testing techniques. The techniques are then classified according to the score achieved in the adequacy of the specific criteria. The results generated by the tool are provided as an ordered list of relevance of each technique. The recommendations are offered to those interested in the selection, who evaluate them and select one to be used in their software project.

SeleCTT was developed with the following features: (i) **search for a specific technique:** the tool supports the visualization of techniques stored in the body of knowledge, which is useful for professionals searching for a specific technique, (ii) **insertion of a technique into the database:** a new concurrent software testing technique proposed by the academy which does not compose the database, or new secondary studies conducted at the academy and new testing techniques identified, can be inserted into the body of knowledge, (iii) **selection of testing techniques:** information about a developing project and that involves its attributes is provided by the user. The tool server receives the attributes in text form through the insert module, and the system compares them with those of the techniques of the body of knowledge, and (iv) **presentation of statistical data:** the tool provides a statistical prediction of its number of accesses and use to be stored for its possible evolution.

Furthermore, SeleCTT implements the characterization scheme proposed in [Melo \(2018\)](#), composed of 21 attributes (Table 13). Such attributes are used in the calculation of adequacy of the techniques to the test design, hence, selection of the most appropriate one.

In the selection process, information about a project under development is provided by the user concerning the attributes in Table 13. Once the tool server receives them, the system compares the project attributes with the attributes of the testing techniques in the body of knowledge, one by one, considering their respective weights (Table 13). Then, the textual values present in the attributes of the software project are compared with the n -techniques (n is one of the techniques stored in the body of knowledge) following the relation given by the adequacy

Table 13 – Attributes of concurrent software testing techniques implemented in SeleCTT (MELO, 2018)

Categories	Attribute	Example values	Weight
Programming model	1. Platform on which the software under test is executed	Linux, Windows, Android	6.15%
	2. Development paradigm adopted for process/thread interaction	Shared memory, message passing, hybrid	5.36%
	3. Programming Language/Runtime libraries in which the software under test was developed	Java, C/MPI, C/Pthreads, Fortran, Ada, Erlang	5.18%
General testing characteristics	4. Type of testing technique	Structural, functional, fault-based, model-based	6.68%
	5. Testing level at which the technique was applied	Unit, integration, system, acceptance, regression	6.24%
	6. Approach for the test data generation	Execution trace, reachability testing, threading schedules, location pairs	4.75%
	7. Input expected for a test case	Code, requirements, models, bytecode	5.80%
	8. Output expected for a test case	Percentage of coverage, execution time, bugs revealed	6.06%
	9. Software quality characteristic to be evaluated by the technique	Effectiveness, efficiency, performance, scalability, overhead	7.03%
	10. Type of empirical study applied for the validation of the testing technique	Case study, controlled experiment	0.00%
Concurrent testing characteristics	11. Type of analysis used by the technique	Static, dynamic	5.80%
	12. Mechanism that forces the execution of different synchronization sequences	Reachability testing, noise injection	6.24%
	13. Type of replay mechanism used for the re-execution of the program during testing	Monitoring, regression scheduler, record-replay, SYN-sequences	5.45%
	14. Program representation that captures relevant information for testing	State space graph, petri nets, prediction model	4.75%
	15. Context/Objective of the concurrent software	HPC, distributed applications, mobile, embedded	6.06%
	16. Technique that treated the state explosion problem	Dynamic partial order reduction, fitness function, preemption bounding	5.36%
	17. Type of concurrent bugs identified by the testing technique	Data race, deadlock, atomicity violation, livelock	7.03%
Tool support	18. Name that identifies the supporting tool	Contest, CHESS, JavaPathFinder	-
	19. Cost associated with the tool	Academic, shareware, open source	6.06%
	20. Programming Language/Runtime libraries supported by the tool	Java, C/MPI, C/Pthreads, C/OpenMP, Fortran, Ada, Erlang	-
	21. Execution platform on which the tool operates	Linux, Windows, Android	-

vector, in which each attribute is compared for each technique. This comparison calculation is based on the study of Dias-Neto and Travassos (2009b).

SeleCTT receives the information of the software project as a parameter and initialization of the weights and techniques. After initialization, for each technique stored, the attributes of technique n are compared with those of the project under development. Each weight is calculated by a comparison of the attributes, and the total weight is the sum of all attribute weights. Finally, the list is ordered with the most suitable techniques in the first positions, returning the result to be provided to the user.

The adequacy calculation equation consider the adequacy value ($v_{adequacao}(i,n)$), where i is the attribute and n is the technique, is calculated by comparing the characteristics present in the attribute. If the attribute has more than one characteristic (e.g., shared memory and message passing), only the similar ones enter the value. In this case, comparing the attribute of the project in development (e.g., shared memory) with the attribute of a body of knowledge technique (e.g., shared memory and message passing) result in 50% adequacy for this attribute. However, the

total adequacy is given by the sum of the adequacy vector multiplied by the respective weight.

This study has added novel features to SeleCTT towards a combined selection of testing techniques, evolving the scope of the tool proposed in [Melo et al. \(2019b\)](#). The next section discusses such features.

4.2 Combined selection

To improve the selection process for testing techniques carried out by SeleCTT-v1, this research proposed a way to help testers to apply the complementary testing techniques stored in the body of knowledge through a complementary choice of testing techniques towards a combined selection of concurrent software testing techniques. According to [Endres and Rombach \(2003\)](#), a combination of different testing techniques outperforms any single technique alone. Several testing techniques available display different and often complementary characteristics and, therefore, we developed new features to evolve SeleCTT to enable testers to benefit from a combined selection. Several testing techniques available display different and often complementary characteristics, which include cost of application, effectiveness in revealing defects, types of defects, among others.

The combined selection proposed in this research consists in grouping two selection attributes, namely *type of concurrent bugs* and *testing level*, present in SeleCTT, to show their correlation to users and support decision-making prioritizing this knowledge. Therefore, users can select testing techniques related to the type of defect in which the software project must be tested and the level of testing to which the technique suggested can be applied.

The adequacy calculation performed by SeleCTT-v1 has not been changed, since it considers the weight of the characterization attributes defined by a survey with specialists conducted by [Melo et al. \(2019b\)](#) and no new attributes have been included. Changes made in SeleCTT code to provide the user with the best options for a combined selection of testing techniques are briefly described. A vector called “features” was added to assist in the visualization of the techniques attributes and also in the grouping of “concurrent bugs”. After it had been ordered by the weight of the attributes in the original version, the following steps were implemented: (i) grouping the techniques by the *type of concurrent bugs* attribute after calculation of the adequacy; (ii) priority is given to *type of concurrent bugs* reported in the project inserted by the user at the beginning of the vector; and (iii) ordering the other groupings of the vector, based on the “type of concurrent bugs” of the techniques of highest adequacy according to the calculations.

SeleCTT-v2 combines the recommended techniques into groups of concurrent bugs, giving higher priority to the faults reported by the tester in the software project, as well as providing the user with information of the testing level at which the technique can be applied (Figure 16). This information is expected to guide the tester in choosing and applying a combination of more than one testing technique that can detect different faults.

Figure 16 – SeleCTT-v2 recommendation of results page.

The tester can also click on the plus icon next to the adequacy percentage of each technique (Figure 17). The information expands and shows a comparison among the attributes defined by the testing technique and those of the software project. The tester is expected to be provided with the possibility of analyzing the most efficiently applied testing technique according to the adequacy of the project attributes.

After the user has decided on techniques to be applied to a software project, he/she can select them checking on the checkbox next to the chosen technique (Figure 18) and save the result.

4.2.1 Novel features

Under this master's project, a scientific initiation project entitled *Evolution of the SeleCTT Tool to Support the Combined Selection of Concurrent Software Testing Techniques in the Industry* (FURLANETTI; SANTOS; SOUZA, 2020), was developed towards collaborating with the extension of SeleCTT-v2 features described in this section.

Feedback on the testing techniques used:

Another novel feature implemented in SeleCTT-v2 enables users to **give feedback on the testing techniques used in a software project** (Figure 19). After the tester applies the testing techniques recommended by SeleCTT-v2, he/she can provide feedback on the recommended

Concurrent Bug: Data race

Structural testing - context-sensitive sampling
dl.acm.org/citation.cfm?id=2336780

MATCH: 61.08%

Test Level: Unit testing

Concurrent Bug: Data race

Structural testing - context-sensitive sampling
dl.acm.org/citation.cfm?id=2336780

MATCH: 61.08%

Test Level: Unit testing

CARISMA: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications.

Attribute	Technique Description	Match	Project Description
Execution platform	Unix	✓	Unix
Context/Objective	HPC	✓	HPC
Programming language/Runtime libraries	Java	✓	Java
Testing technique	Structural testing	✓	Structural testing, Fault-based testing
Test data generation	context-sensitive sampling	✗	No information
Testing level	unit	✓	Unit
Synchronization interleaving mechanism	happens-before relation, locksets	✗	No information
Inputs required	multithreaded program	✓	Multithreaded program, source code in Java
	performance: detection rate, effectiveness:		

Figure 17 – Plus icon to compare information.

technique. This feedback will be available to other SeleCTT-v2 users, which can benefit from information on experiences of other testers in previously registered software projects.

The tester can also further update information in SeleCTT-v2 for providing feedback on the use of the recommended technique. This feedback is available for other SeleCTT-v2 users to access and benefit from information on experiences of other testers in previously registered software projects.

Body of knowledge of software projects:

After the user receives the suggestions of testing techniques to be applied, the information of the software project together with the testing techniques recommended by SeleCTT-v2 and chosen by the tester are stored in a body of knowledge of projects. This new feature is useful to generate a base of knowledge on recommendations and history by the company of techniques chosen in previous projects (Figure 20). It helps the sharing of information among testers searching for projects registered of some similarity with those to be tested.

The body of knowledge gathers important information for the documentation of the software project in SeleCTT-v2 (Figure 21), i.e., project name, its description, quality characteristics evaluated, concurrent bugs to be tested, level of testing at which the technique will be applied, type of testing technique, among others. The availability of this information is relevant for testers to observe project characteristics and to compare the results achieved through the

Concurrent Bug: Data race

Technique	MATCH	Test Level
Structural testing - context-sensitive sampling dl.acm.org/citation.cfm?id=2336780	61.08%	Unit testing
Structural testing - Thread interaction models https://dl.acm.org/doi/abs/10.1145/3208040.3208056	48.94%	Unit testing
Fault-based testing - data race detection dl.acm.org/citation.cfm?id=2734118	48.55%	Unit testing

Concurrent Bug: Deadlock

Technique	MATCH	Test Level
Fault-based testing - Mutation operators ieeexplore.ieee.org/document/4344126/	60.98%	Unit testing
Structural testing - Coverage criteria https://www.worldscientific.com/doi/abs/10.1142/S021819401850033X	54.67%	Unit testing
Structural testing - Coverage criteria https://ieeexplore.ieee.org/abstract/document/7423814	48.94%	Unit testing
Structural testing - coverage-driven approach dl.acm.org/citation.cfm?id=2384651	48.55%	Unit testing

Figure 18 – Selection of a testing technique.

Feedback

Please provide the information below.

Technique CARISMA: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications.

Strengths

Strengths of the technique

Weaknesses

Weaknesses of the technique

Difficulties

Difficulties in using the technique

Satisfaction

Satisfaction in using the technique

Figure 19 – SeleCTT-v2 feedback page.

testing techniques that have been recommended and applied.

Available Projects

Here you can view the available projects.

Legend

[View projects information](#)

Search:

Project name Show all	Inserted by Show all	Inserted on Show all	Expiration Date Show all	Actions
Account	user name	01/11/2019	30/11/0001	View
Airline	user name	15/11/2019	20/12/2019	View
Alarmclock	user name	20/12/2019	12/01/2020	View
Allocationvector	user name	10/10/2019	15/12/2019	View
BoundedBuffer	user name	10/09/2019	18/10/2019	View
Clean	user name	03/11/2019	07/12/2019	View
Daisy	user name	05/12/2019	10/01/2020	View
Deos	user name	03/11/2019	15/12/2019	View
DiningPhilosophers	user name	12/10/2019	22/11/2019	View
Linkedlist	user name	05/12/2019	20/01/2020	View

Showing 1 to 10 of 20 entries

Show 10 entries

First Previous 1 2 Next Last

Figure 20 – Overview of the body of knowledge of software projects.

Account

Title: Account

Inserted by: user name

Inserted on: 01/11/2019

Expiration: 30/11/0001

Execution platform: Unix

Objective: HPC

Programming language: Java

Testing technique: Structural testing, Fault-based testing

Test data generation: No information

Testing level: Unit

Figure 21 – Body of knowledge - project details.

4.3 Concurrent project characterization

The project characterization was necessary for the development of our research, since we aim to raise relevant information about software projects and created a body of knowledge incorporated in SeleCTT-v2. It will be available to users and can also serve as a support material for future empirical evaluations. Moreover, it is expected to enable users to provide feedback on the testing techniques recommended by SeleCTT-v2 and benefit other users, who can access this information, hence, improve the selection of such techniques. The projects documentation can be accessed in the case study laboratory package <https://sites.google.com/usp.br/case-study-package/>. Figure 22 shows the process that structures the information of the characterized software projects.

Figure 22 – Characterization of the projects.

Initially, we conducted a study with the concurrent testing subjects available on Software-artifact Infrastructure Repository (SIR) (DO; ELBAUM; ROTHERMEL, 2005), which is a collection of Java programs that exhibit the characteristics of multithreaded activities. They are useful for studies on the behavior of concurrent program designs and model checking analyses used by Dwyer, Person and Elbaum (2006), who studied factors that affect path-sensitive error detection techniques. Some concurrency subjects are suites of programs compiled by Bandera

(Dwyer *et al.*, 2006) and Java PathFinder (JPF) (Visser; Park; Penix, 2000) projects. Such programs cover nearly all the multi-threaded programs used in the evaluation of path-sensitive Java analyses from the literature (Do; Elbaum; Rothermel, 2005). Some examples also contain variants in a collection of multi-threaded Java programs developed at IBM to support testing and analysis research (Eytani *et al.*, 2007). This set of programs overlaps Bandera and JPF programs to some extent, but it also includes several programs developed for encoding common Java concurrency bug patterns.

SIR provided information for the project characterization process, such as project name, description, programming language and concurrent bug. However, it was complemented towards a robust software project document for the selection process of testing techniques. The body of knowledge implemented in SeleCTT-v2 was used for this purpose and, after the update activity, 124 concurrent software testing techniques guided the development of the project towards meeting the requirements of specific testing techniques registered.

Despite the characterization of 20 software projects, an oracle defined describes the testing technique to be suggested by SeleCTT-v2 according to the information in each project. Its definition was based on a consensus among 3 researchers to ensure that the techniques suggested were the most relevant. The information was then inserted into the body of knowledge.

Table 14 shows a template of the characterized project, divided into three categories, namely (i) general information, (ii) software features, and (iii) tests characteristics to be performed. The attributes are defined in the template. Appendix A describes a software project characterized and four options of concurrent testing techniques to be applied.

Appendix B shows the description of all 20 characterized projects. It describes characteristics for different platforms, such as Linux, Windows, and Unix, and contexts, e.g., HPC applications and distributed systems; all of them present message-passing and shared-memory paradigms. Java is the programming language used. Some projects must be tested at unit and system levels and use structural, fault-based, and model-based testing techniques. The projects show concurrent bugs related to (i) assertion violation, (ii) atomicity violation, (iii) bug-triggering interleavings, (iv) data race, (v) deadlock, (vi) high-level concurrency errors, (vii) null-pointer dereferences, (viii) refinement violations, and (ix) starvation, besides other relevant information.

4.4 Forward snowballing

The initial version of the body of knowledge of the concurrent testing techniques contains 109 primary studies on concurrent testing techniques identified through a systematic mapping conducted by Melo *et al.* (2019a), that selected only concurrent testing techniques that have been evaluated empirically. The studies that compose the body of knowledge were published between 1997 and 2016. Our research required its update, therefore, information on new concurrent testing techniques developed up to (2019) was added following the forward snowballing methodology.

Table 14 – Template of project characterization

General Information	
Attribute	Definition
ID	Project identification
Project Name	Registered project name
Description	Description of a software project
Start date	Start date of the project
End date	End date of the project
Software features	
Development Environment	Execution platform on which the software under development runs
Type of software	Context in which the software under development will be applied
Programming language	Language in which the software is developed
Concurrent paradigm	Paradigm used in the interaction between processes/threads
Techniques suggested	Techniques selected to test the software project
Project status	Status that identifies whether the project is active or inactive
Tests characteristics to be performed	
Test level	Test level at which the technique is applied
Testing technique	Type of testing technique to be applied to the project
Type of analysis	Type of analysis performed by the testing technique
Quality characteristics	Characteristic of the software used in the evaluation of the technique
Types of faults	Concurrent bugs identified by the testing technique.
Support testing tools	Identification of a testing support tool
Inputs	Inputs expected by the testing technique
Outputs	Outputs generated by the testing technique
Program representation	Representation that captures information relevant to the test

The next subsections address the update of the body of knowledge of concurrent testing techniques conducted by forward snowballing.

4.4.1 *Research question*

The following research question was formulated to be answered during forward snowballing: **RQ - 1** - What new concurrent testing techniques were developed and empirically evaluated from 2016 to 2019? The aim is the identification of concurrent testing techniques that provide empirical evidence of their applicability.

4.4.2 *Inclusion and exclusion criteria*

The definition of Inclusion Criteria (IC) and Exclusion Criteria (EC) contributes to the inclusion of primary studies relevant to the research question, and exclusion of those that do not

answer it. The selection procedure used in the forward snowballing was the same defined in the mapping study conducted by Melo *et al.* (2019a) (Table 15).

Table 15 – Forward snowballing inclusion and exclusion criteria (MELO *et al.*, 2019a)

Inclusion criteria
IC1 - Studies that propose testing techniques for concurrent software and apply empirical studies for their evaluation.
Exclusion criteria
EC1 - Studies that do not propose concurrent software testing techniques, studies that propose software testing techniques not related to concurrent applications, and studies that propose techniques only for hardware testing;
EC2 - Studies that propose concurrent program testing with no empirical evaluation;
EC3 - Studies whose authors claim they include an empirical evaluation, however the process is not described in the paper;
EC4 - Studies not written in English and studies not accessible in full-text.

4.4.3 Identification of seed set

The first step for the application of forward snowballing is the identification of the initial set of studies, known as a seed set. According to Felizardo *et al.* (2016), a seed set can be composed of primary studies included in a previous version of an SLR and in the SLR itself, and enables the identification of citations for each of such studies. Towards identifying the citations in our forward snowballing, we searched in the Google Scholar² electronic database. Among the electronic databases (e.g., IEEE Xplore, Scopus, ScienceDirect) that provide information on citations of papers, since, according to Felizardo *et al.* (2018), the use of a generic database is sufficient for the discovery of most studies.

The initial seed set contained 109 studies from the mapping conducted by Melo *et al.* (2019a), which was also considered, thus totaling 110 studies. Appendix C shows the studies that compose the seed set and their number of citations when the search was conducted. The last two columns show the total number of citations from these studies until the moment at which the research had been conducted, whereas the last column displays the number of citations of each study published between 2016 and 2019. In this study, we are interested in the number of citations described in the last column, where we considered the number of citations corresponding to these four years. The total number of papers was 2799; duplicated ones were removed (1681 papers), thus resulting in 1118 papers to be analyzed in the first interaction of forward snowballing, whose overview is outlined in Figure 23.

4.4.4 Iterations

Our forward snowballing followed the guidelines proposed by Wohlin (2014), i.e., (i) each new paper that cites the study in our seed set is examined, (ii) the first screening is based on the information provided by Google Scholar, and (iii) if the information is insufficient for

² <https://scholar.google.com/>

Figure 23 – Forward snowballing overview.

a decision, the citing paper is studied in more detail. First, the abstract is read, and if this is insufficient, the full text is read for the decision on its inclusion.

Three iterations were necessary in the search process, however, the third returned no new study. Table 16 shows an overview of the results. The first iteration started with the 1118 primary studies retrieved from the seed set. Each paper was analyzed according to the inclusion and exclusion criteria defined, thus resulting in the inclusion of 13 studies for the second phase. In this first iteration, a large number of studies were removed, since they did not present any new concurrent testing technique with an empirical evaluation. Among the 13 studies selected in iteration 1, the total number of papers that cited it was 36 new studies, we remove 12 duplicated studies remaining 24 new studies, for the analysis process. Two new studies were then included for iteration 3. They reported 7 citations, of which 2 were duplicates, and 5 were left for analysis. The 5 studies were analyzed, but not included. The search was interrupted in iteration 3, since no study had been included in this iteration.

Table 16 – Iteration forward snowballing results

	Iteration 1	Iteration 2	Iteration 3	Total
Studies	1118	36	7	1161
Included	13	2	0	15
Excluded	1105	22	5	1132
Duplicated	0	12	2	14

Several studies recognize forward snowballing as an efficient search technique for the update of secondary studies (WOHLIN, 2014; FELIZARDO *et al.*, 2016; FELIZARDO *et al.*, 2018). An important measure for systematic literature studies is efficiency concerning number

of included papers out of the total number of papers reviewed (WOHLIN, 2014), conducted as follows:

- Seed set: 2799 papers and 1118 papers included - Efficiency = $1118/2799 = 39.94\%$;
- Iteration 1: 1118 papers from snowballing with other 13 papers included - Efficiency = $13/1118 = 1.16\%$;
- Iteration 2: 36 papers from iteration 1, and 2 included - Efficiency = $2/36 = 5.5\%$;
- Iteration 3: 7 papers reviewed and no paper included - Efficiency = 0% ;
- Overall efficiency: $(1118+13+2+0) / (2799+1118+36+7) = 1133 / 3960 = 28.61\%$.

4.4.5 Overview of results

Table 17 shows a list of the selected studies. Regarding their distribution per year, 2 were published in 2016, 4 in 2017, 2 in 2018 and 7 in 2019.

4.4.6 Analysis and discussion

This section provides the results to our research question according to the studies found by forwarding snowballing.

RQ-1 - What new concurrent testing techniques were developed and empirically evaluated from 2016 to 2019?

This research question aims at the identification of new concurrent testing techniques developed between 2016 and 2019 which provide empirical evidence of their applicability. Such new information can be added to the body of knowledge of concurrent testing techniques in SeleCTT-v2. Tables 18 and 19 show a summary of information on the studies.

Table 18 displays information on the execution platform where the testing technique runs, the context for which it was developed, the programming language implemented, the type of testing technique and level at which it can be used, and the quality attributes empirically evaluated.

Regarding execution platforms, six studies (S01, S04, S06, S08, S14 and S15) (NGUYEN *et al.*, 2017a; WANG *et al.*, 2019b; XUE; SIAMI-NAMINI; NAMIN, 2018; YU; ZUO; XIONG, 2019; WANG *et al.*, 2019a; NGUYEN *et al.*, 2017b) used Linux. S03, S07 and S09 (LI *et al.*, 2016; CAI; LU, 2016; PROTZE *et al.*, 2018) used variations of the Unix system, whereas others used the windows system (S02, S04, S05, S06, S10, S11 and S12) (LIN *et al.*, 2017; WANG *et al.*, 2019b; DING; ZHOU, 2019; XUE; SIAMI-NAMINI; NAMIN, 2018; QI; ZHOU, 2019; XU; LEI; CARVER, 2017; LI *et al.*, 2019). S13 (ANTONINO; GIBSON-ROBINSON; ROSCOE, 2019) did not specify it.

Table 17 – List of studies selected

ID	Title	Public. Year
S01	Concurrent program verification with lazy sequentialization and interval analysis	2017
S02	SPDebugger: A Fine-Grained Deterministic Debugger for Concurrency Code	2017
S03	Effectively manifesting concurrency bugs in android apps	2016
S04	DFTracker: detecting double-fetch bugs by multi-taint parallel tracking	2019
S05	RaceTest: harmful data race detection based on testing technology in WS-BPEL	2019
S06	Testing Multi-Threaded Applications Using Answer Set Programming	2018
S07	Dynamic testing for deadlocks via constraints	2016
S08	Concurrency Bug Avoiding Based on Optimized Software Transactional Memory	2019
S09	Thread-local concurrency: a technique to handle data race detection at programming model abstraction	2018
S10	A Splitting Strategy for Testing Concurrent Programs	2019
S11	Using Delta Debugging to Minimize Stress Tests for Concurrent Data Structures	2017
S12	Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing	2019
S13	Efficient verification of concurrent systems using synchronisation analysis and SAT/SMT solving	2019
S14	AVPredictor: Comprehensive prediction and detection of atomicity violations	2019
S15	Parallel bug-finding in concurrent programs via reduced interleaving instances	2017

Most selected studies present testing techniques for high performance computing (HPC) applications. S03 ([LI et al., 2016](#)) differs from the others, since it was developed for a mobile application context; it shows concurrency bugs in Android apps based on the generation of input events and their schedules for an app. The authors conducted a static-dynamic hybrid analysis to find potentially conflicting resource accesses in an app, which was automatically pressure-tested by guided event and schedule generation. They implemented an AATT tool and evaluated it over thirteen popular real-world open-source apps. It successfully found 9 concurrency bugs out of which 7 were previously unknown.

Among programming languages, Java was the most used (S02, S05, S06, S07, S10, and S11), followed by variations in C/C++/C# (S01, S04, S05, S07, S08, S12, S14, and S15). Since S03 was proposed for mobile applications, the language used was Android. MPI language was employed in S09 ([PROTZE et al., 2018](#)), which proposed a novel technique that accurately detects data races that can occur at higher levels of a concurrent execution. The idea is to introduce the

general concept of Thread-Local Concurrency (TLC) as a new way of translating the concurrency expressed by a high-level programming paradigm into a low execution level understood by the tools. The author's evaluation revealed the technique can detect races accurately with only small additional analysis overheads.

Regarding type of testing technique, most of the studies introduced new testing techniques for structural ones - two for functional technique (S03 and S08), and one for the technique based on defects (S10). S10 (QI; ZHOU, 2019) describes a prototype tool, called SplitRichTest, which adopts the framework of conventional reachability testing while implementing a splitting strategy. Moreover, it generates and exercises fine-grained synchronization sequences and uses an efficient heuristic algorithm to select and sort candidate splitting points. S10 was conducted on representative concurrent Java programs and evaluated the effectiveness of SplitRichTest using mutation testing. The results show the splitting strategy facilitates the generation of more fine-grained synchronization sequences and significantly improves the concurrency fault detection capability of the reachability testing.

Among the testing levels identified, most techniques are applied at the unit level - only two studies presented techniques for the system level (S03 and S08). Convoider tool, introduced in S08 (YU; ZUO; XIONG, 2019), solves problems of software transactional memory related to high human costs for equipping programs with transaction functionality and low compatibility with I/O calls and conditional variables. It automatically transactionalizes target programs with no source code modification and recompiling. The experimental results showed it succeeded in a transparent transaction of twelve real-world applications incurring only 28% runtime overhead and perfectly avoided 94% of thirty-one concurrency bugs reported.

The last column of Table 18 shows the quality attributes and quality characteristics of the software used in the evaluation of the testing technique. Most studies used efficiency and effectiveness as quality attributes. Other quality attributes that appeared were cost, execution time, scalability and performance (S02, S03, S04, and S10). S02 (LIN *et al.*, 2017) proposed a novel approach, called SPDebugger, which provides finer-grained thread controlling over test cases, programs under test, and even a third-party library code for the reproduction of a predesigned thread execution schedule. The evaluation of effectiveness and cost revealed SPDebugger handles more debugging scenarios than the state-of-the-art tools with similar human efforts.

Table 19 provides information on the type of testing analysis conducted, the concurrent paradigm related to it, the concurrent bugs that can be found, and on whether it is supported by a tool for the automation of the process.

The studies addressed the use of static analysis (S01, S02, S03, S04, S06, S10, S13, and S15) and dynamic one (S03, S05, S06, S07, S08, S09, S11, S12, and S14). Static analysis examines a source code without running the program, whereas dynamic analysis requires the execution of the code. S13 (ANTONINO; GIBSON-ROBINSON; ROSCOE, 2019) introduced a static testing analysis, and the synchronization-analysis techniques aim to reveal whether

Table 18 – Information of studies - Part I

ID	Execution platform	Context/ Objective	Programming Language	Testing technique	Testing level	Quality attributes
S01	Linux	HPC	C	Structural	Unit	Effectiveness
S02	Wndows	HPC	Java	Structural	Unit	Effectiveness, Cost
S03	Unix	Mobile	Android	Functional	System	Effectiveness, Execution time
S04	Linux, Windows	HPC	C, C++	Structural	Unit	Effectiveness, Efficiency, Scalability
S05	Wndows	HPC	Java, C++	Structural	Unit	Effectiveness, Efficiency
S06	Linux, Windows	HPC	Java	Structural	Unit	Effectiveness, Efficiency
S07	Unix	HPC	Java, C, C++	Structural	Unit	Effectiveness
S08	Linux	HPC	C, C++	Functional	System	Effectiveness, Efficiency
S09	Unix	HPC	MPI	Structural	Unit	Effectiveness
S10	Windows	HPC	Java	Fault-based	Unit	Effectiveness, Performance
S11	Windows	HPC	Java	Structural	Unit	Effectiveness, Efficiency
S12	Windows	HPC	.Net, C#	Structural	Unit	Effectiveness, Efficiency
S13	-	HPC	-	Structural	Unit	Effectiveness
S14	Linux	HPC	C, C++	Structural	Unit	Effectiveness, Efficiency
S15	Linux	HPC	C	Structural	Unit	Effectiveness

a system state is either unreachable, demonstrating components cannot agree on the order in which they participate in system rules, or unreachable, showing components cannot agree on the number of times they participate in system rules. Such automatic techniques check deadlock and local-deadlock freedom in the PairStatic framework.

S05 (DING; ZHOU, 2019) proposed a technique called RaceTest, for the identification of harmful data races through dynamic detection. It obtains potential data races with static detection, prunes false positives statically with constraint-solving, and generates test cases containing feasible potential races. Finally, it executes test cases towards identifying harmful data races according to a deterministic technique that actively controls thread interleaving. The experimental results showed its effectiveness in reducing several false positives and confirming harmful data races with high accuracy.

S07 (CAI; LU, 2016) designed a technique called ConLock, which analyzes each cycle and its corresponding execution for identifying a set of scheduling constraints necessary for

the triggering of the corresponding deadlock. It performs a second run to enforce the set of constraints that will trigger a deadlock if the cycle is a real one. For each confirmed deadlock, ConLock conducts a static analysis to identify conflicting memory accesses that would also contribute to the occurrence of the deadlock. The analysis enables developers to understand and fix deadlocks. The results of the empirical evaluation showed across all 811 cycles, ConLock confirmed all 16 deadlocks at almost 80%, and for the remaining cycles, it reported steering failures and identified five deadlocks involved in conflicting memory accesses.

Concerning the support tools described, most studies report some testing tools. S01 and S15 used Lazy-CSeq, an approach for sequential consistent concurrent programs that use Pthreads API (NGUYEN *et al.*, 2017a). It can be applied to other concurrency models, as long as a new design of the corresponding lazy sequentialization has been defined. The authors implemented it in the last release of Lazy-CSeq and demonstrated its effectiveness in substantially improving performance for very hard verification problems. S15 (NGUYEN *et al.*, 2017b) presented an approach that does not depend on the tool chosen for the final analysis. It is compatible with weak memory models, and amplifies the effectiveness of tools, since they can find bugs faster and with fewer resources. The authors used Lazy-CSeq as an off-the-shelf final verifier to demonstrate the approach with a small number of cores can find bugs in the hardest known concurrency benchmarks in minutes, whereas other dynamic and static tools fail to do it in hours.

Regarding the new concurrent testing techniques, we observed some details considering the classification presented in the work of Melo *et al.* (2019a), where the new techniques extend the contributions as follows:

- **Structural testing:** our analysis of the studies has led to the following conclusions:
 - S01 uses abstract interpretation to minimize the representation of the concurrent program's (shared global and thread-local) state variables (NGUYEN *et al.*, 2017a);
 - S02 implemented a debugging tool - SPDebugger, which combines a user-defined XML specification and JDI-based (Java Debugging Interface) framework to manipulate thread interleavings without altering the source code (LIN *et al.*, 2017);
 - S04 proposed an approach, called multi-taint parallel tracking to detect double-fetch bugs, that is caused by the race condition between kernel and user space. This study introduces multiple-taint parallel tracking to double-fetch bug detection, which maps the time factor of memory access ordering to the space factor of different tainted variables specific to double-fetch bug features (WANG *et al.*, 2019b);
 - S05 proposed a novel technique that detects harmful data races dynamically based on testing technology in web service-business process execution language (WS-BPEL). It is an effective strategy of test case generation that helps dynamic detection and exposes some semantic race errors caused by incorrect outputs (DING; ZHOU, 2019);

Table 19 – Information of studies - Part II

ID	Testing analysis	Concurrent paradigm	Concurrent bugs	Support tool
S01	Static	Shared memory	Bug-triggering interleavings	Lazy-CSeq
S02	Static	Shared memory	Interleavings	ASM
S03	Dynamic, Static	Shared memory	Data race, atomicity violation	AATT prototype
S04	Static	Shared memory	Data race, double-fetch bugs	-
S05	Dynamic	Shared memory	Interleavings, data race	-
S06	Dynamic, Static	Shared memory	Interleavings, data race, atomicity violation	-
S07	Dynamic	Shared memory	Deadlock	-
S08	Dynamic	Shared memory	Deadlock, data race, atomicity violation	Convoider
S09	Dynamic	Shared memory	Data race	ThreadSanitizer
S10	Static	Message passing	Mutation based errors	SplitRichTest
S11	Dynamic	Shared memory	Bugs inserted by the researcher	TestMinimizer
S12	Dynamic	Shared memory	Thread-safety violations, data race	TSVD
S13	Static	Message passing	Deadlock	DeadIOx
S14	Dynamic	Shared memory	Atomicity-violation, bug-triggering interleavings	AVPredictor
S15	Static	Shared memory	Bug-triggering interleavings	Lazy-CSeq

- S06 introduced an answer set programming (ASP) as a powerful tool for constraint-based test case generation problems, which formally expresses and deterministically exposes race conditions. It also implements adequacy criteria based on threads interleaving coverage (XUE; SIAMI-NAMINI; NAMIN, 2018);
- S07 designed a dynamic technique, called ConLock, which is a constraint-based approach that manipulates program scheduling for confirming deadlocks and handling false positives. Given a predictive run and a set of cycles, the technique analyzes the former and then generates a set of constraints for each cycle (CAI; LU, 2016);
- S09 introduced the concept of thread-local concurrency (TLC) to reason about concurrent execution units within individual operating system threads, illustrating the way concurrency is expressed in different programming paradigms for TLC (PROTZE *et al.*, 2018);
- S11 developed an approach that minimizes stress tests for concurrent data structures and creates a smaller test that produces the same failure by removing some of the threads and/or method invocations from original test (XU; LEI; CARVER, 2017);
- S12 describes a novel technique that uses a lightweight monitoring of the calling

behaviors of thread-unsafe methods, rather than synchronization operations, to dynamically identify bug suspects. A concurrency testing tool was built as part of the integrated build and test environments that have become popular in software companies (LI *et al.*, 2019);

- S13 describes a static testing analysis and investigated the way approximations can formally check scalable concurrent systems, proposing a synchronization analysis for the automatic capture of global invariants and approximating reachability (ANTONINO; GIBSON-ROBINSON; ROSCOE, 2019);
 - S14 designed a prediction-based approach that predicts unmanifested atomicity-violation bugs from a non-buggy execution and comprehensively displays all buggy interleavings. It uses a monitored execution to record execution traces and predict potential buggy interleavings based on the candidate interleavings identified from the trace and controlling executions and verify the predicted buggy interleavings by controlling the thread-scheduling (WANG *et al.*, 2019a); and
 - S15 presents a swarm approach for finding bugs in concurrent programs. It performs code- to-code translation that constructs program variants by placing tiles over the threads, and thus reducing nondeterminism by enabling the occurrence of context-switches only in a selected subset of tiles and inhibiting statement reordering in other selected ones (NGUYEN *et al.*, 2017b).
- **Functional testing:** researchers have developed two new techniques to test the system level - one for android applications - S03 (LI *et al.*, 2016), and another for HPC applications - S08 (YU; ZUO; XIONG, 2019).
 - S03 proposed a technique that simultaneously generates events and schedule in a mobile app, thus yielding a two-phase approach comprised of a pre-processing phase, which finds each event handlers relevant resource access through a hybrid static-dynamic analysis, and a manifestation phase, which proactively generates potentially concurrency-bug-related events, as well as their schedules for further checking;
 - S08 presents a new software transactional memory system, called Convoider, which implements a technique for achieving easier/safer concurrency and avoiding concurrency bugs. It aims at transparently transactionalizing applications with no manual effort, avoiding several types of concurrency bugs, and simultaneously supporting revocable I/O and a proper cover handling. The software transactional memory is a software-level concurrency control mechanism through which programmers can partition codes into transactions and ensure they are executed atomically and in isolation. Convoider makes deadlocks, data races, and impossible atomicity violations.

- **Fault-based testing:** researchers have employed mutation testing to conduct an empirical study on representative concurrent Java programs and evaluate the effectiveness of SplitRichTest, which is a tool proposed by S10 (QI; ZHOU, 2019).

4.4.7 Threats to validity

The results from this study can potentially be affected by the systematic protocol defined, and their threats are discussed below:

1. *Conclusion validity:* regards the reliability in finding studies relevant to the topic addressed in our research (we were concerned about finding new papers from the seed set defined). Towards mitigating it, Google Scholar extracted the paper's citations, therefore, we believe no important paper was missed.
2. *Internal validity:* towards dealing with this threat, we followed the guidelines outlined in Wohlin (2014) for applying forward snowballing.
3. *External validity:* this threat concerns with our ability to generalize the results of this study. The forward snowballing updated the mapping performed by Melo *et al.* (2019a). Towards a consistent result from the studies selected, the results were validated through a peer-review conducted by other authors.

4.5 Final remarks

This chapter has provided an overview of SeleCTT (v1 and v2) for contextualizing the way the tool was developed and how SeleCTT can be evolved through the development of SeleCTT-v2 with several modifications for supporting the combined selection of testing techniques. Among the new features implemented (Figure 24) are a combination of two attributes, namely (i) type of concurrent bugs, and (ii) testing level, and the way SeleCTT-v2 arranges the recommended techniques into groups of concurrent bugs, giving higher priority to faults reported by the tester in the software project, and providing the user with information of the testing level at which the technique can be applied. The chapter also describes the development of the body of knowledge of software projects and the feature that enables users to give feedback about the recommended testing techniques.

The process was outlined for characterizing software projects, and a study conducted among the concurrent testing subjects available in the SIR provided some relevant information for such a task. This information was complemented towards a robust software project document for the selection of testing techniques. 20 software projects were characterized, and an oracle defined describes the testing technique to be suggested by SeleCTT. The definition of the oracle was based on a consensus among 3 researchers for ensuring the techniques were the most relevant ones. The information was then inserted into the body of knowledge.

Figure 24 – SeleCTT-v2 process overview.

Another important activity was the application of a forward snowballing to the mapping conducted by [Melo et al. \(2019a\)](#), towards updating the body of knowledge incorporated in SeleCTT-v2. The initial version of the body of knowledge contained information on 109 primary studies published only until 2016. After the update, studies published between 2016 and 2019 and 15 new concurrent testing techniques were included, thus totaling 124 concurrent testing techniques. We analyzed and discussed the studies towards answering the following research question: *What new concurrent testing techniques were developed and empirically evaluated from 2016 to 2019?*

All activities conducted and described in this chapter were fundamental for the development of this research. The features proposed for the evolution of SeleCTT-v2 support testers in a combined selection of testing techniques. The body of knowledge must be update, so that SeleCTT-v2 also remains update and indicates the most recent testing techniques. The projects characterized are important for the experiments and can be used as a supporting material for other interested users.

COMBINED SELECTION EVALUATION

This chapter reports on a case study conducted for the evaluation of the combined selection approach. Section 5.1 addresses concepts relevant to the case study design, the motivation, goals and research questions to be investigated. Section 5.2 describes the preparation and the instruments required for the study, which was divided into two different scenarios with four activities towards expanding our observations on the testing technique selection process.

Section 5.3 outlines a quantitative analysis of the two scenarios and we discuss the results. Section 5.4 is devoted to the threats to validity.

5.1 Case study design

Empirical studies in software engineering and their contributions to knowledge expansion have continuously grown (RUNESON; HÖST, 2009). According to Runeson *et al.* (2012), a case study in software engineering is an empirical inquiry that draws on multiple sources of evidence for investigating an instance (or a small number of instances) of a contemporary software engineering phenomenon within its real-life context, especially when the boundary between phenomenon and context cannot be specified. The next subsections provide information on a case study that evaluated the combined selection approach proposed in this research.

5.1.1 *Rationale for the study*

According to Runeson *et al.* (2012), the researcher should be clear about the reasons for undertaking a case study. In academic research, a common reason for starting a case study is to make a novel contribution to the body of knowledge on a specific subject. The main purpose of this study is to assess whether the combined selection approach proposed and implemented in SeleCTT can recommend better concurrent testing techniques in comparison to its previous version, and assist testers in choosing better ways to combine testing techniques.

5.1.2 Study goals

The template described in Basili and Rombach (1988) was used for the definition of the case study goals. We conducted a case study to evaluate the combined selection approach (SeleCTT-v2) with the previous version of the SeleCTT-v1. It aimed to *Analyze* the suggestions of concurrent software testing techniques produced by SeleCTT-v2 against those provided by SeleCTT-v1, *for the purpose of comparison, with respect to their effectiveness, from the point of view of the researchers, in the context of the choice of a set of software testing techniques to validate a concurrent software project.*

A combined selection refers to a tester choosing two or more concurrent testing techniques from the options recommended by SeleCTT-v2 to complement different characteristics in a concurrent software project during the testing activity.

5.1.3 Research questions

Towards answering our research questions, we divided the observations into two scenarios. The first, i.e., *SeleCTT evaluation*, will enable conclusions related to RQ1, and *Scenario 2 - Manual selection of testing techniques by testers*, will provide data to answer RQ2. The following research questions were established:

RQ1. Does the use of the combined selection approach of SeleCTT-v2 imply more effective recommendations of testing techniques in comparison to the recommendation made by SeleCTT-v1?

We observed the way SeleCTT (v1 and v2) recommended testing techniques for 20 concurrent software projects for answering RQ1. The projects described in the previous chapter, were extracted from a benchmark repository composed of concurrent software code and information about concurrent bugs¹. Additional information was defined for complementing data on the project, including the most relevant testing techniques for each project. A correct testing technique combination is the one defined in the oracle for each concurrent software project. The oracle was defined by the research group, which considered the distinct characteristics of the project for determining the correct testing technique.

We analyzed the behavior for the 20 concurrent software projects that were inserted in SeleCTT (v1 and v2) to collect the results regarding the recommendation of testing techniques made for each project. The effectiveness measure was applied for the evaluation of the results (FRANKL; WEISS, 1993). Effectiveness (E) is a measure of equivalence between the recommendation made by the tools and the one expected (oracle). It is calculated by $E = CR/TR$, where *CR* represents the total number of correct recommendations made by SeleCTT (concerning the oracle), and *TR* denotes the total number of expected recommendations.

¹ <https://sir.csc.ncsu.edu/content/sir.php>

RQ2. How do testers make the combined selection of testing techniques for concurrent software projects?

This question was designed for our understanding of the way testers perform a combined selection of testing techniques for concurrent software projects. In this scenario, testers simulated how they would choose testing techniques for projects, using a set of testing techniques as support. We aimed at identifying the effectiveness of the recommendations of testers to compare with the results of RQ1.

5.2 Data collection

5.2.1 Instruments

The case study was divided into two different scenarios towards expanding our observation on the testing technique selection process. Its laboratory package is available at: <https://sites.google.com/usp.br/case-study-package/>. Each scenario required the use of the following instruments:

- **Scenario 1 - SeleCTT evaluation:**

- (i) documentation of concurrent software projects, which consists of specific characteristics of a software project to be used for the comparison with the attributes of the testing techniques and should be considered in the recommendation;
- (ii) definition of the testing oracle, necessary for the evaluation of the response expected for each project;
- (iii) development of a script for the inclusion of the software projects for ensuring the same information had been included in SeleCTT (v1 and v2);
- (iv) records of recommended testing techniques collected through spreadsheets exported by SeleCTT as a result.

- **Scenario 2 - Manual selection of testing techniques by testers:**

- (i) a form for characterizing the participants, i.e., their academic degree and experience in software engineering, concurrent programming and testing of concurrent software;
- (ii) documentation with characteristics of both projects and techniques for guiding the participants in the recommendation requested during the tasks;
- (iii) records of the recommendation for each project, which included a form with the recommendations and their responses;
- (iv) a feedback form for the collection of participants' impressions on the recommendation task.

5.2.2 Procedures

The study has two scenarios with four activities each illustrated in Figure 25.

Figure 25 – Overview of case study activities.

Scenario 1 - SeleCTT evaluation:

Activity 1: Characterization of software projects

20 concurrent software projects introducing programs with concurrent bugs were taken from a repository of benchmarks proposed and evaluated by several studies were used (DO; ELBAUM; ROTHERMEL, 2005). The author defined the features for the characterization of each project at SeleCTT. Table 20 provides some information on the projects characterized.

Activity 2: Oracle definition

The oracle presents two techniques expected as a return for each concurrent software project and which identify the best combination suggested by SeleCTT. The oracle was reviewed and analyzed by two specialists in the field of concurrent software testing. Table 21 shows the software testing techniques selected and considered in the oracle. A letter (ID) is associated with each of them and will be used in the analysis of the results. Table 22 displays the techniques, which are part of the defined oracle.

Activity 3: Scripts development

Scripts were developed to include the concurrent projects information in SeleCTT (v1 and v2). The scripts used to insert the projects in SeleCTT (v1 and v2) can be accessed in the case study laboratory package <<https://sites.google.com/usp.br/case-study-package/>>.

Activity 4: Records of suggested testing techniques

This activity consists of the extraction of spreadsheets with the testing techniques

Table 20 – Overview of characterized software projects

ID	Project name	Concurrent bug types
P1	Account	Deadlock, Data race
P2	Airline	Data race, Atomicity violation
P3	Alarmclock	Null-pointer dereferences, Starvation
P4	Allocationvector	Bug-triggering interleavings, Deadlock
P5	BoundedBuffer	Deadlock, Atomicity violation
P6	Clean	Refinement violations, Deadlock
P7	Daisy	Assertion violation, Atomicity violation
P8	Deos	Assertion violation, Deadlock
P9	DiningPhilosophers	Deadlock, High-level concurrency errors
P10	LoseNotify	Deadlock, Data race
P11	Linkedlist	Atomicity violation, Null-pointer dereferences
P12	Logging event	Deadlock, Null-pointer dereferences, Atomicity violation
P13	Nested Monitor	Deadlock, Starvation
P14	Piper	Deadlock, Data race
P15	Producerconsumer	Data race, Assertion violation
P16	Raxextended	Data race, Null-pointer dereferences
P17	Readerswriters	Deadlock, Data race
P18	Reorder	Atomicity violation, Starvation
P19	Replicatedworkers	Deadlock, Assertion violation
P20	SleepingBarber	Deadlock, Bug-triggering interleavings

recommended by SeleCTT for each concurrent software project. The spreadsheets with the raw results of SeleCTT (v1 and v2) can be accessed in the case study laboratory package <https://sites.google.com/usp.br/case-study-package/>.

Scenario 2 - Manual selection of testing techniques by testers:

Activity 1: Participants fill the characterization form

In this scenario, we have 3 testers, we will call them participants using the term (P1, P2 and P3). We select participants with knowledge on software testing, concurrent programming, and testing of concurrent programs. A form (Appendix D), was designed and applied for characterizing their knowledge on the topics of the research.

Activity 2: Instructions and documentation for the task

The participants were instructed on the combined selection task and also accessed a supporting material with documentation of the projects and the testing techniques similar to the documentation in Appendix A, where the suggested techniques expected were omitted and their order was mixed. They were asked to carefully read the material for basing their selection.

Activity 3: Combined selection task

Having information for each concurrent software project, the participants were asked to recommend the most suitable technique to be applied in a complementary way. The time for

Table 21 – Software testing techniques for concurrent programs selected for our case study

ID	Software testing technique for concurrent programs
A	A Lightweight and Portable Approach to Making Concurrent Failures Reproducible
B	An Automation-Assisted Empirical Study on Lock Usage for Concurrent Programs
C	An Evaluation of Data Race Detectors Using Bug Repositories
D	ASN: A Dynamic Barrier-Based Approach to Confirmation of Deadlocks from Warnings for Large-Scale Multithreaded Programs
E	BALLERINA: automatic generation and clustering of efficient random unit tests for multithreaded code
F	CARISMA: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications
G	Clash of the Titans: Tools and techniques for hunting bugs in concurrent programs
H	Comparative assessment of testing and model checking using program mutation
I	Concurrency Debugging with Differential Schedule Projections
J	Coverage metrics for saturation-based and search-based testing of concurrent software
K	Distributed reachability testing of concurrent programs
L	Effective random testing of concurrent programs
M	Efficient Execution Path Exploration for Detecting Races in Concurrent Programs
N	Efficient mutation testing of multithreaded code
O	Generating effective tests for concurrent programs via AI automated planning techniques
P	Generating unit tests for concurrent classes
Q	How Good is Static Analysis at Finding Concurrency Bugs?
R	Incremental testing of concurrent programs using value schedules
S	Location pairs: a test coverage metric for shared-memory concurrent programs
T	Multithreaded Test Synthesis for Deadlock Detection
U	MuTMuT: Efficient Exploration for Mutation Testing of Multithreaded Code
V	On the Effectiveness of Contracts as Test Oracles in the Detection and Diagnosis of Functional Faults in Concurrent Object-Oriented Software
W	Selective mutation testing for concurrent code
X	Testing Multi-Threaded Applications Using Answer Set Programming

the development of the task was not measured, since it may vary according to the amount of information described in the project, or the participant's experience.

Activity 4: Participants fill the feedback form

In the last activity of scenario 2, the participants were asked to fill in a feedback form (Appendix E), about the combined selection task.

5.2.3 Data analysis

The quantitative analysis in scenario 1 compared the effectiveness of SeleCTT (v1 and v2). In scenario 2, we observed how 3 testers made recommendations to compare their effectiveness, and analyzed the data from both characterization and feedback forms.

Table 22 – Oracle of testing techniques

ID	Project name	Technique 1	Technique 2
P1	Account	F	H
P2	Airline	S	I
P3	Alarmclock	O	Q
P4	Allocationvector	E	D
P5	BoundedBuffer	A	B
P6	Clean	S	U
P7	Daisy	L	G
P8	Deos	R	L
P9	DiningPhilosophers	P	S
P10	LoseNotify	U	M
P11	Linkedlist	C	O
P12	Logging event	R	O
P13	Nested Monitor	X	Q
P14	Piper	V	K
P15	Producerconsumer	L	O
P16	Raxextended	O	P
P17	Readerswriters	A	S
P18	Reorder	A	Q
P19	Replicatedworkers	D	H
P20	SleepingBarber	E	N

5.3 Results and interpretation

5.3.1 Scenario 1: Quantitative analysis

The performance of SeleCTT (v1 and v2) in recommending the combination of testing techniques was observed. Our proposal consists in the grouping of two selection attributes (e.g., concurrent bugs and testing level). In SeleCTT-v2, we considered a correct result its returning the two expected techniques (the combination) in first place for a concurrent bug detailed in the concurrent software project. Towards a comparison with SeleCTT-v1 which only returns an ordered list of testing techniques for each project, we considered a correct result its returning the same first two testing techniques defined in the oracle.

Table 23 shows the results of effectiveness for the 20 concurrent projects. Regarding SeleCTT (v1 and v2), in the combined selection approach proposed, the effectiveness showed a 10-point improvement in comparison to SeleCTT-v1, and achieved 95%. Therefore, SeleCTT-v2 provided better recommendations; it identified 19 correct recommendations (CR) and recommended 1 wrongly combination compared to SeleCTT-v1, which identified 17 CR and 3 incorrect combinations. The high effectiveness values (close to 1) indicate SeleCTT-v2 was able to recommend a representative set of CR for concurrent software projects.

Table 24 shows an overview of the results of the effectiveness of SeleCTT (v1 and v2)

Table 23 – Results from SeleCTT evaluation

Tool	Project ID	CR	TR	Effectiveness
SeleCTT-v1	P1 + P2 + ... + P20	17	20	0.85
SeleCTT-v2	P1 + P2 + ... + P20	19	20	0.95

in recommending techniques for each concurrent software project. Below is an analysis of the incorrect recommendations by SeleCTT (v1 and v2).

SeleCTT-v1 wrongly recommended the combination of *MuTMuT: Efficient Exploration for Mutation Testing of Multithreaded Code* and *Multithreaded Test Synthesis for Deadlock Detection* for P6 SeleCTT-v1, which is not the most suitable, according to the following aspects: (i) the two techniques are applied for fault-based testing and P6 specifies it must be tested for both fault-based and structural testing; (ii) SeleCTT-v1 does not consider one of the tools specified in the project, i.e., JavaPathfinder, and the program representation specified in the project that uses control flow graph; and (iii) the combination does not test the project for one of the concurrent bugs specified in the project (e.g, refinement violations). Such aspects highlight the combination provided by SeleCTT-v1 will cause the tester to apply an incorrect combination of testing techniques that does not cover all characteristics of the concurrent project and might demand costs for the software under test. The ideal combination would be *MuTMuT: Efficient Exploration for Mutation Testing of Multithreaded Code* and *Location pairs: a test coverage metric for shared-memory concurrent programs*, since such techniques complement each other, thus guaranteeing the testing of all characteristics defined in P6 missed by the incorrect combination made by SeleCTT-v1.

Concerning the results of SeleCTT-v1 for P13, the combination of the techniques recommended is *Testing Multi-Threaded Applications Using Answer Set Programming* and *Multithreaded Test Synthesis for Deadlock Detection*, which did not consider the following characteristics of the concurrent software project: (i) the output provided was not specified in the project (e.g., number of bugs detected); (ii) the two techniques can test programs related to the concurrent paradigm of shared memory, however, the project requires a technique that also tests the message-passing paradigm; (iii) the program representation of this combination considers only the data flow representation, and the project specifies the use of call graph and lock graph; (iv) none of the techniques can be used with the FindBugs tool defined in the project; and (v) the combination does not guarantee after testing, the project will be free from the concurrent bug - starvation defined in the design specification. Therefore, the ideal combination would be *Testing Multi-Threaded Applications Using Answer Set Programming* and *How Good is Static Analysis at Finding Concurrency Bugs?*, which can test all the characteristics defined in P13.

SeleCTT-v1 also made an incorrect recommendation for P18, suggesting the combination of *A Lightweight and Portable Approach to Making Concurrent Failures Reproducible* and *Location pairs: a test coverage metric for shared-memory concurrent programs*, which is not the

Table 24 – Effectiveness values of SeleCTT recommendations (v1 and v2) for software projects

Project ID	Effectiveness	
	SeleCTT-v1	SeleCTT-v2
P1	1.00	1.00
P2	1.00	1.00
P3	1.00	1.00
P4	1.00	1.00
P5	1.00	1.00
P6	0.00	1.00
P7	1.00	1.00
P8	1.00	1.00
P9	1.00	1.00
P10	1.00	1.00
P11	1.00	1.00
P12	1.00	1.00
P13	0.00	1.00
P14	1.00	1.00
P15	1.00	1.00
P16	1.00	1.00
P17	1.00	1.00
P18	0.00	1.00
P19	1.00	0.00
P20	1.00	1.00
Mean	0.85	0.95
Std Dev	0.37	0.22

most suitable, since (i) such techniques cannot test P18 for the concurrent bug specified (e.g., starvation) - they are more appropriate for the finding of bugs related to high-level concurrency errors, atomicity violation, refinement violations, data race, and deadlock; and (ii) none of the techniques can be used with the tool specified in the project (e.g., FindBugs). The most suitable combination would be *A Lightweight and Portable Approach to Making Concurrent Failures Reproducible* and *How Good is Static Analysis at Finding Concurrency Bugs?*, since they ensure the project will be free of the bugs specified or more appropriate to the testing of P18, and use the tools known by the testers.

SeleCTT-v2 made an incorrect recommendation for P19, because we considered a correct result its returning the two expected techniques (the combination) in first place for a concurrent bug detailed in the concurrent software project. SeleCTT-v2 recommended the same technique (e.g., *Comparative assessment of testing and model checking using program mutation*) for the two concurrent bugs specified (e.g., deadlock, assertion violation), which we consider an incorrect combination, since it may cause the tester to apply the same technique and miss some p19 characteristic, such as (i) the project specifies it is within the context of HPC and distributed systems, thus requiring the testing considers the two contexts; (ii) the input defined in the project

ignored the multithreaded program characteristic; (iii) the analysis type considered only dynamic analysis, whereas the project requires aspects of statistical analysis be considered; and (iv) the concurrent paradigm does not consider the message-passing specified in P19. The ideal combination would be *ASN: A Dynamic Barrier-Based Approach to Confirmation of Deadlocks from Warnings for Large-Scale Multithreaded Programs* and *Comparative assessment of testing and model checking using program mutation*, and SeleCTT-v2 recommends only one of them.

We also calculate the confidence interval (CI) for the collection of results of SeleCTT (v1 and v2) based on [Grissom and Kim \(2005\)](#). CI is usually interpreted as the range of values that encompasses the population or ‘true’ value, estimated by a certain statistics, with a given probability ([NAKAGAWA; CUTHILL, 2007](#)). In our context, if it were replicated a large number of times, roughly 95% of the 95% CIs calculated from the samples would be expected to include the true value of the population parameter of interest. Therefore, we can deduce the value of the population parameter lies within this envelope (5% chance of wrongness). The CI results indicate a 95% probability of SeleCTT-v1 recommending a correct combination, i.e., between 0.81 and 0.89, with a 0.04 margin of error. In comparison with SeleCTT-v2 and considering a 0.02 margin of error, the CI results indicate 95% probability of its recommending a correct combination between 0.93 and 0.97 (Figure 26).

Figure 26 – Confidence interval for SeleCTT (v1 and v2) results.

Therefore, RQ1, i.e., *Does the use of the combined selection approach of SeleCTT-v2 imply more effective recommendations of testing techniques in comparison to the recommendation made by SeleCTT-v1?* has been answered- effectiveness is higher in SeleCTT-v2, i.e., the use of the combined selection approach implies more effectiveness in recommend a testing techniques.

5.3.2 Scenario 2: Quantitative analysis

In scenario 2, we observed 3 testers performing the combined selection task towards understanding their procedure and identifying the effectiveness of their recommendations. We asked the participants to fill in a form for characterizing their knowledge, which revealed a level of education above a master's degree and a medium level in practice with concurrent software testing in the industry. Figure 27 shows the results from the experience with the research topics related to the study (e.g., software engineering, software testing, concurrent programming, concurrent software testing, and testing techniques selection).

The participants' experience was measured according to the following values: (1) none; (2) studied in class or in a book; (3) practiced in classroom projects; (4) practiced in a project in the industry; and (5) practiced in several projects in the industry. P1 and P2 have similar knowledge and P3 has higher knowledge in the five research topics.

Figure 27 – Characterization of the participants.

Towards evaluating the participants' effectiveness in the combined selection of testing techniques, we asked them to order the techniques according to their relevance for each software project. They were provided with information on 4 options of testing techniques for the task. For analysis purposes, the first two techniques ordered by the participants were considered a form of combined selection for application in the concurrent software project.

The calculation of the effectiveness was similar to the previous scenario ($E = CR/TR$), where CR represents the total number of correct recommendations made by participants (concerning the oracle) and TR denotes the total number of expected recommendations. The following values were considered: (0) both techniques recommended are incorrect, (0.50) at least one technique of the combined selection is correct, and (1) both techniques recommended are correct.

Table 25 shows the results for the 20 concurrent projects. The highest value was 80%, obtained by P3, which demonstrates P3's higher level of experience in the research topics inves-

tigated may have influenced the results. P1 and P2 had approximate results for effectiveness, i.e., 58% and 50%, respectively. In terms of effectiveness, we can state P3 made better recommendations, with 16 CR and 4 wrong combinations. P2 and P1 made between 10 and 11 CR and at least 10 wrong combinations. High values of effectiveness (close to 1) indicate the capacity of testers to perform well in the selection of testing techniques for concurrent software projects.

Table 25 – Results of participants' recommendations

Participants	Project ID	CR	TR	Effectiveness
P1	P1 + P2 + ... + P20	11,5	20	0.58
P2	P1 + P2 + ... + P20	10	20	0.50
P3	P1 + P2 + ... + P20	16	20	0.80

Table 26 shows an overview of the participants' results concerning recommendations for each concurrent software project. Incorrect recommendations will be analyzed.

Table 26 – Effectiveness values of participants' recommendations for software projects

Effectiveness			
Project ID	P1	P2	P3
P1	0.50	0.50	1.00
P2	0.50	0.00	1.00
P3	0.50	0.50	1.00
P4	0.50	0.50	0.50
P5	0.50	0.50	1.00
P6	1.00	0.50	1.00
P7	0.50	0.50	1.00
P8	0.50	1.00	0.50
P9	0.50	0.50	0.50
P10	1.00	0.50	1.00
P11	0.50	0.50	1.00
P12	0.50	1.00	0.50
P13	0.50	0.50	0.50
P14	1.00	0.50	1.00
P15	0.00	0.00	0.00
P16	1.00	0.50	1.00
P17	0.50	0.00	1.00
P18	0.50	1.00	1.00
P19	0.50	0.50	0.50
P20	0.50	0.50	1.00
Mean	0.58	0.50	0.80
Std Dev	0.24	0.28	0.30

P2 showed the worst performance, i.e., 3 incorrect suggestions for P2, P15, and P17, since they did not consider all the characteristics specified for those projects. P2 incorrectly recommended the following combination for P17: *An Automation-Assisted Empirical Study on Lock Usage for Concurrent Programs* and *CARISMA: a context-sensitive approach to race-*

condition sample- instance selection for multithreaded applications. It is not the most suitable for testing P17, since it misses some characteristics specified in the concurrent project, such as (i) the execution platform of those techniques do not run in windows, which is the platform of P17 development; (ii) the input required for is not the same available in P17; (iii) the techniques evaluate only effectiveness, and P17 requires the evaluation of other quality characteristics (e.g., performance, reproducibility, efficiency, scalability); and (iv) the tool support that supports the two techniques cannot be automated by the tools described in P17. Moreover, P2 made 14 partially CR, ensuring at least one technique of the combination was appropriate for the software project. P2 made 3 correct suggestions for P8, P12, and P18, since they guarantee the tester will apply two testing techniques that are complementary and encompass all the specified characteristics of the concurrent software project described.

P1 made 15 partially correct, 4 CR (P6, P10, P14, and P16), and only 1 incorrect recommendation (P15): *Efficient Execution Path Exploration for Detecting Races in Concurrent Programs* and *CARISMA: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications*. The combination is not suitable, since (i) the two techniques test the software with a structural technique, which is not the type defined in the project (e.g., model-based and formal method-based testing); (ii) the quality characteristic to be evaluated is only effectiveness, which is not enough according to the quality characteristics in P15 (e.g., performance, prediction capability); (iii) it cannot test P15 for finding the concurrent bug assertion violation specified in P15; and (iv) the supporting tool used is not specified in the project.

P3 showed the best performance, i.e., 6 partially correct, 13 CR, and 1 incorrect recommendation for P15, similar to P2. The ideal combination would be *Effective random testing of concurrent programs* and *Generating effective tests for concurrent programs via AI automated planning techniques*, since the techniques can complement themselves when applied together, ensuring the fulfillment of all characteristics defined in P15, i.e., (i) both techniques test different types of techniques and meet the requirements specified in the project; (ii) the quality characteristics evaluated must encompass all characteristics specified in P15; (iii) the techniques can find the concurrent bugs that validate P15 – this characteristic ensures the software is free of such defects; and (iv) they can be automated by CalFuzzer and Penelope tools, specified in the project.

Concerning the participants' opinions given in the feedback form. They were asked if they had missed information in the selection process towards our checking the suitability of the documentation and instructions for the combined selection task. Only P1 reported "*The technique performance, despite having a report describing the technique, not all of them I could imagine the real application*", which suggests the participant missed a real example of use of the technique for facilitating the selection. Although it was an interesting observation, in a real environment, the tester does not often have experience with other testing techniques and/or a real example of application of a new one. We must highlight the need for a tool that supports decision-making by indicating the most suitable technique to be applied to a software project.

The participants were also asked about their uncertainties for a correct choice of the technique. P2 reported some difficulty in choosing the most interesting parameters to be considered: “*Which parameters to take into account when selecting techniques*”. We investigated the problem, which is recurring in the selection of testing techniques. The way it was identified by the participant should be highlighted, since the context of our study is similar to a real context in the industry.

We also asked the participants to explain the way they based their selection. P1 reported the choice was based on attributes, such as development paradigm, type of analysis, concurrent bug and testing level. P2, on the other hand, reported the selection was based on the correlation among available techniques and type of testing to be performed. P3 considered the testing tools that automate the technique and the concurrent bugs focused by each technique.

Therefore, we can answer RQ2, i.e., *How do testers make the combined selection of testing techniques for concurrent software projects?*, i.e., testers take into account different aspects, e.g., development paradigm, type of analysis, concurrent bug, testing level, type of testing and testing tools, which highlights their importance for the combined selection approach in SeleCTT-v2. The tool considers the selection attributes of concurrent bugs and testing level for suggesting the best combination of testing techniques for a concurrent software project. Moreover, considering the effectiveness of the testers where they can recommend the most correct combinations of testing techniques to a concurrent software project (the highest value was achieved by P3 (80%), and the lowest was obtained by P2 (50%)), concluding that the effectiveness ranged between 50% and 80% among testers, in the next section we compare the results gathered between scenarios.

5.3.3 Comparison between Scenarios

This section is devoted to a comparison between the results of the two scenarios observed. When performing the combined selection task, testers achieved good results with a 50% effectiveness rate in the worst case, and 80% in the best case. However, the results were obtained individually and according to only the testers’ knowledge and practical experiences with software testing. If they could access a tool that supports the selection process, the effectiveness of the results would have been 95%, as evidenced by the results of SeleCTT-v2. SeleCTT can also help inexperienced testers to make the best decision, thus avoiding poor choices of testing techniques regarding the finding of important defects in the developed software product.

Scenario 2 identified the way testers based their decisions and the importance of some selection attributes for guiding a selection activity (e.g., development paradigm, type of analysis, concurrent bug, testing level, type of testing, and testing tools). Therefore, the contributions of this research regarding the use of attributes, such as *concurrent bug* and *testing level* provide a better combination of testing techniques according to the characteristics of a software project.

5.4 Threats to validity

The potential threats are discussed towards guiding the interpretation of the developed study (RUNESON *et al.*, 2012; WOHLIN *et al.*, 2012):

1) *Construct validity*: a possible threat to the case study plan might be the results of the testing techniques defined in the oracle having a bias from the researcher who defined it. Towards mitigating this threat, the oracle was reviewed by two other researchers, and would be modified in case of disagreements.

2) *Internal validity*: it concerns the way factors not considered can affect the variables and relationships investigated, therefore, our study investigated two scenarios. In the first, SeleCTT (v1 and v2) were compared for the choice of the most effective approach for recommending combined testing techniques. The previous version recommends only a list of techniques for a software project. The first and second techniques indicated for comparison with those suggested in the new approach were considered correct for data analysis. In scenario 2, we observed how testers made the combined selection for contrasting the results with the performance of SeleCTT. The knowledge and experience of the participants were assumed representative of testers who work with testing of concurrent programs.

3) *External validity*: it refers to conditions that limit the generalization of the results to industrial practice. The limitation on the number of projects used in the case study can be considered a threat to validity. Towards mitigating it, we characterized a representative context of concurrent software projects with a wide variety of concurrent bugs. The number of testers who participated in the study can be considered a threat, to mitigate it participants were selected with an adequate level of knowledge in the topics covered in this study. Moreover, the set of techniques chosen is representative of the population as a whole, since the body of knowledge used has been updated with new concurrent testing techniques available at SeleCTT.

4) *Conclusion validity*: it concerns issues that affect the drawing of correct conclusions about the study. Therefore, we used different instruments to collect quantitative and qualitative data, which enabled analyses of evidence from the recommendations made by SeleCTT (v1 and v2) in scenario 1 and those made by testers in scenario 2. Observations made by the researchers during the study, as well as the characterization and feedback form filled by the participants were considered and contributed to the results.

5.5 Final remarks

This chapter reported on a case study that assessed whether the combined selection approach proposed and implemented in SeleCTT-v2 can recommend better concurrent testing techniques than its previous version, and whether it can support testers in choosing better ways for combining testing techniques.

Two research questions were elaborated from the observations in the two scenarios proposed, and we calculated the effectiveness of the recommendations made by SeleCTT (v1 and v2) in scenario 1, and those made by the testers in scenario 2. The instruments used in the case study, as well as the activities performed were described.

Based on the results from SeleCTT-v2, we can conclude the combined selection approach improves the effectiveness of recommendations of concurrent testing techniques, which demonstrates SeleCTT-v2 is less likely to make an incorrect recommendation of combined techniques for a software project, thus avoiding costs associated with the incorrect application of a testing technique. If the testers could access a tool that supports the selection process, the effectiveness of the results would have reached a higher value, as evidenced by our approach. Our future aim is to conduct research with testers using SeleCTT for measure their effectiveness.

The next chapter provides the results, emphasizing the contributions made, the limitations and the possibility of exploring other directions in future studies.

CONCLUSIONS

This chapter summarizes the research developed in this master project. Section 6.1 highlights the main results, and Section 6.2 addresses the contributions of this work. Section 6.3 discusses some limitations faced by the research, and Section 6.4 outlines some suggestions for future work towards improvements and advances in the state-of-the-art of testing techniques selection. Finally, Section 6.5 displays the publications achieved and studies under development.

6.1 Results

Among the results of this research, we can highlight:

- Identification of the state-of-the-art of approaches for testing techniques selection towards the extraction of information on their evaluation and attributes that support them;
- Design and conduction of a survey that gathered responses from 185 practitioners of software industries that work directly with software testing. The aim was the identification of the software testing practices in Brazilian industries, towards knowledge of such practices and an overview of the latest testing techniques, selection processes, challenges faced, and tools and metrics used;
- Identification and categorization of attributes used in testing technique selection approaches and search for new ones in concurrent software testing for aggregating the selection approach proposed in SeleCTT-v2. However, no new attributes were found;
- Design of a catalog of selection attributes for assisting testing practitioners in choosing the most suitable testing techniques to be applied to a particular software project. The catalog was classified into 8 categories, namely (i) general testing characteristics, (ii) tools, (iii) test team, (iv) software project, (v) ODC faults, (vi) web applications, (vii) concurrent software, and (viii) model-based testing;

- Conduction of a forward snowballing for updating the body of knowledge of concurrent testing techniques in SeleCTT-v2. The strategy included 15 new studies between 2016 and 2019. The update of the body of knowledge is crucial for users to access the latest information on concurrent testing techniques;
- Definition of an approach that supports the combined selection of concurrent testing techniques towards improving the suggestions of testing techniques made by SeleCTT-v2. It prioritizes two selection attributes to guide the suggestion, namely concurrent bugs and testing level;
- Expansion of this master's project through the establishment of a scientific initiation project entitled *Evolution of the SeleCTT Tool to Support the Combined Selection of Concurrent Software Testing Techniques in the Industry*;
- Design of a body of knowledge of projects included in SeleCTT-v2, so that users can access and view those registered and use the information as a basis for decision-making on the choice of a testing technique;
- Inclusion of a feature in SeleCTT-v2 through which users provide feedback on the testing technique suggested. The feedback can also be viewed by other users of the tool, which contributes to the sharing of experiences and supports the selection process;
- Characterization of software projects through the documentation of concurrent software programs available in the SIR repository¹. It can be used by other researchers in empirical studies, or as a supporting material for the selection of testing techniques in concurrent software;
- Design and definition of a case study conducted to evaluate the combined selection approach proposed in our research. The results obtained suggest that the combined selection approach (SeleCTT-v2) implies more effectiveness in performing the recommendation of concurrent testing techniques.

6.2 Revisiting the contributions

Among the main contributions of this research are:

- Characterization of the testing technique selection approaches proposed in the literature;
- Recognition of software testing practices applied in Brazilian software industries;
- Design of a catalog of selection attributes for guiding testers in the selection of testing techniques;

¹ <https://sir.csc.ncsu.edu/content/sir.php>

- Definition of a combined selection approach that suggests testing techniques to be used to improve the testing process;
- Development of a body of knowledge of software projects for maintaining the records of stored projects and supporting the tester in decisions on the testing technique to be applied;
- Facilitation of knowledge sharing among testers through their feedback when using the testing technique suggested by SeleCTT-v2, that is also available online at <http://www.labes.icmc.usp.br/~selectt/home>;
- Update of the body of knowledge of concurrent testing techniques with the inclusion of new studies published until 2019;
- Characterization of twenty software projects through robust documentation that describe specific characteristics of concurrent software projects based on programs used in empirical studies;
- Education of human resources in the field of testing techniques selection considering the development and co-orientation of a scientific initiation project and an undergrad student in his conclusion course work related to this master's research.
- Support testers in the task of testing techniques selection, offering automated resources to combine testing techniques that allow testers to explore the complementary aspects of concurrent testing techniques.

6.3 Limitations

Below are the challenges and limitations faced during the development of the research:

- The contribution of the catalog was not initially evaluated;
- The update of the body of knowledge of concurrent testing techniques included only those validated through an empirical study. Knowledge of other information sources, such as practices by experienced testers in companies should be added;
- Finding testers with practical experience in concurrent software in the industry to participate in the evaluation of SeleCTT-v2 was a difficult task; several ways of contact were attempted, however, no positive response was obtained.

6.4 Future work

Below are some suggestions for future work related to the topic:

- Application of other research methods (e.g., ethnography and grounded theory) with more qualitative aspects through formal interviews or focus groups in software companies towards the understanding of the way the process of selection of testing techniques is performed and proposals of new approaches;
- Assessment with testers for validating the influence of the catalog proposed;
- Extension of the selection attributes to other domains (e.g., mobile applications);
- Identification of the way human factors influence the testing techniques selection process. The present study was motivated by the survey results, which showed testers consider the scope of the project, time, cost, complexity, risk and previous experiences in the selection of a testing technique;
- Definition of a systematic process that efficiently guides testers in the industry in the testing technique selection.

6.5 Publications

During the development of the research, the following papers were written, submitted, and published originated by this master's work:

Full-papers

- ★ Santos, I., Melo, S. M., Souza, P. S. L., and Souza, S. R. S. (2019). Testing techniques selection: A systematic mapping study. In Proceedings of the XXXIII Brazilian Symposium on Software Engineering, SBES 2019, pages 347–356, Salvador, BA, Brazil.
- ★ Santos, I. and Souza, S. R. S. (2019). Study and definition of project attributes for selection of testing techniques for concurrent software. In IX Workshop on Theses and Dissertations of CBSOFT (WTDSOFT 2019), pages 24–30, Salvador, BA, Brazil. SBC.
- ★ Santos, I., Melo, S. M., Souza, P. S. L., and Souza, S. R. S. Towards a unified catalog of attributes to guide industry in software testing technique selection. In: IEEE International Conference on Software Testing, Verification and Validation (ICST), 2020, Porto, Portugal. IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2020. v. 1. p. 398-407.

Journal (Submitted)

- ★ *A survey into Brazilian companies towards an outline of the power behind testing practices.* The study is in the final stages of adjustments and minor corrections and will be submitted to the Journal of Software Engineering Research and Development (JSERD).

Other publications were originated from collaborations in disciplines and the teaching program (PAE):

Full-papers

- ★ Andrade, S., Santos, I., Junior, C. B., Júnior, M. C., Souza, S.R.S., and Delamaro, M. E. (2019). On applying metamorphic testing: an empirical study on academic search engines. In Proceedings of the 4th International Workshop on Metamorphic Testing, MET@ICSE 2019, Montreal, QC, Canada, May 26, 2019, pages9–16.
- ★ Paschoal, L. N., Santos, I., Mori, A., Conte, T. U., Souza, S. R. S. What are the differences between group and individual modeling when learning BPMN?. In: Simpósio Brasileiro de Sistemas de Informação (SBSI), 2020, São Bernardo do Campo - SP. Simpósio Brasileiro de Sistemas de Informação (SBSI), 2020.

BIBLIOGRAPHY

AMMANN, P.; OFFUTT, J. **Introduction to software testing**. [S.l.]: Cambridge University Press, 2016. Citations on pages [23](#) and [33](#).

ANTONINO, P.; GIBSON-ROBINSON, T.; ROSCOE, A. Efficient verification of concurrent systems using synchronisation analysis and sat/smt solving. **ACM Transactions on Software Engineering and Methodology (TOSEM)**, ACM New York, NY, USA, v. 28, n. 3, p. 1–43, 2019. Citations on pages [71](#), [73](#), and [77](#).

BASILI, V. R.; ROMBACH, H. D. The tame project: Towards improvement-oriented software environments. **IEEE Transactions on software engineering**, IEEE, v. 14, n. 6, p. 758–773, 1988. Citations on pages [41](#), [42](#), and [82](#).

BERTOLINO, A. Software testing research: Achievements, challenges, dreams. In: IEEE COMPUTER SOCIETY. **2007 Future of Software Engineering**. [S.l.], 2007. p. 85–103. Citation on page [31](#).

BOURQUE, P.; FAIRLEY, R. E. *et al.* **Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0**. [S.l.]: IEEE Computer Society Press, 2014. Citations on pages [31](#), [33](#), and [34](#).

CAI, Y.; LU, Q. Dynamic testing for deadlocks via constraints. **IEEE Transactions on Software Engineering**, IEEE, v. 42, n. 9, p. 825–842, 2016. Citations on pages [71](#), [74](#), and [76](#).

COPELAND, L. **A practitioner's guide to software test design**. [S.l.]: Artech House, 2004. Citation on page [35](#).

COTRONEO, D.; PIETRANTUONO, R.; RUSSO, S. A learning-based method for combining testing techniques. In: IEEE PRESS. **Proceedings of the 2013 International Conference on Software Engineering**. [S.l.], 2013. p. 142–151. Citation on page [27](#).

_____. Testing techniques selection based on odc fault types and software metrics. **Journal of Systems and Software**, Elsevier, v. 86, n. 6, p. 1613–1637, 2013. Citations on pages [26](#), [47](#), [49](#), [51](#), and [53](#).

DELAMARO, M.; JINO, M.; MALDONADO, J. **Introdução ao teste de software**. [S.l.]: Elsevier Brasil, 2017. Citations on pages [31](#), [32](#), [33](#), [34](#), and [36](#).

DIAS-NETO, A. C.; TRAVASSOS, G. H. Evaluation of {model-based} testing techniques selection approaches: An external replication. In: IEEE. **Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on**. [S.l.], 2009. p. 269–278. Citation on page [24](#).

_____. Model-based testing approaches selection for software projects. **Information and Software Technology**, Elsevier, v. 51, n. 11, p. 1487–1504, 2009. Citation on page [60](#).

_____. Supporting the combined selection of model-based testing techniques. **IEEE Transactions on Software Engineering**, IEEE, v. 40, n. 10, p. 1025–1041, 2014. Citations on pages [27](#) and [47](#).

DING, Z.; ZHOU, Z. Racetest: harmful data race detection based on testing technology in ws-bpel. **Service Oriented Computing and Applications**, Springer, p. 1–14, 2019. Citations on pages [71](#), [74](#), and [75](#).

DO, H.; ELBAUM, S.; ROTHERMEL, G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. **Empirical Software Engineering**, Springer, v. 10, n. 4, p. 405–435, 2005. Citations on pages [66](#), [67](#), and [84](#).

DWYER, M. B.; HATCLIFF, J.; HOOSIER, M.; RANGANATH, V.; WALLENTINE, T. *et al.* Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In: SPRINGER. **International Conference on Tools and Algorithms for the Construction and Analysis of Systems**. [S.l.], 2006. p. 73–89. Citation on page [67](#).

DWYER, M. B.; PERSON, S.; ELBAUM, S. Controlling factors in evaluating path-sensitive error detection techniques. In: **Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering**. [S.l.: s.n.], 2006. p. 92–104. Citation on page [66](#).

ENDRES, A.; ROMBACH, H. D. **A handbook of software and systems engineering: Empirical observations, laws, and theories**. [S.l.]: Pearson Education, 2003. Citation on page [61](#).

EYTANI, Y.; HAVELUND, K.; STOLLER, S. D.; UR, S. Towards a framework and a benchmark for testing tools for multi-threaded programs. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 19, n. 3, p. 267–279, 2007. Citation on page [67](#).

FELIZARDO, K. R.; MENDES, E.; KALINOWSKI, M.; SOUZA, É. F.; VIJAYKUMAR, N. L. Using forward snowballing to update systematic reviews in software engineering. In: **Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement**. [S.l.: s.n.], 2016. p. 1–6. Citations on pages [69](#) and [70](#).

FELIZARDO, K. R.; SILVA, A. Y. I. da; SOUZA, É. F. de; VIJAYKUMAR, N. L.; NAKAGAWA, E. Y. Evaluating strategies for forward snowballing application to support secondary studies updates: emergent results. In: **Proceedings of the XXXII Brazilian Symposium on Software Engineering**. [S.l.: s.n.], 2018. p. 184–189. Citations on pages [69](#) and [70](#).

FINK, A. **The survey handbook**. [S.l.]: Sage, 2003. Citation on page [40](#).

FRANKL, P. G.; WEISS, S. N. An experimental comparison of the effectiveness of branch testing and data flow testing. **IEEE Transactions on Software Engineering**, IEEE, v. 19, n. 8, p. 774–787, 1993. Citation on page [82](#).

FURLANETTI, A. B.; SANTOS, I.; SOUZA, S. R. S. Evolution of the SeleCTT tool to support the combined selection of concurrent software testing techniques in the industry. **Universidade de São Paulo. Relatório técnico.**, São Carlos, SP, Brasil., 2020. Citation on page [62](#).

GRAMA, A. **Introduction to parallel computing**. [S.l.]: Pearson Education, 2003. Citation on page [37](#).

GRANDE, A. da S.; NETO, A. C. D.; RODRIGUES, R. de F. Providing trade-off techniques subsets to improve software testing effectiveness: using evolutionary algorithm to support software testing techniques selection by a web tool. In: **Advances in Artificial Intelligence-SBIA 2012**. [S.l.]: Springer, 2012. p. 222–232. Citation on page [27](#).

GRISSOM, R. J.; KIM, J. J. **Effect sizes for research: A broad practical approach**. [S.l.]: Lawrence Erlbaum Associates Publishers, 2005. Citation on page 90.

HANSEN, P. B. The invention of concurrent programming. In: **The origin of concurrent programming**. [S.l.]: Springer, 2001. p. 3–61. Citation on page 36.

HOWDEN, W. E. Functional program testing. **IEEE Transactions on Software Engineering**, IEEE, n. 2, p. 162–169, 1980. Citation on page 36.

IEEE, I. A. **IEEE 610.12-1990 - IEEE Standard Glossary of Software Engineering Terminology**. 1990. <<http://standards.ieee.org/findstds/standard/610.12-1990.html>>. Citations on pages 32 and 39.

ISABEL, S. S.; TRAVASSOS, G. H. Features of software testing techniques for use in projects web. **14th Ibero-American Conference on Software Engineering and 14th Workshop on Requirements Engineering, CIBSE 2011**, p. 129–142, 2011. Citations on pages 26, 47, and 53.

KOJIMA, H.; TAKAHASHI, J.; OHTA, T.; KAKUDA, Y. A model for concurrent states and its coverage criteria. In: IEEE. **Autonomous Decentralized Systems, 2009. ISADS'09. International Symposium on**. [S.l.], 2009. p. 1–6. Citation on page 24.

KRAWCZYK, H.; WISZNIEWSKI, B.; MORK, H. Classification of software defects in parallel programs. **Relatório Técnico, Faculty of Electronics, Technical University of Gdansk, Poland**, Citeseer, 1994. Citation on page 38.

LI, G.; LU, S.; MUSUVATHI, M.; NATH, S.; PADHYE, R. Efficient scalable thread-safety-violation detection: finding thousands of concurrency bugs during testing. In: **Proceedings of the 27th ACM Symposium on Operating Systems Principles**. [S.l.: s.n.], 2019. p. 162–180. Citations on pages 71 and 77.

LI, Q.; JIANG, Y.; GU, T.; XU, C.; MA, J.; MA, X.; LU, J. Effectively manifesting concurrency bugs in android apps. In: IEEE. **2016 23rd Asia-Pacific Software Engineering Conference (APSEC)**. [S.l.], 2016. p. 209–216. Citations on pages 71, 72, and 77.

LIN, Z.; ZHOU, Y.; ZHONG, H.; CHEN, Y.; YU, H.; ZHAO, J. Spdebugger: A fine-grained deterministic debugger for concurrency code. **IEICE Transactions on Information and Systems**, The Institute of Electronics, Information and Communication Engineers, v. 100, n. 3, p. 473–482, 2017. Citations on pages 71, 73, and 75.

LU, S.; JIANG, W.; ZHOU, Y. A study of interleaving coverage criteria. In: ACM. **The 6th joint meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering: companion papers**. [S.l.], 2007. p. 533–536. Citation on page 24.

MELO, S. M. **Um framework para avaliação sistemática de técnicas de teste no contexto de programação concorrente**. Phd Thesis (PhD Thesis) — Universidade de São Paulo, 2018. Citations on pages 19, 23, 24, 25, 27, 29, 32, 38, 43, 59, and 60.

MELO, S. M.; CARVER, J. C.; SOUZA, P. S. L.; SOUZA, S. R. S. Empirical research on concurrent software testing: A systematic mapping study. **Information and Software Technology**, Elsevier, v. 105, p. 226–251, 2019. Citations on pages 19, 67, 69, 75, 78, and 79.

MELO, S. M.; MOURA, F. M.; SOUZA, P. S. L.; SOUZA, S. R. S. SeleCTT: An infrastructure for selection of concurrent software testing techniques. In: **Proceedings of the IV Brazilian Symposium on Systematic and Automated Software Testing**. New York, NY, USA: Association for Computing Machinery, 2019. (SAST 2019), p. 62–71. ISBN 9781450376488. Available: <<https://doi.org/10.1145/3356317.3356324>>. Citations on pages 24, 25, 29, 51, 57, 59, and 61.

MELO, S. M.; SOUZA, P. S. L.; SOUZA, S. R. S. Towards an empirical study design for concurrent software testing. In: IEEE. **Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE), 2016 Fourth International Workshop on**. [S.l.], 2016. p. 49–49. Citation on page 42.

MELO, S. M.; SOUZA, S. R. S.; SOUZA, P. S. L.; CARVER, J. C. How to test your concurrent software: an approach for the selection of testing techniques. In: ACM. **Proceedings of the 4th ACM SIGPLAN International Workshop on Software Engineering for Parallel Systems**. [S.l.], 2017. p. 42–43. Citations on pages 26, 45, 47, 54, and 59.

MITRA, P.; CHATTERJEE, S.; ALI, N. Graphical analysis of mc/dc using automated software testing. In: IEEE. **Electronics Computer Technology (ICECT), 2011 3rd International Conference on**. [S.l.], 2011. v. 3, p. 145–149. Citation on page 34.

MOURA, F. M.; MELO, S. M.; SOUZA, S. R. S. SeleCTT: ferramenta para automatização do processo de seleção de técnicas de teste de software concorrente. **Universidade de São Paulo. Relatório técnico N° 423**, São Carlos, SP, Brasil., 2018. Citations on pages 25 and 29.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The art of software testing**. [S.l.]: John Wiley & Sons, 2011. Citations on pages 23, 32, 34, 35, and 36.

NAKAGAWA, S.; CUTHILL, I. C. Effect size, confidence interval and statistical significance: a practical guide for biologists. **Biological reviews**, Wiley Online Library, v. 82, n. 4, p. 591–605, 2007. Citation on page 90.

NETO, A. C. D.; RODRIGUES, R. de F.; TRAVASSOS, G. H. Porantim-opt: Optimizing the combined selection of model-based testing techniques. In: IEEE. **Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on**. [S.l.], 2011. p. 174–183. Citations on pages 27, 45, and 47.

NETO, A. C. D.; TRAVASSOS, G. H. Porantim: An approach to support the combination and selection of model-based testing techniques. In: IEEE. **Automation of Software Test, 2009. AST’09. ICSE Workshop on**. [S.l.], 2009. p. 1–9. Citations on pages 27, 47, and 55.

NGUYEN, T. L.; FISCHER, B.; TORRE, S. L.; PARLATO, G. Concurrent program verification with lazy sequentialization and interval analysis. In: SPRINGER. **International Conference on Networked Systems**. [S.l.], 2017. p. 255–271. Citations on pages 71 and 75.

NGUYEN, T. L.; SCHRAMMEL, P.; FISCHER, B.; TORRE, S. L.; PARLATO, G. Parallel bug-finding in concurrent programs via reduced interleaving instances. In: IEEE PRESS. **Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering**. [S.l.], 2017. p. 753–764. Citations on pages 71, 75, and 77.

NIDHRA, S.; DONDETI, J. Black box and white box testing techniques-a literature review. **International Journal of Embedded Systems and Applications (IJESA)**, v. 2, n. 2, p. 29–50, 2012. Citation on page 34.

PACHECO, P. **An introduction to parallel programming**. [S.l.]: Elsevier, 2011. Citation on page 37.

PROTZE, J.; SCHULZ, M.; AHN, D. H.; MÜLLER, M. S. Thread-local concurrency: a technique to handle data race detection at programming model abstraction. In: ACM. **Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing**. [S.l.], 2018. p. 144–155. Citations on pages 71, 72, and 76.

QI, X.; ZHOU, H. A splitting strategy for testing concurrent programs. In: IEEE. **2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)**. [S.l.], 2019. p. 388–398. Citations on pages 71, 73, and 78.

ROBSON, C. Real world research: A resource for social scientists and practitioner-researchers., 2nd edn.(blackwell publishing: Oxford, uk.). 2002. Citation on page 40.

ROTHERMEL, G.; HARROLD, M. J. Analyzing regression test selection techniques. **IEEE Transactions on software engineering**, IEEE, v. 22, n. 8, p. 529–551, 1996. Citation on page 27.

RUNESON, P.; HÖST, M. Guidelines for conducting and reporting case study research in software engineering. **Empirical software engineering**, Springer, v. 14, n. 2, p. 131, 2009. Citation on page 81.

RUNESON, P.; HOST, M.; RAINER, A.; REGNELL, B. **Case study research in software engineering: Guidelines and examples**. [S.l.]: John Wiley & Sons, 2012. Citations on pages 40, 81, and 95.

SANTOS, I.; MELO, S. M.; SOUZA, P. S. L.; SOUZA, S. R. S. Testing techniques selection: A systematic mapping study. In: ACM. **Proceedings of the XXXIII Brazilian Symposium on Software Engineering**. [S.l.], 2019. p. 347–356. Citations on pages 19, 26, 27, 28, and 45.

SARMANHO, F. S.; SOUZA, P. S. L.; SOUZA, S. R. S.; SIMÃO, A. S. Structural testing for semaphore-based multithread programs. In: SPRINGER. **International Conference on Computational Science**. [S.l.], 2008. p. 337–346. Citations on pages 24 and 39.

SOUZA, S. R. S.; SOUZA, P. S. L.; MELO, S. M.; SILVA, R. A.; VERGILIO, S. R. Teste de programas concorrentes. In: _____. Rio de Janeiro, RJ: Elsevier Editora Ltda, 2017. (In: Delamaro, Marcio and Jino, Mario and Maldonado, José. (Org.). *Introdução ao Teste de Software.*), p. 261–296. Citations on pages 19, 38, and 39.

SOUZA, S. R. S.; VERGILIO, S. R.; SOUZA, P. S. L.; SIMAO, A.; HAUSEN, A. C. Structural testing criteria for message-passing parallel programs. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, v. 20, n. 16, p. 1893–1916, 2008. Citations on pages 24 and 39.

TANENBAUM, A. S. **Structured computer organization**. [S.l.]: Pearson Education India, 2016. Citation on page 36.

TANENBAUM, A. S.; STEEN, M. V. **Distributed systems: principles and paradigms**. [S.l.]: Prentice-Hall, 2007. Citation on page 37.

VEGAS, S. Identifying the relevant information for software testing technique selection. In: IEEE. **Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on**. [S.l.], 2004. p. 39–48. Citations on pages 26 and 47.

VEGAS, S.; BASILI, V. A characterisation schema for software testing techniques. **Empirical Software Engineering**, Springer, v. 10, n. 4, p. 437–466, 2005. Citations on pages [24](#), [26](#), [45](#), and [47](#).

VEGAS, S.; JURISTO, N.; BASILI, V. Packaging experiences for improving testing technique selection. **Journal of Systems and Software**, Elsevier, v. 79, n. 11, p. 1606–1618, 2006. Citations on pages [26](#), [45](#), and [47](#).

VICTOR, M.; UPADHYAY, N. Selection of software testing technique: A multi criteria decision making approach. In: **Trends in Computer Science, Engineering and Information Technology**. [S.l.]: Springer, 2011. p. 453–462. Citations on pages [26](#), [27](#), [45](#), and [47](#).

VISSER, W.; PARK, S.; PENIX, J. Using predicate abstraction to reduce object-oriented programs for model checking. In: **Proceedings of the third workshop on Formal methods in software practice**. [S.l.: s.n.], 2000. p. 3–182. Citation on page [67](#).

VOS, T. E.; MARIN, B.; ESCALONA, M. J.; MARCHETTO, A. A methodological framework for evaluating software testing techniques and tools. In: IEEE. **2012 12th international conference on quality software**. [S.l.], 2012. p. 230–239. Citation on page [45](#).

WANG, P.; KRINKE, J.; ZHOU, X.; LU, K. Avpredictor: Comprehensive prediction and detection of atomicity violations. **Concurrency and Computation: Practice and Experience**, Wiley Online Library, p. e5160, 2019. Citations on pages [71](#) and [77](#).

WANG, P.; LU, K.; LI, G.; ZHOU, X. Dftracker: detecting double-fetch bugs by multi-taint parallel tracking. **Frontiers of Computer Science**, Springer, v. 13, n. 2, p. 247–263, 2019. Citations on pages [71](#) and [75](#).

WOHLIN, C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: **Proceedings of the 18th international conference on evaluation and assessment in software engineering**. [S.l.: s.n.], 2014. p. 1–10. Citations on pages [69](#), [70](#), [71](#), and [78](#).

WOHLIN, C.; RUNESON, P.; HÖST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLÉN, A. **Experimentation in software engineering**. [S.l.]: Springer Science & Business Media, 2012. Citations on pages [17](#), [40](#), [41](#), [42](#), and [95](#).

WOJCICKI, M. A.; STROOPER, P. An iterative empirical strategy for the systematic selection of a combination of verification and validation technologies. In: IEEE COMPUTER SOCIETY. **Proceedings of the 5th International Workshop on Software Quality**. [S.l.], 2007. p. 9. Citation on page [27](#).

XU, J.; LEI, Y.; CARVER, R. Using delta debugging to minimize stress tests for concurrent data structures. In: IEEE. **2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)**. [S.l.], 2017. p. 35–46. Citations on pages [71](#) and [76](#).

XUE, X.; SIAMI-NAMINI, S.; NAMIN, A. S. Testing multi-threaded applications using answer set programming. **International Journal of Software Engineering and Knowledge Engineering**, World Scientific, v. 28, n. 08, p. 1151–1175, 2018. Citations on pages [71](#) and [76](#).

YANG, C.-S.; POLLOCK, L. L. The challenges in automated testing of multithreaded programs'. 1997. Citation on page [37](#).

YOO, S.; HARMAN, M. Regression testing minimization, selection and prioritization: a survey. **Software Testing, Verification and Reliability**, Wiley Online Library, v. 22, n. 2, p. 67–120, 2012. Citation on page [33](#).

YU, Z.; ZUO, Y.; XIONG, W. Concurrency bug avoiding based on optimized software transactional memory. **Scientific Programming**, Hindawi, v. 2019, 2019. Citations on pages [71](#), [73](#), and [77](#).

PROJECT CHARACTERIZATION

Project name: Account	
Description: Account is a project that simulates the existence of two bank accounts. The program implements the transference of money between them. The functionalities were implemented using locks (synchronize), and there are bugs related to deadlock and data race.	
Start date: 01/11/2019	End date: 15/01/2020
Characterization attributes	Values
Software features	
Software execution platform = Development Environment	Unix
Context/Objective = Type of software	HPC
Programming language	Java
Concurrent process paradigm	Shared memory
Suggested techniques	<p>4 - CARISMA: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications.</p> <p>5 - Comparative assessment of testing and model checking using program mutation.</p>
Project status	Inactive
Tests characteristics intended to be performed	
Test level desired for the project	Unit
Type of testing technique	Structural testing, Fault-based testing
Analysis type	Dynamic
Project quality characteristics	Effectiveness
Types of faults to be revealed	Data Race, Deadlock
Support testing tools	Carisma, Contest
Inputs available for the technique	Multithreaded program, source code in Java
Expected outputs	Number of faults detected
Program representation	No Information

Suggested Techniques

Technique 1: CARISMA: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications.	
Characterization attribute	Values
Execution platform	Unix
Context/Objective	HPC
Programming language	Java
Testing technique	Structural testing
Test data generation	Context-sensitive sampling
Testing level	Unit
Synchronization interleaving mechanism	Happens-before relation, locksets
Inputs required	Multithreaded program
Outputs (Results generated)	Performance: detection rate, effectiveness: detection rate, time overhead: slowdown of the strategy and space, overhead: memory consumption
Quality attributes	Effectiveness
Type of study	Case study
Testing analysis	Dynamic
Concurrent paradigm for process interaction	Shared memory
Replay mechanism	Monitoring
Program representation	No Information
State space reduction	Probabilistic sampling
Concurrent bugs	Data race
Tool name	Carisma
Cost with the tool	Academic
Platform that the tool operates	Unix
Understandability (need to know some additional technology)	Knowledge of the language in which the software is implemented
Dependencies (the approach depends on another)	It does not depend on another approach

Technique 2: Comparative assessment of testing and model checking using program mutation.

Characterization attribute	Values
Execution platform	Linux
Context/Objective	HPC
Programming language	Java
Testing technique	Fault-based testing
Test data generation	Mutation operators
Testing level	Unit
Synchronization interleaving mechanism	Synchronization events, thread schedules
Inputs required	Source code in Java
Outputs (Results generated)	Number of faults detected
Quality attributes	Effectiveness
Type of study	Experiment
Testing analysis	Dynamic
Concurrent paradigm for process interaction	Shared memory
Replay mechanism	Deterministic replay
Program representation	No information
State space reduction	Parallel randomized state-space search
Concurrent bugs	Deadlock
Tool name	ExMan, Contest, JPF
Cost with the tool	Academic
Platform that the tool operates	Linux
Understandability (need to know some additional technology)	Knowledge of the language in which the software is implemented
Dependencies (the approach depends on another)	It does not depend on another approach

Technique 3: Selective mutation testing for concurrent code

Characterization attribute	Values
Execution platform	Linux
Context/Objective	HPC
Programming language	Java
Testing technique	Fault-based testing
Test data generation	Selective mutation
Testing level	Unit
Synchronization interleaving mechanism	Selective mutation
Inputs required	Java code
Outputs (Results generated)	Time to execute schedules, number of generated mutants, number of killed mutants, number of hit mutants, mutation scores
Quality attributes	Effectiveness
Type of study	Experiment
Testing analysis	Dynamic
Concurrent paradigm for process interaction	Shared memory
Replay mechanism	Scheduling
Program representation	No Information
State space reduction	Selective mutation
Concurrent bugs	Concurrency bugs
Tool name	Comutation
Cost with the tool	Academic
Platform that the tool operates	Linux
Understandability (need to know some additional technology)	Knowledge of the language in which the software is implemented
Dependencies (the approach depends on another)	It does not depend on another approach

Technique 4: Multithreaded Test Synthesis for Deadlock Detection

Characterization attribute	Values
Execution platform	Linux
Context/Objective	HPC
Programming language	Java
Testing technique	Fault-based testing
Test data generation	Randomized test case generation
Testing level	Unit
Synchronization interleaving mechanism	Synthesized tests
Inputs required	Java bytecode
Outputs (Results generated)	Number of deadlocks synthesized, numbers corresponding to the synthesized multithreaded tests, deadlocks detected
Quality attributes	Effectiveness
Type of study	Case study
Testing analysis	Dynamic
Concurrent paradigm for process interaction	Shared memory
Replay mechanism	Lock dependency relation
Program representation	Object-sensitive lock graph
State space reduction	Constraint-based
Concurrent bugs	Deadlock
Tool name	OMEN
Cost with the tool	No information
Platform that the tool operates	Linux
Understandability (need to know some additional technology)	Knowledge of the language in which the software is implemented
Dependencies (the approach depends on another)	It does not depend on another approach

References

Technique 1	ZHAI, Ke et al. CARISMA: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. 2012. p. 221-231.
Technique 2	BRADBURY, Jeremy S.; CORDY, James R.; DINGEL, Juergen. Comparative assessment of testing and model checking using program mutation. In: Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007). IEEE, 2007. p. 210-222.
Technique 3	GLIGORIC, Milos et al. Selective mutation testing for concurrent code. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. 2013. p. 224-234.
Technique 4	SAMAK, Malavika; RAMANATHAN, Murali Krishna. Multithreaded test synthesis for deadlock detection. In: Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. 2014. p. 473-489.

Final Recommendation

Technique 1	CARISMA: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications.
Technique 2	Comparative assessment of testing and model checking using program mutation
Technique 3	Selective mutation testing for concurrent code
Technique 4	Multithreaded Test Synthesis for Deadlock Detection

CHARACTERIZED PROJECTS LIST

ID	Project Name	Project Description
P1	Account	Account is a project that simulates the existence of two bank accounts. The program implements the transference of money between them. The functionalities were implemented using locks (synchronize), and there are bugs related to deadlock and data race.
P2	Airline	This program is from a benchmark suite developed at IBM to support testing and analysis research and simulates an airline software. The project presents the concurrent paradigm of shared memory. The software project developed will be applied in the context of HPC programs, and it was developed using Java language in the Linux execution platform. This software project should be tested at the level of Unit and System. This project presents concurrent bugs related to data race and atomicity violation.
P3	Alarmclock	This is a simulation of an Alarm Clock. One thread of kind Clock calls a method tick() every millisecond (or any other unit of time) and updates the clock. Method tick() is a part of class Monitor to ensure that when the thread changes current time, no one is there to interfere. Other threads of kind Client call method wakeme(int interval) when they want to sleep for the interval. A list of locks is maintained for Client threads. Each thread calculates its waketime and goes to sleep on a lock with a key that's equal to wakeup time. This project presents concurrent bugs related to null-pointer dereferences and starvation.
P4	Allocationvector	It is a concurrency example originally in the IBM suite of objects that allocates space in the vector. The project presents the concurrent paradigm of message passing. The software project developed will be applied in the context of HPC and distributed systems programs, and it was developed using Java language in the Linux execution platform. This software project should be tested at the level of Unit and System. Also, this project presents concurrent bugs related to bug-triggering interleavings and deadlock.

Table continued from previous page

P5	BoundedBuffer	The program uses a bounded buffer monitor into which the Producer thread puts characters and from which the Consumer thread gets characters. The Consumer runs slightly faster than the Producer, so start the Producer first and wait until a few buffer slots are filled before starting the Consumer. The project presents the concurrent paradigm of shared memory. The software project developed will be applied in the context of distributed systems, and it was developed using Java language to Windows and Unix execution platform. This software project should be tested at the Unit level. Also, it presents concurrent bugs related to deadlock and atomicity violation.
P6	Clean	This project introduces the concurrent paradigm of shared memory. The software project, developed in Java language and in Linux execution platform, will be applied in HPC programs. It should be tested at the Unit level towards finding faults related to refinement violations and deadlock. The parameters for the program are number of first task threads (iFirstTask=1), number of second task threads (iSecondTask=1), and number of iterations (iterations=12).
P7	Daisy	The daisy subject is a prototype implementation of a NFS UNIX-like filesystem. This project presents the concurrent paradigm of shared memory. The software project developed will be applied in the context of HPC programs, and it was developed using Java language. This software project should be tested at the Unit level. Daisy expresses an assertion violation and an atomicity violation bug.
P8	Deos	Deos is a scheduler from a real-time executive for avionics systems. The project presents the concurrent paradigm of shared memory. The software project developed will be applied in the context of HPC programs, and it was developed using Java language in the Windows execution platform. This software project should be tested at the Unit level. Also, it presents concurrent bugs related to assertion violation and deadlock.
P9	Dining philosophers	The diningPhilosophers program is the classic concurrency problem first posited by Dykstra and Hoare implemented in Java. The project presents the concurrent paradigm of shared memory. The software project developed will be applied in the context of HPC programs, and it was developed using Java language. This software project should be tested at the Unit level. Also, it presents concurrent bugs related to deadlock and high-level concurrency errors.
P10	LoseNotify	The project introduces the concurrent paradigm of shared memory. The software project, developed in Java language and in Linux execution platform, will be applied in HPC and distributed systems programs. It should be tested at the Unit level towards finding faults related to deadlock and data race. The parameters for the program are number of wait threads (iWait=1), number of notify threads (iNotify=1), and number of iterations (iterations=12).
P11	Linkedlist	The linkedlist program presents a project that uses and creates different types of lists, the concurrent paradigm of shared memory and message passing. The software project developed will be applied in the context of distributed systems and HPC programs, and it was developed using Java language to Windows and Unix execution platform. This software project should be tested at the unit level. Also, it presents concurrent bugs related to deadlock and data race.

Table continued from previous page

P12	Logging event	The project record the events log of thread interactions. It presents the concurrent paradigm of shared memory and message passing. The software project developed will be applied in the context of distributed systems and HPC programs. It was developed using Java language to Windows and Unix execution platform. This software project should be tested at the Unit level. It presents concurrent bugs related to deadlock, null-pointer dereferences, and atomicity violation.
P13	Nested monitor	The Nested Monitor program is a non-deterministic implementation of a bounded buffer that uses semaphores instead of Java condition-based synchronization. It may deadlock because of nested monitor calls, depending on the thread ordering executed by the scheduler. A fixed version of the program that does not use synchronized get() and put() methods is included. This project presents concurrent bugs related to deadlock and starvation.
P14	Piper	The program introduces the concurrent paradigm of message passing. The software project, developed in Java language for Linux execution platform, will be applied in distributed systems. It should be tested at the level of the system, towards finding faults related to deadlock and data race.
P15	Producer consumer	The subject presented here is the Java version of the classic producer-consumer problem that provides an inadequate concurrency solution, which results in a race condition. The project introduces the concurrent paradigm of shared memory. The software project, developed in Java language, will be applied in HPC programs. It should be tested at the Unit level, towards finding faults related to data race and assertion violation.
P16	Raxextended	This group calendar scheduling application expresses a race condition in the concurrency. The project introduces the concurrent paradigm of shared memory. The software project, developed in Java language, will be applied in HPC programs. It should be tested at the Unit level, towards finding faults related to data race and null-pointer dereferences.
P17	Readerswriters	The program provides Readers with concurrent access to a resource, while restricting Writers to exclusive access. Thread access to the shared resource is depicted by a light blue (cyan) arc segment, and is allowed by starting the Reader threads at different times for the creation of an execution in which Writer threads never gain access to the resource. The project introduces the concurrent paradigm of shared memory. The software project, developed in Java language for Windows execution platform, will be applied in HPC and distributed systems programs. It should be tested at the Unit level, towards finding faults related to deadlock and data race.
P18	Reorder	The project introduces the concurrent paradigm of shared memory. The software project, developed in Java language for Windows execution platform, will be applied in distributed systems and HPC programs. It should be tested at the unit level, towards finding faults related to atomicity violation and starvation.
P19	Replicated workers	The project introduces the concurrent paradigm of shared memory and message passing. The software project, developed in Java language for Linux execution platform, will be applied HPC and distributed systems programs. It should be tested at the level of system and unit, towards finding faults related to deadlock and assertion violation.

Table continued from previous page

P20	Sleeping barber	The program is based upon the famous Sleeping Barber problem, and introduces the concurrent paradigm of message passing and shared memory. The software project, developed in Java language for Linux execution platform, will be applied in HPC programs. It should be tested at the unit level, towards finding faults related to deadlock and bug-triggering interleavings.
-----	-----------------	--

FORWARD SNOWBALLING SEED SET LIST

ID	Study title	Public. Year	All Citations	Citations 2016-2019
1	Types for safe locking: Static race detection for Java	1997	259	79
2	Incremental integration testing of concurrent programs	2002	60	4
3	Concurrent software verification with states, events, and deadlocks	2005	78	19
4	Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs	2006	121	20
5	AVIO: Detecting atomicity violations via access interleaving invariants	2006	448	95
6	Enforcer - Efficient failure injection	2006	17	1
7	Reachability testing of concurrent programs	2006	154	26
8	Runtime analysis of atomicity for multithreaded programs	2006	190	26
9	A combinatorial testing strategy for concurrent programs	2007	58	17
10	A race-detection and flipping algorithm for automated testing of multi-threaded programs	2007	75	25
11	Comparative assessment of testing and model checking using program mutation	2007	25	2
12	Effective random testing of concurrent programs	2007	105	37
13	Iterative context bounding for systematic testing of multithreaded programs	2007	481	124
14	CTrigger: Exposing atomicity violation bugs from their hiding places	2008	315	80
15	Fair stateless model checking	2008	96	23
16	Incremental testing of concurrent programs using value schedules	2008	25	4
17	Learning from mistakes: a comprehensive study on real world concurrency bug characteristics	2008	899	317
18	Race directed random testing of concurrent programs	2008	375	135
19	TM UNIT : A Transactional Memory Unit Testing and Workload Generation Tool	2008	0	0
20	A meta heuristic for effectively detecting concurrency errors	2009	15	3

Table continued from previous page

21	Clash of the Titans: Tools and techniques for hunting bugs in concurrent programs	2009	22	6
22	Kendo: efficient deterministic multithreading in software	2009	416	90
23	Saturation-based testing of concurrent programs	2009	32	8
24	A Lightweight and Portable Approach to Making Concurrent Failures Reproducible	2010	15	3
25	A platform for search-based testing of concurrent software	2010	34	11
26	A randomized scheduler with probabilistic guarantees of finding bugs	2010	233	103
27	Composable Specifications for Structured Shared-Memory Communication	2010	12	2
28	ConMem: Detecting severe concurrency bugs through an effect-oriented approach	2010	120	38
29	Distributed reachability testing of concurrent programs	2010	27	11
30	How Good is Static Analysis at Finding Concurrency Bugs?	2010	21	9
31	InstantCheck: Checking the determinism of parallel programs using on-the-fly incremental hashing	2010	16	5
32	Kivati: Fast detection and prevention of atomicity violations	2010	63	19
33	MuTMuT: Efficient Exploration for Mutation Testing of Multithreaded Code	2010	51	21
34	Preemption sealing for efficient concurrency testing	2010	35	11
35	Applications of Model Reuse When Using Estimation of Distribution Algorithms to Test Concurrent Software	2011	8	1
36	ConSeq: Detecting Concurrency Bugs through Sequential Errors	2011	121	45
37	Coverage guided systematic concurrency testing	2011	79	32
38	Detection of asynchronous message passing errors using static analysis	2011	38	13
39	Efficient data race detection for distributed memory parallel programs	2011	48	24
40	Maximizing synchronization coverage via controlling thread schedule	2011	6	3
41	MultiOtter: Multiprocess Symbolic Execution	2011	7	3
42	RACEZ : A Lightweight and Non-Invasive Race Detection Tool for Production Applications	2011	58	25
43	SOS: Saving Time in Dynamic Race Detection with Stationary Analysis	2011	25	13
44	Testing Concurrent Programs on Relaxed Memory Models	2011	66	40
45	Automatic testing environment for multi-core embedded software - ATEMES	2012	22	9
46	BALLERINA: automatic generation and clustering of efficient random unit tests for multithreaded code	2012	55	24
47	CARISMA: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications	2012	31	16
48	Coverage metrics for saturation-based and search-based testing of concurrent software	2012	25	12

Table continued from previous page

49	DTAM: Dynamic Taint Analysis of Multi-threaded Programs for Relevancy Malay Ganai	2012	33	14
50	Exploiting parallelism in deterministic shared memory multiprocessing	2012	29	7
51	Finding Atomicity-Violation Bugs through Unserializable Interleaving Testing	2012	23	13
52	Location pairs: a test coverage metric for shared-memory concurrent programs	2012	10	5
53	Maple: a coverage-driven testing tool for multithreaded programs	2012	138	76
54	Multicore Acceleration of Priority-Based Schedulers for Concurrency Bug Detection	2012	54	22
55	On a Technique for Transparently Empowering Classical Compiler Optimizations on Multithreaded Code	2012	2	1
56	SAM: Self-adaptive dynamic analysis for multithreaded programs	2012	8	1
57	Sound Predictive Race Detection in Polynomial Time	2012	143	88
58	TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs	2012	68	41
59	Verification and Coverage of Message Passing Multicore Applications	2012	9	4
60	A Modular Approach to Model-based Testing of Concurrent Programs	2013	4	3
61	An approach to testing commercial embedded systems	2013	11	8
62	An Automation-Assisted Empirical Study on Lock Usage for Concurrent Programs	2013	11	7
63	An Empirical Evaluation of the Cost and Effectiveness of Structural Testing Criteria for Concurrent Programs	2013	6	4
64	An empirical study of data race detector tools	2013	7	7
65	ASP: Abstraction Subspace Partitioning for Detection of Atomicity Violations with an Empirical Study	2013	6	5
66	Automated Classification of Data Races Under Both Strong and Weak Memory Models	2013	18	15
67	Con2Colic Testing	2013	62	47
68	Efficient concurrency bug detection across inputs	2013	28	17
69	Efficient mutation testing of multithreaded code	2013	12	10
70	Generating effective tests for concurrent programs via AI automated planning techniques	2013	7	4
71	Generating unit tests for concurrent classes	2013	26	19
72	On the Effectiveness of Contracts as Test Oracles in the Detection and Diagnosis of Functional Faults in Concurrent Object-Oriented Software	2013	10	7
73	Runtime Prevention of Concurrency Related Type-state Violations in Multithreaded Applications	2013	15	13
74	Selective mutation testing for concurrent code	2013	50	38

Table continued from previous page

75	Synchronization identification through on-the-fly test	2013	5	2
76	Test-Data Generation for Testing Parallel Real-Time Systems	2013	7	5
77	The Impact of Concurrent Coverage Metrics on Testing Effectiveness	2013	18	10
78	TRADE: Precise Dynamic Race Detection for Scalable Transactional Memory Systems	2013	4	3
79	Variable and thread bounding for systematic testing of multithreaded programs	2013	14	7
80	A model-free and state-cover testing scheme for semaphore-based and shared-memory concurrent programs	2014	3	3
81	An Efficient and Flexible Deterministic Framework for Multithreaded Programs	2014	7	6
82	Bitra: Coverage-guided, automatic testing of actor programs	2014	31	23
83	CLAP: Recording Local Executions to Reproduce Concurrency Failures	2014	106	61
84	ConcBugAssist: Constraint Solving for Diagnosis and Repair of Concurrency Bugs	2014	49	46
85	ConLock: A Constraint-based Approach to Dynamic Checking on Deadlocks in Multithreaded Programs	2014	48	33
86	Efficient Deterministic Multithreading Without Global Barriers	2014	57	27
87	Efficient Execution Path Exploration for Detecting Races in Concurrent Programs	2014	3	2
88	Embedding Semantics of the Single-Producer/ Single-Consumer Lock-Free Queue into a Race Detection Tool	2014	3	3
89	Empirical evaluation of a new composite approach to the coverage criteria and reachability testing of concurrent programs	2014	9	8
90	Pegasus: Automatic Barrier Inference for Stable Multithreaded Systems	2014	4	3
91	RECONTEST: Effective Regression Testing of Concurrent Programs	2014	25	23
92	Toward a Methodology to Expose Partially Fixed Concurrency Bugs in Modified Multithreaded Programs	2014	1	1
93	Using SPIN for automated debugging of infinite executions of Java programs	2014	12	6
94	Advances in noise-based testing of concurrent software	2015	12	10
95	Are concurrency coverage metrics effective for testing: a comprehensive empirical investigation	2015	16	13
96	ASN: A Dynamic Barrier-Based Approach to Confirmation of Deadlocks from Warnings for Large-Scale Multithreaded Programs	2015	14	11
97	Assertion Guided Symbolic Execution of Multithreaded Programs	2015	36	34
98	Concurrency Debugging with Differential Schedule Projections	2015	25	22
99	Concurrency Testing Using Schedule Bounding: an Empirical Study	2015	52	36
100	Detecting Event Anomalies in Event-based Systems	2015	40	38

Table continued from previous page

101	Effective and Precise Dynamic Detection of Hidden Races for Java Programs	2015	35	35
102	Multithreaded Test Synthesis for Deadlock Detection	2015	32	26
103	Race Detection for Android Applications	2015	109	86
104	SATCheck: SAT-Directed Stateless Model Checking for SC and TSO	2015	39	38
105	Synthesizing Racy Tests	2015	28	25
106	Synthesizing Tests for Detecting Atomicity Violations	2015	23	22
107	Unfolding based automated testing of multithreaded programs	2015	12	12
108	Value-deterministic Search-based Replay for Android Multithreaded Applications	2015	3	2
109	An Evaluation of Data Race Detectors Using Bug Repositories	2016	6	5
110	Empirical research on concurrent software testing: A systematic mapping study	2019	2	2
Total:			7162	2799
Duplicated:			-	1681
Included:			-	1118

PARTICIPANT CHARACTERIZATION FORM

Participant Characterization Form

Name:

Affiliation:

Academic degree:

☐ Graduate

☐ Master

☐ Ph.D

☐ Postdoc

Number of publications on concurrent software testing:

☐ 1 - 5

☐ 6 - 10

☐ 11 - 15

☐ 16 - 20

☐ More than 20

Experience level with concurrent software testing at the industry:

☐ Low

☐ Medium

☐ High

Number of software projects using testing techniques for concurrent software that you have participated:

☐ 1 - 5

☐ 6 - 10

☐ 11 - 15

☐ 16 - 20

☐ More than 20

Check the alternatives below according to your experience in each of the areas:

	None	I studied in class or book	Practiced on 1 project in the classroom	Used in 1 project in the industry	Used in several projects in the industry
How do you evaluate your experience in software engineering?					
How do you evaluate your degree of knowledge in software testing?					
How do you evaluate your knowledge of concurrent programming?					
How do you evaluate your knowledge of concurrent software testing?					
Have you had the experience of selecting software testing techniques for a software project? How do you rate your experience?					

PARTICIPANT FEEDBACK FORM

Participant Feedback Form

Name:

Affiliation:

Please identify the information you missed during the selection process for testing techniques:

Please identify the doubts generated during the selection process for testing techniques:

Please state in general how you based the recommendation made for each project: