**Structural testing criteria for concurrent programs
considering loop executions**

Silvia Margarita Diaz Diaz

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

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data de Depósito:

Assinatura: _____

Silvia Margarita Diaz Diaz

Structural testing criteria for concurrent programs considering loop executions

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. *EXAMINATION
BOARD PRESENTATION COPY*

Concentration Area: Computer Science and
Computational Mathematics

Advisor: Prof. Dr. Paulo Sérgio Lopes de Souza

USP – São Carlos
March 2019

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

D542s   Diaz Diaz, Silvia Margarita
              Structural testing criteria for concurrent
              programs considering loop executions / Silvia
              Margarita Diaz Diaz; orientador Paulo Sérgio Lopes
              de Souza. -- São Carlos, 2019.
              143 p.

              Dissertação (Mestrado - Programa de Pós-Graduação
              em Ciências de Computação e Matemática
              Computacional) -- Instituto de Ciências Matemáticas
              e de Computação, Universidade de São Paulo, 2019.

              1. Structural testing. 2. Testing criteria. 3.
              Concurrent programs. I. Lopes de Souza, Paulo
              Sérgio, orient. II. Título.

Silvia Margarita Diaz Diaz

**Critérios de teste estrutural para programas concorrentes
considerando a execução de loops**

Dissertação apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC-USP, como parte dos requisitos para obtenção do título de Mestra em Ciências – Ciências de Computação e Matemática Computacional. *EXEMPLAR DE DEFESA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Paulo Sérgio Lopes de Souza

USP – São Carlos
Março de 2019

This project is dedicated to my family.

ACKNOWLEDGEMENTS

To my mom Margarita, my dad Fabio, my sister Natalia and my brother Luis, for their infinite love, for always being present and showing their support despite the distance.

To my husband Juan Camilo, for his unconditional love, support, and patience, thank you for always standing next to me, and for taking out geckos for me.

To my other family Carmen and Rafael, for their support and for praying for me.

To my advisor Professor Paulo Sérgio, for his academical and personal orientation, thank you for the trust, support and time in the development of this project.

To Professor Simone, for her guidance in the resolution of doubts and orientation.

To my LaSDPC colleagues: Vinicius, Leonildo, Henrique, for their advice, cooperation and friendship.

To the LaBES colleagues, specially to: Maria, Ricardo, and Rodolfo for their guidance in the experimental study of this project.

To the University of Sao Paulo and the Institute of Mathematical and Computational Sciences, for giving me the opportunity to develop my postgraduate degree, and to the CNPq for the financial support to develop this project.

RESUMO

DIAZ DIAZ, S. M. **Crterios de teste estrutural para programas concorrentes considerando a execuo de loops**. 2019. 143 p. Dissertao (Mestrado em Cincias – Cincias de Computao e Matemtica Computacional) – Instituto de Cincias Matemticas e de Computao, Universidade de So Paulo, So Carlos – SP, 2019.

A programaco paralela imperativa para melhorar o desempenho e a resoluo eficiente de problemas, tendo uma demanda crescente na implementao de tcnicas de programaco paralela. Isso implica novos desafios no teste de software para garantir a qualidade e confiabilidade. O teste estrutural uma tcnica que permite a identificao de defeitos de concorrncia, analisando a estrutura interna do programa. No entanto, os programas concorrentes so no-determinsticos, com desafios na atividade de teste, exigindo o uso de mtodos estruturados para revelar defeitos. Os critrios de teste suportam a seleo de casos de teste de forma sistemtica, analisando estaticamente elementos de programas concorrentes. Foi descoberto que atualmente existem lacunas na definio de critrios de teste contemplando cenrios com elementos dinmicos, como a execuo de primitivas de comunicao dentro de loops. O objetivo deste projeto definir critrios estruturais para orientar a seleo de casos de teste, revelando erros relacionados ao no-determinismo e melhorando a confiabilidade de programas concorrentes. Foi desenvolvida uma Taxonomia de Defeitos Concorrentes, identificando e classificando os tipos de defeitos de concorrncia encontrados na literatura relacionada. A anlise de tais defeitos, a seleo de caminhos de loop, o nmero de iteraes de loop e loops aninhados permitem modelar os critrios de testes estruturais propostos. Foram definidos novos conjuntos e associaes relacionadas aos fluxos de comunicao e sincronizao de programas de passagem de mensagens, estabelecendo um modelo para os critrios de teste. O modelo de teste proposto foi implementado no prottipo de ferramenta de teste chamada ValiMPI, considerando as associaes definidas para os critrios propostos, gerando elementos necessrios e cobertura de avaliao aps a identificao dos ns de loop. Para a avaliao da aplicao dos critrios, foi realizado um estudo emprico com validao estatstica, indicando os resultados para custo, efetividade e *strength*. A avaliao experimental demonstrou que os critrios de teste propostos geram elementos necessrios que suportam a identificao de defeitos presentes em diferentes iteraes dos loops, quando existem eventos de comunicao com comportamento nodeterminstico.

Palavras-chave: Teste estrutural, Programas concorrentes, Critrios de teste estrutural.

ABSTRACT

DIAZ DIAZ, S. M. **Structural testing criteria for concurrent programs considering loop executions**. 2019. 143 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, 2019.

Parallel programs are imperative for improving performance and problem solving, having an increasing demand on implementing efficient parallel programming techniques. This entails new challenges on software testing to ensure their quality and reliability. Structural testing is a technique that allows the identification of concurrency defects by analyzing the internal structure of the program. However, the non-determinism of concurrent programs has implications in the testing activity, requiring the use of structured methods to reveal defects. Testing criteria support the selection of test cases in a systematical form by statically analysing elements of concurrent programs. We found that there are currently gaps in the definition of testing criteria contemplating scenarios with elements that are dynamically evaluated, such as the execution of communication primitives inside loops. The objective of this project is to define structural testing criteria to guide the selection of test cases, improving the reliability of concurrent programs by revealing non-determinism related errors present in repetition structures. We developed a Concurrent Defects Taxonomy, identifying and classifying concurrency types of defects found in related literature. The analysis of such defects, paths inside loops, number of loop iterations, and nested loops allow us to model the proposed structural testing criteria. We define new sets and associations related to communication and synchronization flows for message-passing programs, establishing a model for testing criteria. We implemented the proposed test model in ValiMPI, a testing tool prototype, considering the new concepts defined in our test model, generating required elements and evaluating coverage after constructing loop paths. For the application evaluation of criteria we perform an empirical study with statistical validation, indicating the results for cost, effectiveness and strength. Our experimental evaluation demonstrated that the proposed testing criteria generates required elements that support the identification of concurrency defects occurring in different loop iterations, when having communicational events with non-deterministic behavior.

Keywords: Structural testing, Concurrent programs, Structural testing criteria.

LIST OF FIGURES

Figure 1 – Shared memory and message passing architectures (TANENBAUM, 2007) .	35
Figure 2 – ValiInst module	47
Figure 3 – ValiElem module	48
Figure 4 – ValiExec module	49
Figure 5 – ValiEval module	50
Figure 6 – Parallel Control Flow Graph for GCD program	78
Figure 7 – Required elements file example for All-sync-events-loop	87
Figure 8 – Automata descriptor file example for All-sync-events-loop	87
Figure 9 – DFA for All-events-s-loop criterion	87
Figure 10 – DFA for All-events-r-loop criterion	88
Figure 11 – DFA for All-sync-events-loop criterion	88
Figure 12 – DFA for All-defs-recv-loop criterion	88
Figure 13 – DFA for All-s-uses-loop criterion	89
Figure 14 – DFA for All-s-c-uses-loop criterion	89
Figure 15 – DFA for All-s-p-uses-loop criterion	90
Figure 16 – ValiEval results for All-sync-events-loop	94
Figure 17 – Cost for required elements for criteria	110
Figure 18 – Cost for number of adequate test cases for criteria	111
Figure 19 – Cost for non-executable required elements for criteria	111
Figure 20 – Criteria effectiveness	112
Figure 21 – Dendogram for the strength of AERL test case set	120
Figure 22 – Dendogram for the strength of AESL test case set	121
Figure 23 – Dendogram for the strength of ASEL test case set	121
Figure 24 – Dendogram for the strength of ASUL test case set	122
Figure 25 – Dendogram for the strength of ASCUL test case set	122
Figure 26 – Dendogram for the strength of ASPUL test case set	123

LIST OF ALGORITHMS

Algorithm 1 – GCD Master	26
Algorithm 2 – GCD Slave	27

LIST OF TABLES

Table 1 – Main characteristics of related work studies	58
Table 2 – Testing tools present in the selected studies	59
Table 3 – Concurrent Defects Taxonomy for shared and distributed memory programs	68
Table 4 – Definition sets for GCD program	79
Table 5 – Test model sets for GCD program	79
Table 6 – Required elements for structural testing loop criteria	80
Table 7 – L.<function> file structure	82
Table 8 – Loop path selection methods	84
Table 9 – Base pattern of required elements for loop criteria	86
Table 10 – Loop structural testing criteria - Regular expression and required elements pattern	91
Table 11 – Required elements for All-sync-events-loop criteria	93
Table 12 – Features of benchmarks analyzed during selection	98
Table 13 – List of C/MPI message-passing benchmarks	100
Table 14 – Number of injected defects in each program	104
Table 15 – Coverage with adequate set of test cases	108
Table 16 – Required elements/non-executable required elements	108
Table 17 – Number of adequate test cases	109
Table 18 – Criteria effectiveness	112
Table 19 – Strength for GCD (Two slaves)	113
Table 20 – Strength for GCD (Three slaves)	114
Table 21 – Strength for Global sum	114
Table 22 – Strength for Sieve of Eratosthenes	114
Table 23 – Strength for Sieve of Eratosthenes (fixed processes)	115
Table 24 – Strength for Jacobi	115
Table 25 – Strength for Matrix multiplication	115
Table 26 – Strength for Trapezoidal rule	115
Table 27 – Strength for Odd even sort	116
Table 28 – Strength for Search number	116
Table 29 – Strength for Philosopher’s dining	116
Table 30 – Grouping table for number of required elements - Kruskal-Wallis	117
Table 31 – Grouping table for number of adequate test cases - Kruskal-Wallis	117
Table 32 – Grouping table for number of non-executable required elements - Kruskal-Wallis	118

Table 33 – Grouping table for number of criteria effectiveness - Kruskal-Wallis 119

CONTENTS

1	INTRODUCTION	23
1.1	Context and motivation	23
1.2	Objective	27
1.3	Work organization	27
2	THEORETICAL BASIS	29
2.1	Initial considerations	29
2.2	Software testing	29
2.2.1	<i>Structural testing</i>	31
2.3	Concurrent programs	34
2.3.1	<i>Structural testing of concurrent programs</i>	36
2.4	Final considerations	37
3	PREVIOUS WORK	39
3.1	Initial considerations	39
3.2	Methods and approaches for testing concurrent programs	39
3.3	Structural testing of message-passing programs	41
3.4	ValiPar testing tool	44
3.4.1	<i>ValiPar versions</i>	45
3.4.2	<i>ValiMPI</i>	46
3.5	Empirical evaluation of structural testing criteria	50
3.6	Final considerations	52
4	RELATED WORK	53
4.1	Initial considerations	53
4.2	Loop path selection	53
4.3	Approaches for testing concurrent programs	54
4.3.1	<i>Testing criteria for concurrent programs</i>	56
4.4	Testing tools for concurrent programs	57
4.5	Final considerations	60
5	CONCURRENT DEFECTS TAXONOMY	61
5.1	Initial considerations	61
5.2	Preliminary studies on concurrency defects definition	61

5.3	Concurrency defects taxonomy	66
5.4	Final considerations	69
6	STRUCTURAL TESTING CRITERIA FOR CONCURRENT PRO-	
	GRAMS	71
6.1	Initial considerations	71
6.2	Introduction	71
6.2.1	<i>Number of loop iterations</i>	72
6.2.2	<i>Nested loops</i>	73
6.3	Test model concepts	74
6.4	Structural testing criteria for message passing programs	76
6.5	Test model exemplification	77
6.6	Final Considerations	79
7	TESTING CRITERIA IMPLEMENTATION	81
7.1	Initial considerations	81
7.2	Modifications in ValiMPI modules	81
7.2.1	<i>ValiInst</i>	81
7.2.2	<i>ValiElem</i>	82
7.2.3	<i>ValiExec</i>	92
7.2.4	<i>ValiEval</i>	92
7.3	Final considerations	94
8	EXPERIMENTAL VALIDATION	97
8.1	Initial considerations	97
8.2	Benchmarks	97
8.3	Experimental design	100
8.3.1	<i>Objective</i>	100
8.3.2	<i>Metrics</i>	101
8.3.3	<i>Object definition</i>	102
8.3.4	<i>Hypotheses formulation</i>	102
8.3.5	<i>Variable selection</i>	102
8.3.6	<i>Experimental setup</i>	103
8.3.7	<i>Threats to validity</i>	106
8.4	Execution of experiments and analysis of results	107
8.4.1	<i>Cost results</i>	107
8.4.2	<i>Effectiveness results</i>	110
8.4.3	<i>Strength results</i>	113
8.5	Hypotheses testing	116
8.5.1	<i>Cost analysis</i>	116

8.5.2	<i>Effectiveness analysis</i>	119
8.5.3	<i>Strength analysis</i>	119
8.6	Discussion of results	123
8.7	Final considerations	125
9	CONCLUSION	127
9.1	Research characterization	127
9.2	Contributions	128
9.3	Challenges and limitations	129
9.4	Future work	129
BIBLIOGRAPHY		131
APPENDIX A	BENCHMARKS DOCUMENTATION	139

INTRODUCTION

1.1 Context and motivation

Concurrent programs are widely used in all kinds of software applications, providing solutions to improve efficiency, real-time support, optimization, and performance. Parallel programming in cooperation with parallel architectures manage processor and memory resources, achieving faster and more efficient programs. Because of this, parallel systems are build for applications that require the mentioned characteristics, such as data analysis, climate modeling, energy research, bio-engineering processing and other real-world programs ([PACHECO, 2011](#)). The reliability and accuracy of parallel systems are essential to accomplish their purpose and avoid failure repercussion.

Non-determinism is a characteristic of parallel programs caused by synchronization and communication events in parallel programs, where it is possible to execute different paths having the same input data, obtaining different expected outcomes. As software development techniques are enforced to deal with the often changing requirements when focusing in parallel systems, testing methodologies also need to be adapted and enhanced to ensure concurrent software quality. Such characteristics generate challenges for both software development and software testing.

Structural testing is an appropriate technique to test concurrent programs. It is possible to reveal defects related to concurrency events by analysing the structure of the program, guiding the selection of test cases to cover required elements provided by testing criteria. Testing criteria establishes rules to systematically select test data to cover required elements based on the control, data, communication and synchronization flows of concurrent programs.

The implementation of exhaustive and robust testing methods could diminish the risks and impact of threats caused by the insufficient or lack of software quality, also preventing additional and unnecessary efforts in the software development process. Therefore, it is imperative to define

testing criteria to guide the test activity.

Due to the complexity of concurrent programs, the testing activity is not trivial, and using sequential programs and standard testing approaches do not provide sufficient elements to improve the quality of parallel systems. Such difficulty is a consequence of the inherent non-determinism in the execution of concurrent programs, their dynamic events, the presence of infeasible paths, and the selection of adequate test data. These aspects should be the focus of a solution to obtain the highest reliability at the lowest cost.

When a program executes the same block of code more than once, it might find a previously unidentified defect, as there are different data interactions. Communication primitives inside loops are an example of such situation, where new interleavings among processes occur in each iteration. Thus, it is appropriate to analyze the behavior of elements inside loops for control, data and communication flow of parallel programs.

In order to exemplify our proposal, observe Algorithms 1 and 2, which present an implementation for the Greatest Common Divisor - GCD (DOURADO, 2015). The program calculates the GCD for three numbers using three processes: a Master process and two Slaves. Variable *iter* controls the number of iterations, and *result* controls the result of the algorithm. The Master process reads the three values and sends to Slave 1 the first and the second values, and to Slave 2 the second and third values. Each process calculates the GCD of the two received numbers and sends it back to the Master process. If both values are equal to one it means that the GCD is 1 and the algorithm finishes, otherwise, both values are sent to Slave 1 to calculate the final GCD. The Master process finishes the execution when sending both Slaves the value 0 (see line 6 in Algorithm 2).

An observability error related to synchronization events (DELAMARO; MALDONADO; JINO, 2007) can be injected if the variable *firstValue* is changed for *secondValue* in line 34. If the input values of the algorithm are $x = 2$, $y = 4$, $z = 8$, the *while* loop instruction in line 8 will be executed two times since the values received in the first iteration will be different (Slave 1 sends value 2 and Slave 2 sends value 4), so the *result* variable will not be modified and it will re-evaluate the condition of the mentioned *while*.

Thus, in the second iteration, if the value from Slave 2 is received first than the value from Slave 1 (*secondValue* = 4 and *firstValue* = 2), the *if* in line 33 will have as result for variable *result* the value of variable *secondValue*, which in this case is 4. This is an error that we could identify if the execution happens in such order. On the other hand, if the value from Slave 1 is received first than the value from Slave 2 (*secondValue* = 2 and *firstValue* = 4), the *if* in line 33 will have as a result the value of variable *secondValue*, which in this case is 2 and it is the expected result for the GCD for 2, 4, 8. This means that the error will not be found because it will return a valid output but having a defect.

Current coverage criteria for concurrent programs (SOUZA *et al.*, 2008; SOUZA;

[SOUZA; ZALUSKA, 2014](#)) is not defined to reveal these types of errors. Moreover, other coverage criteria does not involve the execution of loops in concurrent programs (see Section 4.3).

For the development of this project we performed a systematic mapping review ([DIAZ; SOUZA, 2017](#)), where we found a gap in the definition of structural testing criteria that contemplates dynamic aspects in concurrent programs. In most of the literature, coverage testing do not necessarily specify how to generate required elements that guide the selection of test cases. Indeed, most of the studies apply implicit criteria for path selection. The heuristic behind structural testing is clear: the systematic execution of specific parts of a program increases the chances to reveal unknown defects improving software quality. However, the lack of explicit testing criteria might lead to the generation of redundant test cases, increasing cost and the use of incomplete sets of test data that are not capable to reveal unknown errors.

The definition of testing criteria is a challenge because it involves the construction of test models, analysis of types of defects in concurrent programs and the development of a supporting tool, as well as applicability evaluation through experimental studies. These activities imply a significant effort, since there are different paradigms and parallel architectures that require testing evaluations. However, once established a test model through testing criteria, the cost of performing structural testing decreases by guiding a specific selection of test cases. Test data generation systems might also be enhanced by the implementation of testing criteria and other dynamic analysis methods.

Another gap is the lack of studies using empirical evaluation methods with statistical support, to validate testing tools, methods and test models, when in fact these are also important for parallel programs reliability.

We employ an approach for criteria definition that supports dynamic aspects in the execution of concurrent programs, enabling the identification of defects in distinct occurrences of synchronization and communication events.

In this work we propose a set of structural testing criteria contemplating the number of loop iterations, nested loops, non-determinism defects, and selection of paths in repetition structures. The modeling of proposed criteria is guided by an analysis of synchronization and non-determinism related defects, which is based on a Concurrent Defects Taxonomy [5.3](#).

Through the implementation of criteria in ValiMPI tool and Software Experimental Engineering ([WOHLIN *et al.*, 2012](#)), we evaluate the application of our test model in message-passing programs, demonstrating that the proposed criteria is effective to reveal concurrency defects. The empirical study design contemplates the evaluation of effectiveness, application cost and strength among criteria.

Algorithm 1 – GCD Master

```

1: Define  $x, y, z, firstValue, secondValue, thirdValue, sent, values[2], iter, result$ ;
2:  $iter \leftarrow 2$ ;
3:  $result \leftarrow -1, sent \leftarrow 0$ ;
4:  $read(x, y, z)$ ;
5:  $firstValue \leftarrow x$ ;
6:  $secondValue \leftarrow y$ ;
7:  $thirdValue \leftarrow z$ ;
8: while  $result = -1$  do
9:   while  $sent < iter$  do
10:    if  $sent = 0$  then
11:       $values[0] \leftarrow firstValue$ ;
12:       $values[1] \leftarrow secondValue$ ;
13:       $send(sent + 1, values, 2)$ ;
14:    else
15:       $values[0] \leftarrow secondValue$ ;
16:       $values[1] \leftarrow thirdValue$ ;
17:       $send(sent + 1, values, 2)$ ;
18:    end if
19:     $sent \leftarrow sent + 1$ ;
20:  end while
21:  while  $sent > 0$  do
22:     $receive(*, values, 1)$ ;
23:    if  $sent = 2$  then
24:       $secondValue \leftarrow values[0]$ ;
25:    else
26:       $firstValue \leftarrow values[0]$ ;
27:    end if
28:     $sent \leftarrow sent - 1$ ;
29:  end while
30:  if  $iter = 2$  AND  $(firstValue = 1$  OR  $secondValue = 1)$  then
31:     $result \leftarrow 1$ ;
32:  else
33:    if  $iter = 1$  then
34:       $result \leftarrow firstValue$ ;
35:    end if
36:  end if
37:   $iter \leftarrow iter - 1$ ;
38: end while
39:  $values[0] \leftarrow 0$ ;
40:  $values[1] \leftarrow 0$ ;
41:  $send(1, values, 2)$ ;
42:  $send(2, values, 2)$ ;
43:  $print(result)$ ;

```

Algorithm 2 – GCD Slave

```

1: Define firstValue, secondValue, values[2], result;
2: while true do
3:   receive(0, values, 2);
4:   firstValue  $\leftarrow$  values[0];
5:   secondValue  $\leftarrow$  values[1];
6:   if firstValue = 0 AND secondValue = 0 then
7:     break;
8:   else
9:     while firstValue  $\neq$  secondValue do
10:      if firstValue < secondValue then
11:        secondValue = secondValue – firstValue;
12:      else
13:        firstValue = firstValue – secondValue;
14:      end if
15:    end while
16:  end if
17:  values[0]  $\leftarrow$  firstValue;
18:  send(0, values, 1);
19: end while

```

1.2 Objective

The objective of this project is to propose structural testing criteria for concurrent programs in the presence of communication primitives inside loops. We focus on message-passing programs with point-to-point communication patterns, contemplating different concerns when executing repetition structures.

1.3 Work organization

This work presents the theoretical basis on software testing and concurrent programs in Chapter 2. Previous work developed by our research group and the groundwork for our proposal is presented in Chapter 3. Related works on loop path selection and other approaches to test concurrent programs, exposing the state of the art and gaps in the area are shown in Chapter 4. A Concurrent Defects Taxonomy is introduced in Chapter 5, contemplating shared and distributed memory programs issues. Chapter 6 presents our main contribution: the structural testing criteria, by proposing new concepts and associations. Chapter 7 presents the development process and details of criteria implementation in ValiMPI tool. The experimental validation and discussion of results of the application of our proposed criteria is shown in Chapter 8. Finally, Chapter 9 exposes the conclusion of this project and proposes future work.

THEORETICAL BASIS

2.1 Initial considerations

This chapter presents the main theoretical basis involved in this project and how they relate to each other. Additionally, we describe the fundamental concepts employed in the development of the project.

Section 2.2 introduces the principal terminologies and techniques of software testing, Section 2.2.1 presents the structural testing technique. Section 2.3 exposes the concepts of concurrent programs and Section 2.3.1 discusses structural testing of concurrent programs. Final considerations of this chapter are found in Section 2.4.

2.2 Software testing

Software testing is the part of software engineering that intends to ensure correctness, discovering errors in early stages and reducing the risk of software failure. The main objective is to identify defects that might cause an error and/or failures in a program execution (DELAMARO; MALDONADO; JINO, 2007; AMMANN; OFFUTT, 2008). Thus, it has been taking priority in software development whereas a failure might have a big impact.

Testing is not a low-time consuming activity, so it is desirable to use tools and methods to make this task more efficient, also allowing the achievement the software testing objectives. There are two concepts involved in software testing that should be differentiated: verification and validation (AMMANN; OFFUTT, 2008). When we verify a program, the goal is to determine if the software product accomplishes the requirements from each development phase. Validation is related to the evaluation of the finished software product to guarantee conformity with the program's objective.

There are differences among the concepts of fault, mistake, error and failure. Whilst

there might have diverse definitions in the existing bibliographical base, we describe them as follows (DELAMARO; MALDONADO; JINO, 2007; AMMANN; OFFUTT, 2008):

- Mistake: a human action that produces a fault, for example a developer or a designer might introduce a flaw. It could be a design mistake.
- Fault or defect: might cause an error, due to incorrect data or procedure definition in the program.
- Error: an inconsistent or unexpected state during an execution that is the manifestation of a fault.
- Failure: external incorrect behavior related to the software requirements, or different from the expected outcome.

Test data $d \in D$ are elements in the input domain D of a program, the possible input values for the program. A test case TC is a pair formed by the test data and the expected result of the execution of the program with an element from the domain $\langle d, S(d) \rangle$, where $S(d)$ is the outcome of the program execution with test data d .

Test case selection requires the previous identification of subdomains, as choosing all possible test data to test all possible scenarios is an unpractical problem (DELAMARO; MALDONADO; JINO, 2007), even more for larger programs. The goal of defining a test subdomain is to avoid the execution of every possible path, exercising similar cases instead. Partition or subdomain testing consists of looking for establishing test case subdomains, however, the big problem with this is to identify and obtain those sets. An approach to select subdomains is by means of testing criteria (DELAMARO; MALDONADO; JINO, 2007).

Testing criteria are rules that guide the selection of test data through the generation of required elements. They provide structured and practical forms to decide which test cases are adequate for the program, and improve the quality while reducing the cost of generating test data by avoiding redundancy (DELAMARO; MALDONADO; JINO, 2007; AMMANN; OFFUTT, 2008). Criteria should be a finite set of low cardinality that considers both control and data flows, covering all branches and at least a reference to every computational result. Additionally, each instruction from the program must be executed and support the finding the most quantity of defects.

A set of tests cases TC is *C-adequate* if it satisfies the requirements of a criterion C . That is to say, there is at least a test case $t \in TC$ for each sub-domain determined by C .

There are diverse approaches for testing software, such as functional, structural and error-based methods. Functional or black-box testing looks for finding defects in terms of the software's functional requirements (or software specifications) and the perspective of the user. It implies that no information is extracted from the source code to execute a test case, by only

evaluating external descriptions of the functionality of the program (AMMANN; OFFUTT, 2008). Error-based or mutation testing uses information from the most frequent types of defects when developing software, by generating faulty versions of the program through error seeding techniques and analyzing their outcomes (DELAMARO; MALDONADO; JINO, 2007). The objective is to find test cases that reveal the different outputs of the original version of a program and its faulty versions.

Depending on the test method, testing criteria differ by the type of information used to establish subdomains and the way to obtain test data. The most commonly mentioned in software testing are functional, structural and defect-based (DELAMARO; MALDONADO; JINO, 2007; AMMANN; OFFUTT, 2008). Our project focuses on structural testing.

To mention testing criteria in functional testing: equivalence partitioning (identification of subdomains), exhaustive testing (which in practice is not viable because it takes all of the possible inputs) and random testing (random selection of test data). On the other hand, the analysis of mutant programs determines the selection of test data for error-based methods.

2.2.1 Structural testing

Structural or white box testing is a technique that uses the internal structure of a program for verification and validation, defining test cases based on branches, conditions and statements. Structural testing determines the execution of required elements extracted from the source code, following the rules established by structural testing criteria. The main goal of white-box techniques is to execute distinct and feasible logical paths by means of test cases. It is expected that this testing technique potentially forces the generation of new test cases, in order to reach higher coverage levels (DELAMARO; MALDONADO; JINO, 2007). However, structural testing requires a greater knowledge of the program compared to functional testing, making it more expensive to implement (AMMANN; OFFUTT, 2008).

Programs are represented by a Control Flow Graph (CFG), a directed graph defined by a tuple (N, E, s) , where N is the set of nodes or blocks (blocks are in-line code sequences) and a set E of edges that represents the possible deviations among the nodes. A path P in a CFG is a finite sequence of nodes and edges in the graph that follows the logic control flow of the program, such that $P = (n_1, n_2, \dots, n_m)$ where $m \in N$ is the total number of nodes of the CFG, and there is an edge from n_j to n_{j+1} for $j = (1, 2, \dots, j-1)$ (ROTHERMEL; HARROLD; ORSO, 2005). Control flow elements stand on requirements to cover program statements, nodes, edges and paths.

Data flow requirements involve the paths that occur from the definition of a variable to a reference of such definition (DELAMARO; MALDONADO; JINO, 2007). Variables have occurrences depending on their definition or use. The definition of a variable x happens when a value is stored in a memory position, a procedure call by reference or when is assigned as an

output parameter and in an input sentence. A variable is undefined when its location on memory is no longer available and there is no access to its value. The use of a variable is a reference to its definition, when the value is utilized.

Required elements in structural testing are program elements related to control and data flows, such as nodes, edges, paths, definitions, uses or associations between them. They are determined by testing criteria and should be covered by a test case (AMMANN; OFFUTT, 2008).

One of the advantages of using structural testing criteria is the cost reduction of testing activity, since criteria guides the selection of test data in a formal and structured form, avoiding to overlap test sub-domains (AMMANN; OFFUTT, 2008). Implementing testing criteria also favors traceability, regression testing, and test automation, by clearly defining test cases and a limit to apply them as the set of test cases should be finite while effective. Another advantage of structural testing is the high coverage of faults when using automation.

When having a set of input values to execute a program and a path is exercised successfully, such path is known as executable path. In structural techniques, there is not an established method to define whether a path is feasible or not, being it an undecidable problem (YANG; CHUNG, 1992). It is also a non-trivial problem to find a criterion that exercises all of the required elements (DELAMARO; MALDONADO; JINO, 2007).

The main disadvantage of structural testing are the absent or infeasible paths. An infeasible path is one that is not covered by any test data after program execution, that is to say, there is a set of input data that do not execute the path (DELAMARO; MALDONADO; JINO, 2007). Infeasible paths might appear when there is no test subdomain data that any test case is able to execute, or either because of a conditional statement (YANG; CHUNG, 1992; ROTHERMEL; HARROLD; ORSO, 2005). An example is when a program does not implement a function, then no test case will cover it. Infeasible paths lead to non-executable required elements, having an impact in the coverage analysis of a criterion.

Structural testing criteria are classified based on complexity, data flow and control flow (DELAMARO; MALDONADO; JINO, 2007). Cyclomatic complexity obtains test requirements by establishing linearly independent paths. Control flow criteria involve the execution of all-nodes, all-edges and all-paths sets of criteria, which is impractical and, in theory, it is not applicable when having big-size programs. Data flow criteria base on the relationship between the definitions and uses of a variable, also known as a *du-pair* or *def-use*.

A computational use or *c-use* occurs when a variable value is assigned, it can be found in an allocation sentence and represents nodes in the CFG. The predicate use or *p-use* happens when a variable is evaluated in a conditional statement, it is a pair that represents the edges of the graph (DELAMARO; MALDONADO; JINO, 2007; AMMANN; OFFUTT, 2008). A free definition path or *def-clear* happens when there are no re-definitions of a variable between the

first and the last nodes of a graph, in other words, the variable is not re-defined nor its value is changed. *Du-paths* are paths between a *def-use* association and have two characteristics: (1) They are simple paths (all nodes must be different except if it is the first or the last) and (2) They are *def-clear* paths. A global definition in a node i occurs when there is a definition of the variable in i and there is also a *def-clear* path to another node.

The *association* concept has a more comprehensive definition. According to (DELAMARO; MALDONADO; JINO, 2007), it might be any of the following: a ***du-path***, a ***definition-c-use*** association (triple $\langle i, j, x \rangle$, where a node i has a global definition of variable x and $j \in N_{c-use}$, the set of nodes that have a *c-use* and a *def-clear* path from i to j), or a ***definition-p-use*** association (triple $\langle i, (j,k), x \rangle$, where $j,k \in N_{p-use}$, the set of nodes that have a *p-use* and a *def-clear* path from i to j,k). Several studies that intend to define criteria are based on data flow associations.

A *def-use* graph proposed in (RAPPS; WEYUKER, 1982) is an extension of the CFG, in which for every node i with a variable x definition, the following sets are defined: *c-use*(i) (variables with a global *c-use* in i), *def*(i) (variables with global definitions in i) and *p-use* (variables with *p-use* in edge i,j). A family of criteria is introduced in (DELAMARO; MALDONADO; JINO, 2007; RAPPS; WEYUKER, 1982):

- **All-definitions:** every definition of a variable must be exercised, either for a *c-use* or for a *p-use*.
- **All-uses:** all of the associations between a definition and its *c-use* or *p-use* must be exercised by the test cases and at least by a *def-clear* path.
- **All-du-paths:** requires the execution of every association between a definition and its uses by all of the *def-clear* paths and all of the loop-free paths (every node is different including first and last).

Another family of criteria is the potential-uses, proposed in (MALDONADO, 1991). It bases on the possibility of the occurrence or existence of a variable use from its definition, i.e., there is a *def-clear* path between the definition and the possible use. Therefore, it requires that *def-clear* paths from a node with a definition are exercised, independently if there is a use of the variable in the path. It allows to verify whether a variable value was modified or not.

A potential-du-path (DELAMARO; MALDONADO; JINO, 2007) is a path in which there is a definition of the variable in a node i and a *def-clear* path from i to node k and from i to edge j,k , such that the path formed by nodes i to j is a loop-free path. The family of criteria based on potential-uses is:

- **All-potential-uses:** requires the execution of at least one *def-clear* path from the definition of a variable to every node or edge that could be possibly reached.

- **All-potential-uses/du:** must be exercised at least a potential-du-path from a variable definition to every node or edge that might be reached.
- **All-potential-du-paths:** all of the potential-du-paths must be executed from all the defined variables to every node and edge that could possibly be reached.

2.3 Concurrent programs

A process is an abstraction or instance of an executing program that has a life cycle, memory space, and a control thread (TANENBAUM, 2007). Programs might be constituted by several processes and each process has at least a single thread of control. However, sometimes it is useful to have multiple threads of control in the same address space executing in a concurrent mode. In a broader sense, a thread is a kind of lightweight process, which in turn is a running program that shares data with other threads from the same process, source code, address space, descriptors and signals (TANENBAUM, 2007; SILBERSCHATZ; GALVIN; GAGNE, 2008).

The nature of concurrent programs differ from sequential programs, whose execution is strictly in consecutive order and deterministic: a task must end so that the next one can start. Parallelism aims to improve efficiency and so there are processes executing at the same time, requiring a parallel architecture that allows dividing the tasks. Accordingly, two or more processes (or threads) have to run on multiple processors (or even multi-cores) at the same time to be considered *parallelism*. *Concurrency* is a pseudo-parallelism and is present when several threads (or processes) are executing on a single processor, not simultaneously. When single core processors implement concurrency, it provides the illusion of parallelism, but there are actually processes being scheduled on the processor in a time-shared way: when a process is executing the others wait for the processor's liberation to continue.

Distributed computing is a multi-computer model, consists of multiple systems connected by a network in which each node has its own memory and processor. Occasionally their resources are geographically located in different places, and the interaction between processes of its components is based on a message passing paradigm (TANENBAUM, 2007).

There are differences among the terms of parallel, concurrent, and distributed computing (PACHECO, 2011). In concurrent computing processes can be executing at any instant of time, and in parallel computing processes are executing coordinately to solve a problem. In distributed computing, processes in different programs might have to cooperate to accomplish an objective.

Concurrent programs are naturally non-deterministic as different executions of a process with the same input may produce different outputs. It is a consequence of synchronization and communication events among processes. Non-determinism makes the establishment of a minimal set of test cases (ARORA; BHATIA; SINGH, 2016) more difficult because there are more paths to evaluate. The communication attribute of processes enables the interference of their execution with the execution of another process, and synchronization restricts the simultaneous

access (mutual exclusion) to maintain the integrity of the variables or data (DELAMARO; MALDONADO; JINO, 2007; TANENBAUM, 2007).

Multiple instruction, multiple data (MIMD) systems support simultaneous execution of multiple instruction operations over multiple data elements (TANENBAUM, 2007). These systems consist of a group of asynchronous processing units with its own control unit and ALU (Arithmetic Logic Unit), which means that each processor can be executing different instructions in a given point of time. The two principal MIMD systems are shared memory and distributed memory systems (TANENBAUM, 2007; BRITO *et al.*, 2010). Figure 1 illustrates the differences among them (DELAMARO; MALDONADO; JINO, 2007).

Figure 1 – Shared memory and message passing architectures (TANENBAUM, 2007)

In shared memory paradigm architectures, the communication among processes happens through shared variables, and synchronization primitives are used for sequence control and establishment of order in the execution. Each processor is connected to a memory system and has access to the shared memory (TANENBAUM, 2007), thus, the address space is the same for all processes. In distributed memory each process has its own memory space and they are connected by an interconnection network, constituting a processor-memory pair for each process. Communication and synchronization events are controlled by message exchange or providing access to memory for other processes, using mechanisms such as send/receive primitives, Remote Call Procedures (RPC), Remote Method Invocation (RMI), and web services (TANENBAUM, 2007).

These two paradigms have turned into two different patterns to design, develop and test concurrent and parallel programs, with their common and own challenges, which we are not going to cover in this literature introduction.

This project focuses on message-passing programs, which rely on distributed memory systems. There are libraries of functions for process communication through send and receive primitives, an implementation of message-passing programs in C language is the Message-Passing Interface, known as MPI. MPI facilitates the development of parallel programs, however, there are a lot of details that programmers should consider when developing with MPI. The data structures, the data or task division, communication patterns (one-sided or two-sided) are some of the aspects that should be studied when programming with MPI. Regardless of its advantages

and disadvantages, MPI API is used in many systems.

2.3.1 Structural testing of concurrent programs

The main objective of structural testing in concurrent programs is to find defects related to communication and synchronization of processes, however, issues regarding sequential programs aspects must be considered as well. The most common types of defects identified when testing concurrent programs are observability and locking errors (DELAMARO; MALDONADO; JINO, 2007). Observability defects relate to synchronization (BRITO *et al.*, 2010) and occur when there are two or more paths that the program could execute. Their input domain is the same but the outcome may be different in every execution and those outputs are not controllable by the tester. Meanwhile, locking defects appear when in an execution path exists at least one element of the test domain for which all of the conditional sentences return false, so that the program does not finishes its execution staying in a locked state (KRAWCZYK; WISZNIEWSKI; MORK, ; BRITO *et al.*, 2010).

Concurrent programs also demand an adaptation on their representation in order to define the structural testing criteria. There are different representation models for concurrent programs, which have evolved through the study of different paradigms and adaptations of sequential programs graph definitions. Some of those models are (DELAMARO; MALDONADO; JINO, 2007; YANG; POLLOCK, 1999; SOUZA *et al.*, 2008):

- **Synchronization graph:** Models the behavior of the program in execution time, establishing the possible synchronizations and synchronization routes.
- **Process graph:** represents a static approach of the program, by modelling the control flow. A path might have more than one route.
- **Parallel control flow graph (PCFG):** is an extension of the GFC utilized for sequential programs. Consists on the definition of a CFG for each process of the concurrent program but adding synchronization edges, which represent the creation of parallel processes and the communication among them. It is defined as CFG^p , with $p = 0, \dots, n-1$ where p is a process, N the set of nodes and E the set of edges. There are two subsets N_s and N_r , which are nodes with send or receive message functions respectively, and E_i and E_s that correspond to sets or communication edges in and between processes respectively (SOUZA *et al.*, 2008). There are elements defined for PCFG in addition to the previously exposed:
  - *Intra-process path*; it only has edges inside the same process that do not connect with others.
  - *Inter-process path*; has communicating edges among different processes.

- *S-use*; a communicational use, when a communication command (send or receive for example) is defined in inter-process edges and allows verifying defects regarding communication aspects.

Infeasible paths in concurrent programs occur in accordance of input data and the program's logic, and might cause deadlock situations depending on the location of communication primitives. Infeasible paths are present in both sequential and concurrent programs, and can be caused by data dependencies and conditionals, or when calling a function in different nodes from different processes (YANG; SOUTER; POLLOCK, 1998).

According to (YANG; POLLOCK, 1999), the main challenges when testing concurrent programs are the representation of the program, the non-determinism and the re-execution of a test case. Other mentioned issues are the detection of errors regarding synchronization, communication and data flow, and the adaptation of the existent sequential-based criteria for concurrent programs. Several studies have addressed some of these issues (YANG; SOUTER; POLLOCK, 1998; BECHINI; TAI, 1998). On the other hand, there are also studies focusing on different approaches to define structural testing criteria that considers both memory paradigms (shared and distributed memory), with the objective of revealing more types of defects at a lower computational cost (SOUZA *et al.*, 2008; SARMANHO *et al.*, 2008).

2.4 Final considerations

In this chapter we described the basic concepts related to structural testing and concurrent programs. The presented theoretical basis establishes the position and contribution area of this project, and also provides a foundation for this work. Our aim is to introduce and contextualize subjects related to structural testing and concurrent programs, before presenting related and previous works.

PREVIOUS WORK

3.1 Initial considerations

This chapter presents the previous works that have been developed by our research group at ICMC/USP. Most of the contributions are related to the TestPar project, which is introduced in Section 3.3.

Section 3.1 describes methods to test concurrent programs that served as guidelines or complements for test models and tools. Section 3.2 presents the base test model for this project, defining all of the concepts. Section 3.3 explains the main features of ValiMPI tool and its architecture, including the different versions and implementation details that are relevant for the development of this project.

Finally, Section 3.4 presents empirical studies performed on structural testing criteria, showing the methodology followed to validate proposed test models.

3.2 Methods and approaches for testing concurrent programs

This section presents related works on different methods and heuristics for testing concurrent programs, from coverage criteria to structured models. Such works are the base for the definition of the testing model ([SOUZA; SOUZA; ZALUSKA, 2014](#); [SOUZA *et al.*, 2008](#)) that this project is approaching.

A data flow based testing model is proposed in ([SOUZA *et al.*, 2013](#)), focusing on issues for both shared and distributed memory paradigms. The use of static analysis extracts the possible definitions and uses of variables, and then dynamically obtains the information of data flow from an execution file, and controlled executions are used for the verification of path coverage

among processes. The definition of a set of nodes and edges related to the communication and synchronization events allow reducing infeasible edges, which represents a higher cost due to the identification of non-executable required elements. Information about data flow can be extracted by the trace file execution (dynamically), while possible global variable definitions and uses are extracted from the source code (statically). The paths coverage of each processes is evaluated using post-mortem methods and controlled executions.

Some new concepts are introduced in (SOUZA *et al.*, 2013), d-def represents a pair of node and data, such that data is definitely defined in the node (has a definite definition). i-def is given by the pair of a thread node and data definition, where data is possibly defined in the node (indefinite or possible definition). Other concept that differs from earlier models (SOUZA *et al.*, 2008) is the message use or m-use, which consists of sending a message with data among threads, whether they belong to the same process or not (SOUZA *et al.*, 2013). The study defines different def-use associations to establish a new set of criteria, furthermore, criteria for non-blocking primitives is considered as well.

A methodology to extract information from concurrent programs for structural testing is proposed in (PRADO *et al.*, 2015), considering method calls, pointer manipulation and different communication and synchronization paradigms. The study bases on the analysis of concurrent Java programs, considering that a thread executed a specific class method. The PCFG represents the concurrent program and the trace file generated after program execution describes dynamic aspects. Both static and run-time information should be congruent to evaluate the coverage of required elements (PRADO *et al.*, 2015). To reduce the size of the GFC for each thread, each method call or block of continued statements are mapped into a single node, named *partial node* (PRADO *et al.*, 2015). The instrumentation process interprets synchronization and communication primitives to the testing model information, so that the testing tool ValiPar generates a synchronization flow that is reproducible statically and dynamically (at run-time).

Mutation testing has also been employed for concurrent programs, in (SILVA, 2013), an analysis of message-passing programs and MPI interface motivates the proposition of mutation operators for testing parallel applications. This methodology can be appropriate when evaluating applicability and effectiveness of testing criteria, as well as to provide diverse testing techniques to reveal concurrency errors (SILVA, 2013). Both point-to-point and collective communication patterns are considered, and each mutation operator is classified according to the type of fault that can possibly appear in the parallel program when applying them. Switching the arguments in an MPI function, replacing send/receive modes, replacing collective function, and removing wait or test functions are some examples of the proposed operators.

Given the importance and the lack of concurrent benchmarks for empirical validations, (DOURADO, 2015; DOURADO *et al.*, 2016) presents a benchmark suite of Java message-passing programs to support testing criteria and tools. Most of the developed benchmarks are relatively small size programs of simple problems, which main focus was the complexity of

communications among processes, the implementation for both distributed and shared memory paradigms, the utilization of different communication patterns and test cases with documentation (DOURADO *et al.*, 2016).

The identification of infeasible paths is not an easy task, therefore, (SOUZA *et al.*, 2013) presents a method to identify infeasible paths for message-passing programs by defining patterns that help figuring out if a path has inconsistent characteristics that could make an execution non-executable.

(DOURADO, 2015) defines some features that should be accomplished by a program to be part of a suite of benchmarks, and considerations to take when employing them. The understanding, supported execution, and outputs legibility of a program facilitate its implementation in any environment. The level of interactions among processes and the Cyclomatic Complexity determine the complexity on communications and thus the possibility to represent real scenarios. The definition of test cases can be convenient when comparing program execution results. A program's execution environment, portability and flexibility are related to the level of adaptation of the program to different environments, flexibility in the number of processes and architecture platforms respectively. The scope of communication and synchronization primitives relates to the consideration of both memory paradigms and communication patterns (point-to-point or collective). Standardized code, documentation and pseudo-code provide guidelines and comprehensive features of a program, since it is important to maintain updated documentation of the benchmark suite. Event based vision and non-determinism are useful characteristics when verifying the impact of different concurrent program executions, comparing testing criteria, methods for coverage evaluation and controlled execution mechanisms. Finally, as the main intention of grouping a set of programs is to evaluate them in a test model or tool, having known injected defects (in a structured and controlled form) allows the verification of criteria effectiveness and comparison of strength.

According to (DOURADO, 2015), it is expected that benchmarks adhere to such features, and having a higher level of accomplishment provides more robust contributions when evaluating testing tools, as well as diminishing impact on validity threats.

3.3 Structural testing of message-passing programs

This section introduces the main concepts of the test model (SOUZA; SOUZA; ZALUSKA, 2014) that supports our proposal. These definitions are important for the development of the test model extension we intend to associate for structural testing criteria of message-passing programs, presented in Chapter 6.

The main objective of (SOUZA; SOUZA; ZALUSKA, 2014) is to contemplate non-blocking send and receive primitives, collective communication patterns, and persistent operations. We are introducing in this Section the concepts that are directly involved with the definition

of our proposed testing criteria based on loop execution.

A concurrent program is determined by a set of n processes $Conc\_Prog = \{p^0, p^1, \dots, p^{n-1}\}$, with a Control Flow Graph CFG^p representing each process p . A parallel program can also be represented as a Parallel Control Flow Graph $PCFG$, composed by all CFG^p . Each CFG^p has a set N^p of nodes and E^p of edges of the graph. There are also two subsets N_s and N_r of N^p that define the nodes that have a send (N_s) and a receive (N_r) primitives. The test model (SOUZA; SOUZA; ZALUSKA, 2014) defines a set N_{sync} that collects the previously mentioned sets in order to avoid replication in the send and receive nodes. Thus, N_{sync} represents the nodes $n_i^p \in N^p$ that have a communication primitive (blocking or non-blocking, and collective or point-to-point). The nodes from a CFG^p have the notation n^p .

Intra-process edges are edges in the same process, while inter-process edges are edges among different processes. Inter-process edges are defined as a set E_s that contains edges representing communication between two processes, such that E_s can be represented by a tuple (n^{p_a}, m^{p_b}) , a communication edge, $n^{p_a} \in N_s$, and $m^{p_b} \in N_r$.

A definition of a variable x happens when its value is stored in a memory position in an assignment statement, input command or output parameters as a function reference (SOUZA; SOUZA; ZALUSKA, 2014). A receive primitive is considered a definition as it sets a value of a variable to a new value. The set of variable definitions at node n^p is $def(n^p)$.

Variables also have occurrences in a program when there are references to their values, known as *uses*. The three types of uses are:

- *Computational use (c-use)*: refers to a computation statement of the variable and relates to a node in the CFG.
- *Predicate use (p-use)*: happens when a variable is used in a conditional predicate, and relates to the intra-process edges of the CFG^p .
- *Communicational use (s-use)*: refers to the uses in a communication primitive, specifically the inter-process edges from a node $n^{p_a} \in N_s$ to a node $n^{p_b} \in N_r$.

An association def-use is a pair comprised by the definition of a variable and its consequent use. The path must be a clear definition path, meaning that from the definition of a variable x to a use there can not exist any other re-definitions of x . In other words, a path $\pi = (n_1, n_2, \dots, n_j, n_y)$ is clear definition with regarding x from n_1 to n_y or edge (n_j, n_y) , if $x \in def(n_1)$ and $x \notin def(n_i)$, for $i = 2 \dots j$ (SOUZA; SOUZA; ZALUSKA, 2014; SOUZA et al., 2008). There are three types of associations:

- *c-use association*: corresponds to the definition of a variable x in a node n^{p_a} and the computational use in a node m^{p_a} with a clear definition path.

- *p-use association*: corresponds to the definition of a variable x in a node n^{p_a} and the predicative use in an edge (m^{p_a}, w^{p_a}) with a clear definition path.
- *s-use association*: conforms the definition of a variable x in a node n^{p_a} and the communicational use in an edge (m^{p_a}, w^{p_b}) with a clear definition path.

There are two more types of associations proposed in (SOUZA *et al.*, 2008) that considers the presence of a c-use and/or p-use from the perspective of two processes, having in mind the analysis of data flow of a variable x that is sent from a process p_a and has a use in the receiving process p_b :

- *s-c-use association*: is given by the definition of a variable x in a node n^{p_a} with a s-use (m^{p_a}, w^{p_b}) , and a c-use in v^{p_b} .
- *s-p-use association*: is given by the definition of a variable x in a node n^{p_a} with s-use (m^{p_a}, w^{p_b}) , and a predicative use in (v^{p_b}, z^{p_b}) .

The test model (SOUZA *et al.*, 2008) defines structural testing criteria from the mentioned sets of nodes and edges from the PCFG, and from the def-use associations. Criteria related to control and synchronization flows are *all-nodes*, *all-edges*, *all-nodes-r*, *all-nodes-s* and *all-edges-s*. Criteria for data and communication flows are *all-defs*, *all-defs-s*, *all-c-uses*, *all-p-uses*, *all-s-uses*, *all-s-c-uses* and *all-s-p-uses*. Note that this family of criteria considers and references sequential associations as well, which were defined in (RAPPS; WEYUKER, 1982).

The model proposed in (SOUZA; SOUZA; ZALUSKA, 2014) presents new criteria for control and communication flows:

- *all-nodes-std*: test sets must execute paths that cover all nodes $n_i^p \in N_{std}$, where N_{std} is a subset of N that contains MPI primitives, including derived data types and dynamic generation of processes (SOUZA; SOUZA; ZALUSKA, 2014).
- *all-nodes-nt*: test sets must execute paths that cover every node $n_i^p \in N_{test} \vee n_i^p \in N_{get\_status} \vee n_i^p \in N_{iprobe}$ at least twice. N_{wait} , N_{get_status} and N_{test} are new sets used to represent nodes that have *wait* or *test* message-passing primitives for the verification of non-blocking MPI requests (SOUZA; SOUZA; ZALUSKA, 2014), while N_{iprobe} represent nodes having primitives to perform non-blocking tests for a message.

Accordingly, testing criteria for data and message-passing flows, as stated in the study:

- *all-nr-uses*: test set must execute paths that cover all the *nr-use* associations, such that a variable x has been defined in a non-blocking receive and its references can be a c-p or s-use in a clear definition path.

- *all-ns-uses*: test set must execute paths that cover all the *ns* – *use* associations, such that a variable *x* has a s-use in a non-blocking send, followed by a c-p or s-use in a clear definition path.

Our proposed test model (Chapter 6) adheres to the extended model as we intend to extend the scope of structural testing criteria for message-passing programs, providing guiding rules for diverse scenarios in parallel programming and particular types of concurrency defects.

3.4 ValiPar testing tool

TestPar is a project dedicated to investigate models and mechanisms to test concurrent programs considering distributed and shared memory paradigms, concerning their inherent non-determinism. The objective of TestPar project is to create, develop and improve models, criteria and tools for structural testing of concurrent and distributed applications programs (PRADO *et al.*, 2015). The project look for developing research on parallel programs testing and quality improvement in order to provide high standards of reliability through such contributions, allowing further investigation to strengthen the knowledge in the area.

TestPar has conceived a testing tool known as ValiPar with versions for message-passing PVM and MPI programs and shared memory programs with Pthreads. An overview of the four main modules and functionalities of the tool, which are common in all versions, are as follows (HAUSEN, 2005; PRADO *et al.*, 2015; De Souza *et al.*, 2005):

- ValiInst: generates the PCFG by extracting static information. It receives as an input the program's source code and the output is the PCFG, information regarding the data flow, flow graphs with deviations, use and definition information and data about message exchange.
- ValiElem: generates the required elements. Receives as input the CFG and other information provided by ValiInst and returns descriptors, required elements and the graph to establish use associations.
- ValiExec: executes the test cases, the input are the test data and the executable program, generating as output the execution trace and the execution paths, number of parallel processes and synchronization sequences of them.
- ValiEval: evaluates coverage for criteria. The input are test criteria, its required elements and the executed paths, to return the evaluation of the code coverage.

The instrumentation of the code is one of the essential functionalities in ValiInst, as it processes and retrieves information from the source code to provide inputs for the other modules. IDeL (SIMÃO *et al.*, 2003) is an instrumentation oriented meta-language that is

instantiated for C in ValiMPI, it takes in the C/MPI code along with the instrumentation semantics description, inserting *check-point* commands to permit the identification of each sentence during the instrumented program execution (with ValiExec) and trace file.

Some studies have adapted ValiPar for the application of new structural criteria, by adding or modifying modules (HAUSEN, 2005; De Souza *et al.*, 2005; SOUZA *et al.*, 2008; SARMANHO *et al.*, 2008).

ValiMPI is the tool that we selected for the implementation and verification of our proposed testing criteria, since it is the most suitable and usable in our research group.

3.4.1 ValiPar versions

As we mentioned before, there are different versions of ValiPar for diverse languages and memory paradigms. This section presents the main contributions of the studies presenting such versions and defined structural testing criteria behind it.

All of the versions have implemented sequential criteria for control flow such as All-nodes and All-edges, and data flow criteria for instance All-C-uses and All-P-uses.

ValiPVM (SOUZA *et al.*, 2008) is a version for PVM (Parallel Virtual Machine) parallel programs in C for distributed memory environments. A characteristic of this version is that has a graphical interface for the user in addition to command line, facilitating testing tasks by visualizing the PCFG. Such set of implemented criteria for control flow (aside from sequential criteria) is: All-nodes-R, All-nodes-S, and All-edges-S. As for data flow criteria: All-S-uses, All-S-C-uses, and All-S-P-uses.

ValiBPel-Web (ENDO *et al.*, 2008) is a tool for structural integration testing for Web Services Composition, as there are few studies focused on testing XML-based programs. The fundamentals of required elements and program representation in ValiPar are also applied in this work, mentioning extended concepts of program graphs. The implemented criteria for both control and data flow are All-nodes-R, All-nodes-S, All-edges-S, All-S-uses, All-S-C-uses, and All-S-P-uses.

ValiPthreads (SARMANHO *et al.*, 2008; SARMANHO, 2009) is a tool that focuses on multi-threaded programs. Proposes a new model for program representation $PCFG_{sm}$, standing for Parallel Control Flow Graph for Shared Memory, from the necessity of differentiating the communication and synchronization among threads rather than processes that have their private own memory. Another important aspect for shared memory programs is the initialization and finalization of threads in program execution, defined by semaphores in the $PCFG_{sm}$ model. The test model contemplates different concepts for associations, as well as for sequential-based criteria. For example, c-use and p-use are defined for statements in local variables, whilst synchronization uses (named sync-uses) refer to statements on semaphore variables. For defining criteria, authors consider *post* and *wait* primitives. In addition, communicational c/p-uses relate

to shared variables associations among threads. Control flow (in addition to sequential criteria) and synchronization-based criteria defined and implemented in ValiPthreads include: All-p-nodes, All-w-nodes, and All-s-edges. As for data flow and communication criteria: All-def-comm, All-defs, All-comm-c-use, All-comm-p-use, and All-sync-use. A post-mortem method based on shared variables communication timestamps is used to find out the order of occurrences after program execution.

ValiPar Service ([PRADO et al., 2016](#)) is oriented to the structural testing of concurrent programs as a Web Service composition. The study proposes the tool to improve testing activity cost (in terms of response time), performance, availability and re-usability, providing contracts to customize service compositions. An experimental evaluation validates the conclusions as a result of a comparison among the Desktop and Service versions of ValiPar. Centralization allows the use of clients in web browsers and other command line interfaces.

ValiErlang ([OLIVEIRA; SOUZA; SOUZA, 2016](#)) is a tool that provides structural testing solutions for functional language programs. As the semantics and paradigm of functional language-based programs are different from imperative programming languages, new structural testing criteria is necessary before implementing a tool. The study considers Erlang elements such as functions, communication and synchronization among processes, concurrency and recursivity to define testing criteria. Some limitations of the first version of the tool are the lack of specifications to control synchronization pairs execution, and the absence of fault-tolerance mechanisms.

3.4.2 ValiMPI

In this section we take a close look at the functionalities of ValiMPI. We make necessary emphasis on modules characteristics and their implementation details, in order to establish a guide to introduce the modifications we developed (described in Chapter 7).

The code source version of ValiMPI we used is the one described in ([MACHADO, 2011](#)), where a new module named ValiSync is added. ValiSync is based on reachability testing and the main objective is to perform controlled executions of path variants obtained from test case executions. Different synchronizations are generated using the trace files as well, and then are deterministically executed increasing coverage.

ValiMPI had the following testing criteria implemented ([HAUSEN; VERGILIO; SOUZA, 2019](#)), before our modifications: All-nodes, All-edges, All-nodes-R, All-nodes-S, All-edges-S, All-C-uses, All-P-uses, and All-S-P-uses.

ValiInst module

ValiInst implements the IDel ([SIMÃO et al., 2003](#)) instance for C to instrument the source code of the MPI parallel program, creating an instrumented file where MPI functions are

substituted by their correspondent ValiMPI function, and each program code line is numbered. Those functions allow the identification of synchronization and communication events without altering the semantics of the program. For example, ValiMPI_Send substitutes MPI_Send, ValiMPI_Irecv replaces MPI_Irecv, and so on.

A check-point command is inserted for each numbered node into the instrumented code, in order to facilitate the identification of their execution during trace file generation. ValiInst also generates CFG for each process represented in the code by structures, containing information of nodes, edges, variable definition (including definitions in receive primitives) and uses, and communication primitives. The module generates a graph description file, representing CFGs for each function in a DOT file that contains the directed graph nodes and their connections through edges. After creating the DOT file with nodes and edges, ValiInst analyzes the instrumented code and settles into the graph file data flow information, including through comments which node has a variable definition, a c-p or s-use, and where are send or receive primitives. Although non-blocking primitives are instrumented, they are not included in the DOT file.

Figure 2 – ValiInst module

ValiMPI instruments the source code of a program considering the functions executed by an specific process. For example, if we have a program with only main function, only one DOT file will be generated. But if we have a program with a "Master" and a "Slave" function, and only one specific process rank will execute "Master", then there will be a DOT file for each function representing its CFG. Figure 2 shows the ValiInst architecture.

A script Vali-cc does the compilation of the instrumented code to include ValiMPI libraries necessary for program execution. The compiled program is used by ValiExec to run the program with the test cases.

ValiElem module

This module uses the graph information with control and data flows to generate required elements and automaton descriptors (HAUSEN, 2005). Figure 3 shows the ValiElem architecture.

All data retrieved from the DOT files is set into a graph structure, which has a description of the nodes, edges, variables, variable definitions, uses and number of synchronization edges. Then, two graphs are created that retrieve data from the DOT files: reduced heir graph and graph-i. The reduced heir graph intends to minimize the number of required edges in ValiMPI to build the structure representing the CFG.

We are focusing on the graphs-i concept. A graph-i is constructed for all variables defined in a node n_i and the clear definition paths from n_i to a node n_j , so n_j will belong to a graph-i if and only if there is no re-definition of the variable between n_i and n_j . Graphs-i do not contain cycles, so in order to avoid infinite paths when generating the graph-i (caused by loops in the program's GFC), when finding a second occurrence or an image of a node, the path is interrupted and the last node is such second occurrence (or image).

Figure 3 – ValiElem module

Graphs-i are used to establish associations for data flow criteria, while the CFG structure obtained by reading DOT files determines the required elements for control flow criteria. For each required element, ValiElem generates a Deterministic Finite Automaton (DFA, or automaton, as we will refer in this work), which represents a regular expression that describes the path that covers such required element. Automata are implemented by descriptor files, which contains a string that describes the automata states, number of states, transition function, and acceptance states.

The pattern for regular expressions is defined for required elements for each criterion, for example, the descriptor for All-S-edges represents the DFA for the regular expression $N^* n_i-p_a n_j-p_b N^*$, where N is the set of nodes of CFG^{p_b} . The regular expression for All-nodes-R is $N^* n_i-p N^*$, where we can observe there is one process involved. For each criterion there could be one or two automata depending on the regular expression for the required elements.

ValiExec module

ValiExec executes test cases, receiving test data and the instrumented executable program

to generate the trace files representing the executed paths. The execution trace file contains the intra-process paths and communication primitives exercised by every process, using the pattern *node-process* to describe the executed node (as they were numbered in ValiInst) and the process.

Each function is instantiated in ValiExec for the generation of trace files, so there will be a trace file of each function and for each process, depending on which function a determined process is executing (as established in ValiElem).

After executing ValiExec, the user can see the outcome of the program to verify its correctness.

Figure 4 shows the ValiElem architecture.

Figure 4 – ValiExec module

ValiEval module

Coverage evaluation is performed by ValiEval. The module's input is the criterion to evaluate coverage.

ValiEval implements the automata using the descriptor file, with the description of states and transition functions of the DFA. The number of automata described in automaton file for a criterion is determined by the number of required elements. For example, consider All-nodes-R criterion, there will be only one automaton because required elements are represented by n^p . On the other hand, if we have the All-S-uses criterion, two automata will represent an association between two processes $(n_i^{p_a}, n_j^{p_b})$, as the DFA requires execution traces for processes p_a and p_b to retrieve the required elements from both trace files.

The implemented automaton receives and reads the traces files to find the required elements occurrences in the exercised paths, which are represented by the automaton's regular expression. The traces from all functions are processed until an automaton recognizes the required element. In cases where there are two automata for a criterion, two trace files from different processes are involved in the automata's recognition, depending on the definition of the required element.

Coverage percentage in ValiEval is the relation of covered required items over the number of executed elements. Therefore, the number of recognized automata determines how many required elements were covered from the total generated for the criterion. Additionally, ValiEval reads the required elements file to compare the covered elements and present to the user the list of uncovered required elements, so that the tester selects additional test cases to cover such missing nodes, edges or associations. Figure 5 shows the ValiEval architecture.

Figure 5 – ValiEval module

3.5 Empirical evaluation of structural testing criteria

Experimental studies have been performed on previous test models presented in Sections 3.2 and 3.3. In this section, we expose the methodology and results of such empirical studies. The objective of implementing structured experimental methods to evaluate measurements of different dimensions such as application cost, effectiveness, and strength.

An evaluation of cost and effectiveness for structural testing criteria of message-passing programs (SOUZA *et al.*, 2008) is presented in (BRITO; SOUZA; SOUZA, 2013). The study intends to contribute through Software Experimental Engineering methods (proposed by (WOHLIN *et al.*, 2012)) to validate the test model in mention, providing statistical evidence and the possibility of future replication. Furthermore, the study evaluates the behavior and characteristics of criteria (SOUZA *et al.*, 2008) among each other. ValiMPI tool is used to execute the programs evaluating criteria, and the measurements are cost (in terms of number of required elements and test cases), effectiveness and strength. It proposes a strategy to apply criteria, in order to obtain the best cost-benefit relation, as some criteria can be very effective but at a higher cost.

All-s-uses (ASU) criterion shows to be strongest, as it includes other criteria considering their coverage with their own set of test cases. Additionally, ASU criterion is the most effective along with All-edges to reveal concurrency errors, while less effective criteria happen to reveal more specific types of defects.

An empirical evaluation on coverage criteria for concurrent programs (SOUZA *et al.*, 2015) validates computational cost and effectiveness using two techniques for testing message-passing programs: structural and reachability testing. Reachability testing combines deterministic and non-deterministic executions to automatically generate the possible sequences of synchronization events (match between send and receive in runtime) sequences to execute them in a controlled form, to remove non-determinism and avoid building an static model (SOUZA *et al.*, 2015). The testing criteria implemented for the experimental study has been described in (SOUZA *et al.*, 2008). Structural testing criteria is used to select test cases and to determine new synchronizations to execute, while reachability testing is used to establish which synchronizations should be exercised. A set of test cases is created for each criterion until obtaining total coverage if possible. Then, non-executable required elements are identified manually (SOUZA *et al.*, 2015). On a first instance, to execute different synchronizations, test cases are replicated several times. However, such repetitions do not ensure that other paths are exercised. Reachability testing supports the selection of new feasible synchronizations, providing a controlled execution solution to increase coverage. Using both approaches reduces computational cost without affecting criteria effectiveness.

A structured experimental study for shared memory programs and structural testing criteria for such programs (SARMANHO *et al.*, 2008) is presented in (MELO *et al.*, 2017). It is also based on the Software Experimental Engineering method by (WOHLIN *et al.*, 2012), using similar experimental design as (BRITO; SOUZA; SOUZA, 2013) concerning metrics, independent and dependent variables. The results show that concurrent criteria causes a higher application cost in relation to sequential criteria, whilst having better effectiveness when revealing defects. Concurrent criteria (CC) tends to include sequential criteria (SC), as CC is capable of covering required elements from SC revealing exposing sequential-based defects. (MELO *et al.*, 2017) also proposes a strategy to apply CC and SC criteria cooperatively.

The diversity of empirical studies regarding their design and statistical methods applied, establishes a new research line for concurrent programs testing focusing on different objectives, as it is important to validate proposed theories and test models to employ them in real applications and situations. Also, experimental validations on testing approaches and tools for concurrent programs are incipient. However, there are similarities regarding the procedures to select the programs to test, inject defects into the code, and the measurements used to evaluate proposals.

3.6 Final considerations

Different testing techniques have been approached to test concurrent programs with both shared and distributed memory paradigms, mainly structural and defect-based testing. We also introduced the testing model for message-passing programs that this project bases on. The concepts and definitions are important for the description of our test model. The described benchmarks characteristics are also relevant for the experimental design of this project.

ValiMPI tool has been a major contribution in the research group to implement and validate test models from different perspectives, languages and paradigms. The versions of ValiMPI provide diversity in the types of contributions of the studies, maintaining the focus of proposing methods to improve the quality of concurrent programs. We acknowledge the importance of experimental studies to validate the viability and potential impact of structural testing criteria. Additionally, the execution of empirical studies is one of the contributions of this project.

RELATED WORK

4.1 Initial considerations

In this chapter we expose related studies in area, found in the systematic mapping performed for this project. Section 4.2 presents related works regarding the selection of paths for loops, and other approaches for testing concurrent programs in Section 4.3 including testing criteria, which were analyzed previous to our test model proposal. Section 4.4 describes testing tools for concurrent programs.

4.2 Loop path selection

There are studies oriented to the selection of paths inside loops to generate test data sets. Some of the proposed methods have been considered for this project in criteria implementation. This section presents main contributions of such studies.

Authors in (WANG; HUANG; GONG, 2013) address the problem of finding forms to generate test cases that consider more than one iteration of a loop, stating that this might decrease the accuracy of tests. Their objective is to generate partial paths from the input node of the CFG to the partially unfolded loop path to generate test cases, and ensure that such path is executable. A *sub-path* is defined as a sequence of nodes that are executed in a loop. When testing only one iteration of a loop, it is named *z-path* coverage. This work proposes generation paths formed by a *prePath*, which goes from the initial node to the loop, and then one of the *sub-paths* (must be feasible). Then, an *integrity* path is formed by *prePath*, the current path of the loop and the path from the end of the loop to the last node of the program. The unfolding process consists of looking for all the possible paths (*prePath* and *sub-paths*) that are the target at least once.

Another study that focuses on the challenge of loop awareness in testing concurrent programs (SIEGEL; ZIRKEL, 2012), uses symbolic execution considering loops, as they produce

an infinite number of reachable states. Authors in (SIEGEL; ZIRKEL, 2012) adapt the solution known as loop invariant symbolic execution originally proposed for sequential programs, to be implemented in parallel programs. The main challenge are loops that span other processes, which introduces a notion of collective loop invariant and a symbolic execution method to verify assertions in message-passing parallel programs with unbounded loops.

A loop-aware Concolic execution as an approach for software testing is proposed in (LI; SHIEH, 2011), obtaining more information of loops to increase the effectiveness by eliminating unnecessary testing. The proposal addresses issues of security and vulnerability in applications. (LI; SHIEH, 2011) considers concrete execution as a black box technique and symbolic execution as a white box technique. Symbolic execution symbolizes each input variable of the program, and the values of the variables are constrained to the conditional branches. This approach is used to predict inputs according to specific constraints in the paths. It also requires more effort, is not very scalable and allows a more straightforward coverage tracking. The objective of Concolic execution is to remove unnecessary testing by obtaining more information about loops. Concolic execution combines both approaches (LI; SHIEH, 2011), relying on the efficiency of concrete execution and the wide coverage of symbolic testing. Inputs are represented as symbolic variables and operations are executed in values that depend on such symbolizations, generating a set of specific input values to be used as input for concrete execution. A set of branch constraints will be generated by each concrete execution. Concolic execution is used to analyze variables related to loops, establishing data dependencies for data and control flows. One of the issues mentioned in (LI; SHIEH, 2011) is that Concolic execution may not detect the relationship between some variables depending on control flow. Thus, loop-aware Concolic execution captures dependencies between loops and variables related to program input. Concolic execution can be divided into four steps: static analysis in the binary executable, generation of initial input, Concolic execution (concrete and symbolic simultaneously), and path constraint solving. Their approach is more efficient when having input-related loops, meaning that at least a variable in the loop structure is related to the input.

4.3 Approaches for testing concurrent programs

In this section we discuss different approaches from structural techniques to test concurrent programs, which focus mainly on static and dynamic analysis methods. Such approaches address diverse issues and concerns when testing concurrent programs, such as testing criteria, controlled execution, test data generation and infeasible paths.

Temporal testing is an approach to treat non-determinism, using non-deterministic executions along with path testing to detect synchronization defects, locating delays in the execution time within the du-paths (definition-use paths) to test (YANG; SOUTER; POLLOCK, 1998). The aim of the study is to automatically generate test cases adopting the all-du-path

criteria. A du-path is admissible if it contains communication commands from its definition to its use, and such paths are generated statically. The program is executed several times, and then the trace results are evaluated considering that if different paths were exercised there is a synchronization error. Non-deterministic testing is also presented in (BECHINI; TAI, 1998) along with deterministic testing, which selects a set of tests with synchronization sequences and forces a deterministic execution with an input to evaluate the feasibility of the path. This study bases on synchronization events sequences to characterize the execution of a concurrent program.

A methodology to generate test data and test cases is proposed in (YANG; POLLOCK, 1999), by using both static and dynamic approaches. A static model extracts information from code, being this crucial for path coverage, and the dynamic approach generates the test cases based on the criteria. These techniques complement each other to provide a solution for path coverage, given that static analysis is insensitive to distinguish the inputs of a concurrent program, so it might provoke false positives and false negative since it must make assumptions. Therefore, dynamic analysis helps to uncover concurrency errors during execution (CHEN; MACDONALD, 2008).

(WONG; LEI, 2008) proposes the use of reachability testing to address path infeasibility. Reachability testing bases on the analysis of execution traces, generating sets of synchronization sequences to force them to execute. A reachability graph is proposed in (WONG; LEI, 2008), in which each node is a reachable state and edges are transitions among them, generated by reachability analysis (starts from initial state and then the next step is found). In such graph all paths must be feasible.

Reachability testing for concurrent programs in (GONG *et al.*, 2007) presents a method based on ESYN-sequences, which is the basis for the coverage measurement, including both synchronization events and non-synchronization events. The first method can check errors as events synchronize and verify if they did improperly. The second allows to cancel the constraint condition improving adaptability. Meanwhile, (HWANG *et al.*, 2014) proposes a scheme that automatically derives a set of inputs to guide reachability testing, but without requiring the analysis of the program structure and semantics. Instead of deriving a sequence of path conditions from the sync-sequence obtained with the method, it performs statement-coverage testing. A formal proof demonstrates that there is a statement coverage if there is solution for all path conditions by the constraint solver, and the concurrent program contains no dead code (HWANG *et al.*, 2014).

In (YOGA; NAGARAKATTE, 2016), the focus is on detecting atomicity violations in parallel programs with and without data races in a trace, by designing appropriate meta-data and by leveraging the dynamic execution structure of a task parallel execution. Given a program with atomicity annotations, they propose to maintain a history of dynamic memory accesses performed by various tasks with each shared memory location, and evaluate if those accesses are

conflict serializable (YOGA; NAGARAKATTE, 2016).

The utilization of unit tests is not very common when testing concurrent programs. Instead, data race detectors are usual, even when commonly presenting many false positives. Schimmel et al (SCHIMMEL *et al.*, 2015) combines both methods to test parallel programs, by executing the program with a data race detector instead of a conventional test runner. The paper presents an automatic generation of parallel unit tests through existing unit tests, which are the input of a data race detector containing parallel test case assertions as well.

Detecting infeasible paths is an expensive task for the testers. Vergilio et al (VERGILIO; MALDONADO; JINO, 2006) approaches this subject and explains the impact for the cost of the data flow testing when there is a significant number of infeasible paths to consider. The paper provides the classification, prediction and identification of these elements in unit and integration tests. Thus, it shows strategies and patterns that should be avoided, even at the beginning of the software life cycle and development.

4.3.1 Testing criteria for concurrent programs

This section presents findings on testing criteria for concurrent programs. Despite they are mostly based on heuristics rather than structured test models, their contributions are relevant for concurrent programs quality and reliability. Additionally, for the definition of our testing model we acknowledge the current state of the art on testing criteria besides previous works.

(MARCOZZI *et al.*, 2017) introduces a framework to describe and compare testing criteria, since most tools support a small set of all the existing criteria. Their objective is to enable the implementation of generic tools. The authors define a generic specification language named Hyperlabel Test Objectives Language for white-box test requirements, in order to encode a wider set of existing criteria. It also proposes a new taxonomy for the coverage criteria basing on reachability constraints.

Another study defines a family of criteria that uses a different representation of the program for coverage testing. Takahashi et al (TAKAHASHI; KOJIMA; FURUKAWA, 2008) defines coverage criteria for concurrent programs, basing on a representation named Concurrent Module Flow Graph (CMFG), where modules are threads and each module has a finite set of blocks. Some of the defined criteria is all-concurrent-path (based on Rapps and Weyuke's all-path criteria), showing a formula to define the necessary number of test cases to satisfy this criterion. Another criterion is All-concurrent-binomial-path, and it aims to reduce the number of test cases, by verifying only two unique block paths instead of all paths between start and end of the module. The study also relies on ConTest and Eraser tools for dynamic testing for race detection.

Coverage criteria for sequential programs cannot cover order of operating shared resources. All-Concurrent-Paths (ACP) is a criterion proposed in (KOJIMA *et al.*, 2009), from a Concurrent module flow graph (represents relation among concurrent statement and its order

of execution), in which a node represents a block. A block divides source code by operation of shared resources. A path in the graph represents an order of execution of blocks in a module, or the order of operating shared resources. It intends to make a reduction of test cases and measurements of criteria in a tool. The study (KOJIMA *et al.*, 2009) refers to the control flow criteria proposed by Rapps and Weyuker (all-paths, all-edges, all-nodes), making analogies between all-paths to the defined all-concurrent-paths and all-edges with their all-concurrent-binomial-path. ACP replaces nodes with blocks. Authors define defects based on data races and conditions between EOB (external operation block): (1) Wrong lock or no lock. Shared resources are not locked in other threads. Maskable and non-maskable EOBs operate the same shared resources at least once but are in different threads. (2) Two stage access. It is possible to change values of shared resources by a thread interrupting between two EOB blocks (that belong to the same thread). (3) Non atomic operation assumed to be atomic. There is a non maskable EOB. After implementing criteria, (KOJIMA *et al.*, 2009) concludes that defects depending on timing can be detected because ACP can cover all combinations of EOB.

Based on concurrency fault models, (LU; JIANG; ZHOU, 2007) proposes testing criteria for interleaving coverage of concurrent programs. The study describes the following criteria: All-interleavings, Thread-pair-interleavings, Single-variable-interleavings, Partial-interleavings, Local-or-remote-define, Local-or-remote-interleave. Authors discuss a cost analysis for the hierarchy of criteria considering measurements based on number of interleavings.

Table 1 summarizes the contributions of the related work, showing the followed *Approach* by the studies. The *Criteria definition* column determines studies that define criteria, however, the studies that propose testing criteria do not follow structured standards to systematically select test cases. In other words, a test model is not defined contemplating sets and associations concepts to generate required elements. As one of the concerns in related works is test data generation for concurrent programs, we expose which studies focus in techniques to create test cases. The *Type of contribution* refers to the main contribution proposed by each study, according to their own conclusions. Note that symbol "X" means non compliance and "√" represents compliance.

The Table 1 allows to visualize the lack of studies that defines testing criteria for concurrent programs, contemplating dynamic aspects while using structural techniques. Our project is focused in addressing such gaps, as is discussed in this work in Chapters 6, 7, and 8.

4.4 Testing tools for concurrent programs

There are different testing tools mentioned on concurrent software testing literature, for both memory paradigms and for diverse programming languages. In this section we present a compilation of such tools to compare and illustrate their main features.

Table 1 – Main characteristics of related work studies

Work reference	Approach	Criteria definition	Test data generation	Type of contribution
(SIEGEL; ZIRKEL, 2012)	Loop-aware/Symbolic execution	X	X	Technique
(WANG; HUANG; GONG, 2013)	Static analysis	X	X	Method for loop path selection
(LI; SHIEH, 2011)	Loop-aware/Concolic execution	X	X	Technique
(CHEBARO <i>et al.</i> , 2014)	Dynamic and static analysis	X	X	Tool prototype
(YANG; SOUTER; POLLOCK, 1998)	Temporal/Structural testing	X	✓	Method for path selection
(BECHINI; TAI, 1998)	Deterministic testing	X	X	Tool-set design
(YANG; POLLOCK, 1999)	Dynamic and static analysis	✓	✓	Data extraction technique
(WONG; LEI, 2008)	Reachability testing	X	✓	Test sequence generation technique
(GONG <i>et al.</i> , 2007; HWANG <i>et al.</i> , 2014)	Reachability testing	X	X	Method
(SCHIMMEL <i>et al.</i> , 2015)	Dynamic and static analysis	X	X	Unit tests generation
(VERGILIO; MALDONADO; JINO, 2006)	Structural testing	X	X	Infeasible paths prediction
(MARCOZZI <i>et al.</i> , 2017)	White-box testing	X	X	Framework to compare testing criteria
(YOGA; NAGARAKATTE, 2016)	Dynamic and static analysis	X	X	Atomicity violation detection technique
(TAKAHASHI; KOJIMA; FURUKAWA, 2008)	Coverage-based testing	✓	X	Method
(KOJIMA <i>et al.</i> , 2009)	White-box testing	✓	X	Method
(LU; JIANG; ZHOU, 2007)	White-box testing	✓	X	Coverage criteria hierarchy
(LOURENÇO <i>et al.</i> , 1997)	Structural testing	X	X	Tool integration prototype
(SANTHIAR; KALEESWARAN; KANADE, 2016)	Event-loop based testing	X	X	Technique for race detection
(DIAZ, 2019)	Structural testing - Loop aware criteria	✓	-	Test model

Dellapasta (Delaware Parallel Software Testing Aid) is a tool developed to analyze concurrent paths for shared memory programs (DELAMARO; MALDONADO; JINO, 2007). This tool has been used for verifying all-du-path coverage criteria (YANG; SOUTER; POLLOCK, 1998), but one of its limitation is that it not suitable for distributed memory programs (SOUZA *et al.*, 2008). CHESS uses model checking to cover different interleavings (PRADO *et al.*, 2015) for .Net parallel programs, while ConAn (Concurrency Analyzer) is utilized for unit testing of concurrent programs (DELAMARO; MALDONADO; JINO, 2007) basing on model testing (ADHianto *et al.*, 2010) as well. ConAn was implemented for a technique to test java interrupts in another study (WILDMAN; LONG; STROOPER, 2004). Wildman et al (WILDMAN; LONG; STROOPER, 2004) presents a method to test Java interrupts and waits using the ConAn tool, achieving statement coverage of the program. The authors verified the technique by defining points of the source code in relation to wait calls, for non-arbitrary interrupts.

Table 2 – Testing tools present in the selected studies

Reference	Tool	Approach	Language
(DELAMARO; MALDONADO; JINO, 2007)	Dellapasta	Shared memory	Pthreads
(PRADO <i>et al.</i> , 2015)	CHESS	Shared memory	.Net
(PRADO <i>et al.</i> , 2015)	Contest	Shared memory	Java
(BRUEGGE; GOTTSCHALK; LUO, 1993)	Bee++	Object-oriented distributed programs	C/C++
(KOSMATOV <i>et al.</i> , 2012)	PathCrawler	Concolic execution	C
(LONG; STROOPER; WILDMAN, 2007; WILDMAN; LONG; STROOPER, 2004)	ConAn	Concurrent analysis for Java components	Java
(VO <i>et al.</i> , 2010)	DAMPI	Dynamic formal verification	C/C++/Fortran (MPI)
(LOURENÇO <i>et al.</i> , 1997)	STEPS	Structural testing for message-passing programs	C (PVM)
(LOURENÇO <i>et al.</i> , 1997)	DDBG	Debugging engine	C (PVM)
(CHEBARO <i>et al.</i> , 2014)	SANTE (Frama-C and PathCrawler)	Static and dynamic analysis	C
(ARTHO <i>et al.</i> , 2005; PERSON <i>et al.</i> , 2011)	Java Pathfinder	Test case generation with symbolic execution	Java
(TERRAGNI; CHEUNG, 2016)	AutoConTest	Coverage driven	Java
(ZHAO <i>et al.</i> , 2006)	Adept	Dynamic execution profiling for shared memory programs	C
(POLURI; RAMANATHAN, 2015)	STABLER	Deterministic dynamic race detection for multithreaded programs	Java
(SANTHIAR; KALEESWARAN; KANADE, 2016)	SparseRacer	Race detection	C/C++
(SIEGEL; ZIRKEL, 2012)	TASS	Tool suite for symbolic execution-based verification	C (MPI)
(FLANAGAN; FREUND; YI, 2008)	Velodrome	Dynamic analysis for atomicity checking in multithreaded programs	Java
(MELO <i>et al.</i> , 2017)	ValiPthread	Structural testing	C (Pthreads)
(YU; SRISA-AN; ROTHERMEL, 2017)	SIMEXPLORER	Dynamic analysis for race condition	C/C++
(ZHANG <i>et al.</i> , 2017)	STEM	Data flow testing	C (Pthreads)
(CHEN <i>et al.</i> , 2009)	HAVE	Dynamic and static analysis - atomicity checking for shared memory programs	Java
(DESOUZA <i>et al.</i> , 2005)	Intel Message Checker	Structural Message-passing programs	C/MPI
(SOUZA <i>et al.</i> , 2008)	ValiPar/ValiMPI/ValiPthreads	Structural testing shared and message-passing programs	Java/ C(PVM,MPI)/ Pthreads

Contest is a tool for Java multithreaded programs, which executes possible thread schedules to find concurrency bugs (PRADO *et al.*, 2015). A proposal in (LETKO, 2010) intends to use Contest to combine dynamic analysis along with search based techniques. The Automated Concurrent Testing (AutoConTest) is a method to automatically generate and run concurrent tests for a given class (TERRAGNI; CHEUNG, 2016). The generation of such tests bases on the generation sequences of threads, joining them into multi-threaded tests. (TERRAGNI; CHEUNG, 2016) proposes a coverage metric, a greedy algorithm to explore search space of possible method call sequences, and a thread scheduler algorithm to increase the context-sensitive interleaving

coverage with a prototype tool.

STEPS focuses on the debugging of distributed memory concurrent programs (DE-LAMARO; MALDONADO; JINO, 2007), and it is used in a debugging environment in (LOURENÇO *et al.*, 1997).

HAVE is an atomicity checker for concurrent programs that integrates dynamic and static analysis (CHEN *et al.*, 2009), since it found an issue when finding atomicity violations using traditional testing methods. Authors present an architecture of tool, with a static analyzer, code instrumenter and a dynamic monitor and speculator. The source code is translated into Static Summary Trees, which represent different structures and occurrences of the program. For loop execution, the study establishes that executing a loop twice is enough to reveal atomicity violations, when every iteration executes the same access operations.

Some aspects of MPI programs can be verified statically: the syntax of the functions, the types of data, and the number and type of arguments. However, to identify deadlock situations it is required to be done dynamically. (DESOUZA *et al.*, 2005) proposes Intel Message Checker, a tool that verifies the correctness of programs in MPI.

Table 2 shows some of the tools for structural testing of concurrent programs that have been cited in different studies, relating them with the synchronization and communication paradigms and languages that they were implemented for.

4.5 Final considerations

The related studies allow to reinforce the gap in the definition and implementation of structural testing criteria for parallel programs that considers the execution of loops. Some contributions are based on testing heuristics, such as the selection of test cases through the knowledge of the tester, or based on dynamic and static analysis techniques. However, there are studies that utilize different techniques to generate test data, such as symbolic and Concolic execution.

We also described tools to test concurrent programs for different approaches and programming languages.

CONCURRENT DEFECTS TAXONOMY

5.1 Initial considerations

This chapter introduces to the classification of types of defects gathered for shared and distributed memory paradigm programs. The Concurrent Defects Taxonomy serves as a base for the identification of errors that our proposed criteria (see Chapter 6) intends to reveal.

Section 5.2 describes the main contributions of studies that focused on concurrency defects, and Section 5.3 presents the Concurrent Defects Taxonomy.

5.2 Preliminary studies on concurrency defects definition

This section presents the main studies considered for the collection and analysis of concurrency types of defects. The following contributions base on analysis of errors, debugging techniques and coverage criteria. We present the description of the defects classifications we find for later unification, in order to provide the definitions of the defects in the proposed complementary taxonomy.

A systematic review presents some studies related to a bug taxonomy (BRITO *et al.*, 2010), which have taken sequential related defects to translate them for concurrent programs. However, such contributions are related to specific research questions formulated by the authors of the review, where the main objective is not to classify all studies on concurrent defects.

A PhD thesis studies design patterns to deduce patterns on concurrency errors (BRADBURY, 2007):

- Non-atomic operations assumed to be atomic bug pattern.
- Two-state access bug pattern (sequence of unprotected operations).

- Wrong lock or no lock bug pattern (a code segment is protected by a lock, and other threads get another different or no lock).
- Double-checked lock bug pattern.
- The *sleep* bug pattern (using *sleep* function to wait for threads to finish, it is recommended to use *join* instead).
- Losing a *notify* bug pattern (when a *notify* is executed before its corresponding wait, and the notify is lost).
- A "blocking" critical section bug pattern (it is assumed that a thread will return to the lock but never does).
- The orphaned thread bug pattern (for example, a master process ends abnormally and the children are left waiting).
- *Notify* instead of *notifyall* bug patterns (some threads with waits are not notified).
- The interference bug pattern (can be generalized with the usual data-race interference, when write access is not specified to a shared variable that can not be modified simultaneously).
- The deadlock (deadly embrace) bug pattern (two or more processes can not continue with their execution because each one is waiting for the other to do something).
- Other missing or nonexistent signals (generalizes the "Losing *notify*" pattern but including a barrier with a *wait*, if a thread does not execute it, all threads stay waiting).
- Starvation bug pattern (the usual case of CPU escalation policies, adding the case of unfair lock acquisition).
- Resource exhaustion bug pattern (a thread runs out of resources because no other thread releases its resources).
- Incorrect count initialization bug pattern (when the number of allowed threads in a semaphore is not defined).

This study ([BRADBURY, 2007](#)) is based on mutation testing, using the exposed patterns to generate concurrency mutation operators.

To solve concurrency problems, ([LOURENÇO; CUNHA, 2007](#)) based on Software Transactional Memory - STM, which is a similar mechanism to database transactions. Such approach intends to reveal errors concerning transactional operations, allowing defining more easily which regions of code must be atomic.

According to ([LU et al., 2008](#)) conclusions, various techniques related to concurrent programming benefit from an in-depth knowledge of the characteristics of concurrency errors. Most programmers think sequentially and that causes the insertion of defects in concurrent programs, adding to this situation, non-determinism makes difficult the identification concurrency errors due to the replication limitations. To write effective test cases, it is necessary to know what

conditions are required to trigger a concurrency error (how many threads, variables, accesses, etc). (LU *et al.*, 2008) makes emphasis in that real world concurrency errors are not proven or tested, given that it is very difficult to characterize them and it is time-consuming to report them due to non-determinism and therefore difficult replication. Also, concurrent programs are more difficult to understand. The bug characteristics found in (LU *et al.*, 2008) are strongly linked to their methodology and to the programs they selected: real large applications such as Open Office, Apache, Mozilla and MySQL. Authors take the database of reported errors and look for those related to concurrency. It can also be limited to client-server and C applications, since apps in Java or operating systems were not examined. (LU *et al.*, 2008) presents characteristics and categories regarding the pattern of errors, manifestation of errors and correction of errors. (LU *et al.*, 2008) separates two types of bugs: deadlock and non-deadlock bugs. Data race is not classified as a pattern since there also exist "benign" data races. Most errors are manifested by interactions between two or a small group of threads, because they usually do not interact closely with others. Some of the deadlock errors are manifested with a single thread, which is when trying to acquire a resource taken by itself. (LU *et al.*, 2008) points out that the study can contribute to the definition of criteria, based on the characteristics of the manifestation of bugs.

(Farchi; Nir; Ur, 2003) defines two types of bug patterns: those that are generated when a design pattern is implemented and when an anti-design pattern is implemented. There are three categories defined:

- Code assumed to be protected. When it is assumed that a code segment is undisturbed by other threads, either explicitly or implicitly.
- When it is assumed that a certain order of execution of concurrent events is not possible.
- When it is assumed that a code segment is non-blocking.

(Farchi; Nir; Ur, 2003) presents examples of such categories, in which the types of defects can be superimposed: (a) Non-atomic operations, (b) Two stage access, (c) Wrong lock or no lock, (d) Double-checked locking, (e) Interleavings that are assumed to never occur (use sleep, losing a notify, thread blocked or dead, critical section blocking, and the orphan thread).

(CHOI; LEWIS, 2000) performs a study with students using the Eraser tool, identifying in three categories the common mistakes that insert errors when coding concurrent programs:

- Data races: when multiple threads read and concurrently write the same data. Some defects in this category are:
  - Shared variable not protected by a lock. By not knowing what data should be shared.
  - Lock not acquired to protect access to a region, even if it existed. For example, when a lock is acquired outside of a loop but is released inside.
  - Accidental sharing, when a variable is shared in all threads without having a lock.

- A program prematurely releases a lock. There is no certainty as to what types of accesses need to be protected by a lock.
- Redundant or unprotected initialization of shared variables.
- Multiple locks to protect a shared variable. Different locks used for reading and writing, and therefore there is no synchronization between the threads performing such operations.
- Deadlock: when the program fails to continue because a thread is waiting for a condition that will never occur.
  - A signal is executed before the *wait*.
  - Unrelated locks used to protect a single shared variable.
  - Primitives test-and-set mistaken with mutexes at the application level.
  - Locks that are never released.
  - Threads try to get back locks they already had.
- Miscellaneous.
  - Unnecessary use of interrupt disablings.
  - Acquisition and release of locks.
  - Use of a single lock to protect all shared data.
  - Unlock the same lock repeatedly.
  - Assume that the unlock-and-stop routine returns with the lock test-and-set re-acquired.

To propose debugging techniques (DESOUZA *et al.*, 2005) focuses on distributed memory errors in MPI programs. It considers three types of errors:

- Synchronization errors (deadlocks or data races).
  - Deadlocks: when a dependency between two tasks may never be satisfied. For example, a blocking send that does not return until the matching receive has been posted. There could be two types:
    - * Timing-dependent deadlock: when the program is non-deterministic due to wildcard receives.
    - * Standard deadlock: it does not have non-deterministic dependencies.
  - Data races:
    - * Interface data race: handles the interface between the process and the call to an MPI primitive. The application can change the contents of the buffer before it is completely sent.
    - * Inter-process data race: two processes try to send the same logical message.

- Matching errors. It can refer to the parameters of the MPI functions or matches between sender and receiver.
  - Call type: type of call to the message passing function.
  - Argument: type of arguments.
  - Length: length of the message.
- Resource errors (such as derived data types or communicators).
  - Allocation: pointers not initialized because data has not been allocated.
  - Initialization: non-initialized data, simple non-initialization errors.
  - Deallocation: at the end of the MPI program resources are not reallocated.

([PEDERSEN, 2006](#)) support the assumption that concurrency bugs might appear on early stages, and describes how in the PCAM (Partitioning, Communication, Agglomeration, Mapping) process to design programs, bugs can be put in the code. Issues on the first two stages might appear in:

- Data decomposition: when doing the decomposition of the serial version of a program to the parallel version.
- Functional decomposition.
- Use of the API: related to calls to MPI APIs, for example, passing data of the wrong type or not understanding how the functions operate.
- Sequential error: common errors of sequential programs.
- Message problem: concerning the content of the messages, not the protocol. For example, wrong data is sent or received.
- Protocol problem: it has to do with the protocol, for example, loss of messages.
- Others: issues that do not belong to the previous categories, for example, wrong permits in a spawn, faulty parallel IO.

Agglomeration and Mapping stages are more related to performance issues ([PEDERSEN, 2006](#)), which is not treated in the paper.

Localizing errors in a program involves a process known as debugging ([KRAWCZYK; WISZNIEWSKI; MORK, 2006](#)). Testing reveals the existence of bugs, while debugging locates and removes them. Thus, ([KRAWCZYK; WISZNIEWSKI; MORK, 2006](#)) describes that were previously formalized for sequential programs that also appear in parallel applications: computation error (two programs with the same domain have different computation outcomes), domain error (two programs with the same computation result have different domains) and sub-case error (two programs with the same computation result, the domain of one is contained in that of another).

([KRAWCZYK; WISZNIEWSKI; MORK,](#)) defines two types of errors. The *observability* error, occurs when for an execution path there is another path with the same input domain, but with different computation results. *Locking errors* happen when in a sub-path of the program all edges of the graph correspond to predicates, for which all are evaluated in false for an element of the domain. Locking errors also relate to processes that do not communicate with the correct process, and the incorrect process may have finished its execution so the system may wait for a response.

A systematic review ([ARORA; BHATIA; SINGH, 2016](#)) mentions defects that can cause performance issues in concurrent programs: deadlock, data race, atomicity violation, starvation, livelock, violation of assertion, communication error, synchronization error.

([SAHOO; RAY, 2016](#)) defines three types of concurrency errors, previous to the definition of coverage criteria based on communication and synchronization dependencies in concurrent programs:

- Two phase locking: is a mechanism for concurrency control that is used to guarantee the integrity of data, preventing conflicting concurrent updates between shared objects.
- Serialization graph checking: serializability is the checking of cycles in an Schedule graph, which is divided into aborts. Issues are more oriented to database transactions.
- Timestamp ordering: a timestamp is assigned to each transaction, and controls or checks access to data depending on the timestamp order.

5.3 Concurrency defects taxonomy

One of the contributions of this project is the concurrent defects taxonomy. The studies referenced in the previous Section have addressed types of defects in concurrent programs, but none to our knowledge have proposed an unified taxonomy considering different sources and experimental research results. Our objective is to consolidate, illustrate, and describe the most common concurrent defects for both shared and distributed memory programs, using a complementary vision. This defect taxonomy can be used as a base for further criteria definition, empirical validations and static/dynamic techniques implementations.

The Concurrent Defects Taxonomy consists of a composition of concurrency types of errors grouped by their common characteristics, based on different studies and scenarios (real applications and testing benchmarks). The proposed taxonomy contemplates five main categories for both shared and distributed concurrent programs: *Data race*, *Deadlock*, *Mismatches*, *Resource*, and *Others*. Table 3 presents the Defects Taxonomy for shared and message-passing concurrent programs.

In order to arrange the compilation of concurrency errors we based on the resulting papers of our systematic mapping review ([DIAZ; SOUZA, 2017](#)), collecting studies from

different approaches and contributions (coverage-based methods, testing frameworks, test models, empirical evaluations, test data generation, and debugging techniques).

We analyze and distinguish the errors described in each study, how they relate to each other and what kind of behavior they cause in the execution and outcomes of concurrent programs (according to the results discussed in the studies). Therefore, we gathered each error scenario and classified them in the categories we defined, avoiding redundancy in each group. We follow a complementary approach, taking the common defects in the existing taxonomies (as an intersection), and including their own differentiated types of defects (unification). In this manner we do not overlap defects among categories. There are not plentiful of studies and specifications regarding taxonomy of errors for message passing programs, as there are for shared memory.

For shared memory there are several studies that classify defects based on non-atomicity of operations and lock acquisition (BRADBURY, 2007; Farchi; Nir; Ur, 2003; CHOI; LEWIS, 2000; SAHOO; RAY, 2016). For distributed memory, (DESOUZA *et al.*, 2005) considers deadlock errors related to synchronization among processes. (LU *et al.*, 2008) and (KRAWCZYK; WISZNIEWSKI; MORK,) present classifications applicable for both memory paradigms. As observed, it is commonly seen (PEDERSEN, 2006; ARORA; BHATIA; SINGH, 2016; DESOUZA *et al.*, 2005) that most defects in concurrent programs relate to deadlock situations and data races issues. *Deadlock* category collects errors caused by synchronization events related to locking management in shared memory programs, and to non-determinism in distributed memory programs. The principal consequence of the manifestation of these types of defects impede the finalization of the program execution.

Data race defects are related to the data integrity involved in synchronization events. We include atomicity violations in this category, as data accessed by threads is affected (LOURENÇO *et al.*, 1997; BRADBURY, 2007). Data race conditions in message-passing programs are treated in (DESOUZA *et al.*, 2005), introducing the concept of interface and inter-process data race regarding the content of sent/received messages and the application buffer. (KRAWCZYK; WISZNIEWSKI; MORK,) introduces the concept of observability error, related to non-determinism in concurrent programs.

The *Mismatches* category is related to matching sender and receiver messages in communication events for distributed memory programs (DESOUZA *et al.*, 2005), which for multi-threaded programs is implicit. It also considers message-passing functions arguments inconsistencies. These types of defects might cause discrepancies in the interchanged data between processes. We identified the Mismatches classification based on the origin of the issues before the communication process happens, rather than focusing on the result of the execution.

Resources category involves the allocation of resources (memory, address space, thread creation, derived data types) and posterior deallocation (DESOUZA *et al.*, 2005), causing unexpected behavior in the program.

Table 3 – Concurrent Defects Taxonomy for shared and distributed memory programs

Category	Shared memory		Distributed memory
Data race	Atomicity violations Two-state access Data-race interference Order violation Accidental race		Interface data race Inter-process data race Send buffer integrity (writing before completion) Observability
Deadlock	Code protection	Lock-related (wrong, non existent, only one, re-obtain, multiple) Double-checked locking	Time-dependent deadlock Standard deadlock Collective communication order call Infinite loop
	Thread blocking	Orphan thread Starvation (Livelock) Block critical section Sleep pattern	
	Interleavings	Notify/Notify all - related (missing) Dead/Blocked thread Signal-related (missing, non existent)	
Mismatches	N/A		Call type Argument mismatch Message length specification Send-Receive mismatch Data type mismatch Data exchange
Resource	Resource exhaustion		Allocation of resources Data/Variable initialization Deallocation Overlapped receive buffers
Others	Incorrect count initialization No variable initialization Unnecessary interrupts		Data decomposition Functional decomposition API use Protocol issue

There are different groupings for concurrency defects, some of them base on design patterns, developer assumptions and whether it is related to synchronization or communication among processes or threads. We established the category *Others* to classify such errors and also defects related to management of the programming language itself or algorithmic logic.

For the validation of the proposed taxonomy we contemplate the performed evaluations in the studies presented in Section 5.2, besides the previous works developed by our research group (Section 3.5) that use some of those taxonomies. We also consider the experimental study of the criteria proposed in this work (Chapter 8).

(BRADBURY, 2007) employs the deduced error patterns to generate concurrency mutation operators for mutation testing empirical assessment. Real and large applications are analyzed in (LU *et al.*, 2008) for error patterns, validating the characterized defects in known concurrent programs. An experimental evaluation using subjects to perform tests in a testing tool is described in (CHOI; LEWIS, 2000), resulting in the classification of common mistakes that insert errors. For the proposition of debugging techniques, (DESOUZA *et al.*, 2005) analyzes the three categories.

Additional validations based on concurrent defects have been treated in empirical studies, relying on common defined errors (BRITO, 2011; MELO *et al.*, 2017). Our work bases on the proposed Concurrent Defects Taxonomy to complete the analysis for testing criteria definition.

A broader evaluation contemplating defects from diverse categories and for both memory paradigm programs is an extensive subsequent work.

5.4 Final considerations

This chapter presented the Defects taxonomy for concurrent programs, describing the methodology to select and collect previous research concerned in concurrency errors. As nowadays are more studies focusing on improving the quality and reliability of concurrent applications through tools, methods, models and frameworks. Therefore, it is important to clearly determine what types of defects should we addressed by research solutions, and propose techniques to reveal errors that might cause failures.

STRUCTURAL TESTING CRITERIA FOR CONCURRENT PROGRAMS

6.1 Initial considerations

In this chapter we propose a structural testing criteria for parallel programs, in the presence of communication primitives inside loops. We detail definitions, associations, considerations and criteria, involving all the elements necessary for a testing model. This is the principal contribution of this project, where our objective is to provide structural testing criteria to guide the selection of test cases and reveal concurrency errors.

Section 6.1 introduces the objective, concerns and approaches to define criteria for the establishment of the test model, Section 6.2 presents the concepts regarding control, data and communication flows of a concurrent program. Finally, Section 6.3 presents the proposed criteria.

6.2 Introduction

A test model establishes the basis for the generation of required elements and selection of test cases, by settling a set of rules known as criteria and using information extracted from the source code of the program.

In order to propose a new set of structural testing criteria, we need to define a test model that introduces the new concepts and a method to retrieve information from data and communication flows of a program. The objective of our structural testing criteria is to consider the presence of loops in concurrent programs, and communication and/or synchronization defects that might be found when analyzing the execution of loops.

We delineate the novel test model considering message-passing concurrent programs, with point-to-point communication patterns and both blocking and non-blocking primitives.

We also consider *for*, *while*, *do ... while* loops for C language in this project. Our test model represents an extension of a previous test model (SOUZA *et al.*, 2008) and an extension of such model (SOUZA; SOUZA; ZALUSKA, 2014).

We focus on data, control and communication elements inside loops in concurrent programs. The nature of these iterative structures in concurrent programs brings challenges related to the definition of different paths, concurrent events and data flow associations. Thus, it is necessary to consider those aspects in the test model before defining the criteria. In the following sections we raise the concerns and discuss solutions related to such aspects.

The types of defects that our proposed criteria approaches are related to distributed memory programs issues (Section 5.3) such as synchronization, communication, resources and mismatch defects. Specifically, observability defects, interface race condition, send/receive mismatch, data type mismatch, overlapped receive buffers, data allocation and data inconsistency. By addressing the types of defects we are able to propose criteria with a focused purpose.

6.2.1 Number of loop iterations

Before proposing criteria, we must define how to select the number of iterations of loops to generate the required elements and identify the associations of events. Some studies address this issue by presenting methods to determine the number of repetitions of a loop statically. However, it has not been completely resolved, as the execution of a loop and its iterations can be only known dynamically. Additionally, since our focus is on structural testing (a white-box technique), we should specify the number of iterations prior to the selection of test cases in order to generate the required elements, rather than following a post-mortem approach (analyzing the loop execution after the program execution) (RAMAMOORTHY; HO; CHEN, 1976; ZANJANI; FULADI; AGHABABA, 2011).

According to (HOWDEN, 1975), there are two main approaches when testing loops: *boundary test*, which implies entering the loop but not iterating it, and *interior test*, indicating the execution of at least one iteration. (HOWDEN, 1975) proposes an approach for choosing loop paths by grouping them into predefined classes to generate test data. Classes can change for different programs, and complex programs might produce large sets of classes challenging the capabilities of the methodology. However, in concurrent programs, the execution of more than one iteration of a loop comprises the execution of different paths and new associations among processes, so it is important to analyze the data, communication and control flows in further iterations.

An approach to select feasible paths in loops is presented in (ZANJANI; FULADI; AGHABABA, 2011), defining rules for effective loop control and specific characteristics of the CFG that the paths must accomplish before using the proposed method. Authors present an algorithm to attempt determining the number of loop iterations, in order to select paths by using

symbolic execution and other methods (such as manually selecting paths and executing only one loop iteration), however, for dependent loops it could generate infeasible paths. (ZANJANI; FULADI; AGHABABA, 2011) also defines two types of loops: *free loops*, in which the number of iterations depends on input data and is unknown, and *dependant loops*, where iterations can be defined in conformity with the executed code.

On the other hand, regarding interleavings among threads, (CHEN *et al.*, 2009) demonstrates that two iterations are sufficient to reveal interleaving defects, when each path executes the same series of interaction events. The demonstration intends to show that the paths in each iteration have equivalent serial traces.

There is a method presented in (RAMAMOORTHY; HO; CHEN, 1976) that selects a constant to define the number of executions of a loop relying on the tester's decision, however, it can also be defined by a test generation data system. It is mainly based on the execution of loops whose number of iterations depends on different inputs.

We employed the approach proposed in (RAMAMOORTHY; HO; CHEN, 1976) to select the number k of iterations, where $k \geq 2$. Our election bases on our previous analysis of the concurrency defects that this test model intends to reveal. Our preliminary empirical studies on several parallel benchmarks with non-deterministic concurrency defects (see Section 8) showed that a greater number of iterations does not contribute to reveal significant errors. However, contributes to increase the number of infeasible iterations and impacting testing activity cost.

The condition of two iterations minimum ($k \geq 2$) has been applied in other contexts, for example those related to pairwise test generation¹. Pairwise testing or coupling effect, considers the execution of relatively simple tests to discover the most of software faults. The hypothesis behind pairwise testing claims that a fault might be revealed with a subset of the program's input domain, concerning to the subject of combinatorial data generation to obtain discrete test data sets. Therefore, we establish a correlation with the combinations of loop iterations in the generation of required elements, where more than two loop executions will contribute to reveal an irrelevant amount of defects.

A discussion on the values of the k constant and its implications when implementing the criteria to generate test cases is presented in Section 8, taking into account our results with the empirical evaluation of criteria.

6.2.2 Nested loops

When there are nested loops, the complexity of the criteria could increase exponentially, as every loop should be executed k times, making such inner loops to iterate for $k^{\text{number of loops}}$ repetitions. In order to reduce a great quantity of occurrences due to nested loops, and considering that the approach of our test model is the revelation of defects associated to parallel programs, the

¹ <https://www.conformiq.com/2012/01/understanding-pairwise-test-generation/>

generation of required elements contemplates loops that have communication or synchronization primitives, or variables in a loop that are directly related to the interaction among processes.

Our approach avoids the analysis of exclusively intra-process associations that might increase the complexity in the implementation of the criteria, but do not contribute to the identification of concurrent-related defects. For example, an association of a definition-c-use.

Additionally, our testing criteria focus mainly on communicational uses inside loops, extending the previous criteria already proposed (SOUZA; SOUZA; ZALUSKA, 2014). For instance, when there is a send primitive inside a loop of a process p_a , and its matching receive in process p_b is not inside a loop, the association event will not be considered in our test model, and this would not generate any required elements. In such case, the association is already contemplated in previously proposed criteria.

6.3 Test model concepts

In our test model a parallel program is represented as $P_{parallel} = \{p^0, p^1, \dots, p^{q-1}\}$ with q processes, where each process p has a Control Flow Graph CFG^p with a series of nodes and edges. Also, a $PCFG$ is formed by all CFG^p of the q processes. We use previous concepts already defined in Section 3.3 from Previous work (Chapter 3).

We define the set L^p that contains all nodes inside loops in a process p . The set $N_{sync-loop}$ represents communication events (nodes with send or receive primitives) that are inside loops of a $PCFG$, and $N_{sync-loop} \subseteq N_{sync}$. A node belonging to a process p in our test model is described as n^p .

The set E_{s-loop} denotes the inter-process edges that are inside a loop, and $E_{s-loop} \subseteq E_s$. Inter-process-loop edges can be represented by a tuple (n^{p_a}, n^{p_b}) where $n^{p_a}, n^{p_b} \in N_{sync-loop}$.

The definition of a variable x is given when a value is assigned to it, and this test model also considers definitions in receive primitives such that $x \in def(n^p) \wedge n^p \in N_{sync-loop}$, as they set the value of a variable that has been received in a message.

Consider a scenario where a variable definition $x \in def(n^p)$ is outside a loop ($n^p \notin L^p$), and an s-use of x is inside the loop: $s-use(x) \in E_{s-loop}$. Existing criteria already aims to cover such associations, and basing on their perspective, it would not be meaningful to generate an association where a second iteration of the s-use is considered as a different event, as there is no re-definition of x and there is no new information involved in the communication process.

Accordingly, this test model requires to identify the variables that are modified inside loops and are related to communication primitives, in order to establish the relationship among the elements of interest in such repetition structures. For example, we do not consider a c-use of a variable defined in a receive primitive in a node n^p such that $n^p \notin N_{sync-loop}$, in other words, the receive definition is outside the loop.

Our test model proposes an extended concept for associations in order to support the novel criteria. Consider communications between two processes p_a and p_b . Nodes $n^p, m^{p_a}, w^{p_b} \in N_{sync-loop}$, an edge $(m^{p_a}, w^{p_b}) \in E_{s-loop}$, and x is a variable such that $x \in def(n^p)$ and the definition could be in a receive primitive where $n^p \in N_{sync-loop}$. Nodes v^{p_b}, z^{p_b} represent a c-use in v^{p_b} or the nodes of a p-use edge (v^{p_b}, z^{p_b}) in the receiving process p_b , such that $v^{p_b}, z^{p_b} \in L^{p_b}$. The extended definition of the loop-associations are:

- **s-use loop association:** describes the association between a variable definition $x \in def(n^{p_a}) \wedge n^{p_a} \in L^{p_a}$ and its communicational use inside a loop $(m^{p_a}, w^{p_b}) \in E_{s-loop}$, with a definition-clear path between n^{p_a} and m^{p_a} . The s-use-loop association is represented by the triple $(n^{p_a}, (m^{p_a}, w^{p_b}), x)$.
- **s-c-use loop association:** is given by the association between a variable definition $x^{p_a} \in def(n^{p_a}) \wedge n^{p_a} \in L^{p_a}$ and its communicational use inside a loop $(m^{p_a}, w^{p_b}) \in E_{s-loop}$, with a definition-clear path between n^{p_a} and m^{p_a} . Followed by a computational use from the variable definition in a receive primitive $x^{p_b} \in def(w^{p_b}) \wedge w^{p_b} \in N_{sync-loop}$ to c-use node $v^{p_b} \in L^{p_b}$, with a definition-clear path from w^{p_b} to v^{p_b} .

The s-c-use association is represented by a tuple $(n^{p_a}, (m^{p_a}, w^{p_b}), v^{p_b}, x^{p_a}, x^{p_b})$, where there is an s-use-loop association $(n^{p_a}, (m^{p_a}, w^{p_b}), x^{p_a})$ and a c-use association $(w^{p_b}, v^{p_b}, x^{p_b})$.

- **s-p-use loop association:** is given by the association between a variable definition $x^{p_a} \in def(n^{p_a}) \wedge n^{p_a} \in L^{p_a}$ and its communicational use inside a loop $(m^{p_a}, w^{p_b}) \in E_{s-loop}$, with a definition-clear path between n^{p_a} and m^{p_a} . Followed by a predicate use from the variable definition in a receive primitive $x^{p_b} \in def(w^{p_b}) \wedge w^{p_b} \in N_{sync-loop}$ to p-use edge (v^{p_b}, z^{p_b}) where $v^{p_b}, z^{p_b} \in L^{p_b}$, with a definition-clear path from w^{p_b} to v^{p_b} .

The s-p-use-loop association is represented by a tuple $(n^{p_a}, (m^{p_a}, w^{p_b}), (v^{p_b}, z^{p_b}), x^{p_a}, x^{p_b})$, where there is an s-use-loop association $(n^{p_a}, (m^{p_a}, w^{p_b}), x^{p_a})$ and a p-use association $(w^{p_b}, (v^{p_b}, z^{p_b}), x^{p_b})$.

There are different scenarios to contemplate when defining testing criteria, for communication, data and control flows. For example, a variable might be defined outside or inside of the loop, or it could be defined at the end of an iteration through different paths. However, our criteria only considers the execution of *send/receive* functions and definitions and uses of variables that are inside loops, for all of the processes involved in the communication scheme. This is because the existing criteria already constitutes required elements that cover paths from a definition outside a loop and with a use inside a loop. If an association for this scenario is created, every loop iteration would be considered as a repetition of a required element rather than a new event, since there are no definitions that change the state of the variable and thus no information from code faults could be revealed. Enclosing this situations makes our proposed criteria to work along the existing testing models and allows the testing activity to be effective.

6.4 Structural testing criteria for message passing programs

We differentiate the execution of communication events inside loops for parallel programs in that each iteration corresponds to a different communication occurrence. For this reason, in this test model each required element in the iteration of a loop with communication primitives refers to an *event*.

The amount of executions of a required element depends on whether it is in a nested loop or not, and is given by the depth d of the loop to which the element (node, edge or association) belongs to. Therefore, if we have it_p representing each iteration of the loop in a process p , then $1 \leq it_p \leq k^d$. If we have two processes involved, it_a would represent the iteration in a loop in process p_a and it_b the iteration number in a loop in process p_b .

In such a manner, the required elements in this test model are replicated based on the number of loop iterations k and nested loop depth d , and so forth we represent required elements differently from the nodes, adding the it_p iteration. If we have a node n^p , the related event is described as $n_{it_p}^p$, where it_p is the iteration number of the required element.

We present the structural testing criteria for concurrent programs proposed in this test model, applying the exposed concepts and considerations.

Control and communication flow structural criteria:

- **All events-s-loop:** the test set must cover all nodes n^p with send primitives where $n^p \in N_{sync-loop}$ for each it_p iteration, such that $1 \leq it_p \leq k^d$. Required elements are represented by $n_{it_p}^p$.
- **All events-r-loop:** the test set must cover all nodes with receive primitives where $n^p \in N_{sync-loop}$ for each it_p iteration, such that $1 \leq it_p \leq k^d$. Required elements are represented by $n_{it_p}^p$.
- **All sync-events-loop:** the test set must cover all inter-process edges $(n^{p_a}, m_i^{p_b}) \in E_{s-loop}$ for each it_p iteration, such that $1 \leq it_p \leq k^d$. Required elements are represented by $(n_{it_p}^{p_a}, m_{it_p}^{p_b})$.

Data and message passing flow structural criteria:

- **All s-uses-loop:** the test set must execute paths that cover all s-use-loop associations for each it_a, it_b iterations, such that $1 \leq it_a, it_b \leq k^d$. Required elements are represented by $(n_{it_a}^{p_a}, (m_{it_a}^{p_a}, w_{it_b}^{p_b}), x)$.
- **All defs-recv:** the test set must execute paths that cover all definitions in receive primitives, where $x \in def(n^p)$ and $n^p \in N_{sync-loop}$ for each it_p iteration such that $1 \leq it_p \leq k^d$. Required elements are represented by $n_{it_p}^p$.

- **All s-c-uses-loop:** the test set must execute paths that cover all s-c-use-loop associations for each it_a, it_b iterations, such that $1 \leq it_a, it_b \leq k^d$. Required elements are represented by $(n_{it_a}^{p_a}, (m_{it_a}^{p_a}, w_{it_b}^{p_b}), v_{it_b}^{p_b}, x^{p_a}, x^{p_b})$.
- **All s-p-uses-loop:** the test set must execute paths that cover all s-p-use-loop associations for each it_a, it_b iterations, such that $1 \leq it_a, it_b \leq k^d$. Required elements are represented by $(n_{it_a}^{p_a}, (m_{it_a}^{p_a}, w_{it_b}^{p_b}), (v_{it_b}^{p_b}, z_{it_b}^{p_b}), x^{p_a}, x^{p_b})$.

We address the issue of non-determinism in this test model by considering the different interactions among processes in the loop. When there is a non-deterministic receive inside loops, the destination process might receive in any order from other processes in any iteration. By generating required elements according to our criteria rules, we are considering all possible interleavings in a loop between two processes, forcing the selection of test cases for the execution of distinct synchronization pairs.

Infeasible required elements are also present in this test model, as they are still one of the disadvantages of structural testing and an undecidable problem (RAPPS; WEYUKER, 1982). However, the consideration of all interleavings in a loop also increases the number of infeasible required elements, as they are generated for communications among all processes even if there is not any interaction between two or more processes. Moreover, the amount of infeasible required elements might grow depending on the number of loop iterations and the depth of nested loops. For example, a non-executable communication event in a loop (for example, due to an s-edge that is never exercised) might have k infeasible repetitions, which is not favorable on a perspective of amount of infeasible elements. Nonetheless, those infeasible communication events are not difficult to identify.

6.5 Test model exemplification

In this section we present an application example of our proposed criteria to a C/MPI program for GCD calculation (Algorithms 1 and 2). We use this program as it has the characteristics in its structure and logic to generate required elements for all of the proposed criteria.

Our aim is to illustrate in a practical and simple exercise the elements of our test model, showing the sets, associations and required elements generated by the proposed criteria. Keep in mind that our criteria replicates required elements (or events) for loop iterations, not nodes in the graph.

Consider the PCFG for the CGD program, with the relevant communication edges and definitions. ValiMPI generates the GFC for each functions executed by processes, so we adapted the graphs to:

- i Illustrate the def-use associations that are pertinent for our criteria, in order to avoid complex representations due to the size of the graph.
- ii Focus on the sections of the graph that are of our interest: loops in the graph, nodes with definitions and communicational uses. We disregard the representation of loops with no communicational primitives and nodes that are not pertinent for the generation of sets and required elements.
- iii Present the PCFG in a legible form, representing the elements of our test model.

Figure 6 presents the PCFG for the GCD program. Dashed lines represent a synchronization edge *s-edge*, and the ones that have a communicational use are marked as *s-use*. Dots indicate that there are other nodes and edges in the graph but are not relevant for this exemplification. Note that we do not include the non-executable *s-edges* in the graph.

Figure 6 – Parallel Control Flow Graph for GCD program

The definition sets involved in our test model are described in Table 4. The $def(26^0) = values$ is defined in a receive primitive.

Table 4 – Definition sets for GCD program

$def(11^1) = values$	$def(11^2) = values$	$def(26^0) = values$
$def(5^1) = values$	$def(5^2) = values$	

Table 5 shows the elements for sets L^p , N_{sync_loop} and E_{s_loop} defined in the test model.

Table 5 – Test model sets for GCD program

$q = 3$
$P_{parallel} = \{p^0, p^1, p^2\}$
$L^{p^0} = \{11^0, 16^0, 17^0, 20^0, 19^0, 22^0, 23^0, 19^0, 18^0, 15^0, 25^0, 26^0, 28^0, 30^0, 31^0, 32^0, 33^0, 29^0, 27^0\}$
$L^{p^1} = \{3^1, 4^1, 5^1, 7^1, 8^1, 9^1, 11^1, 12^1, 10^1, 14^1, 15^1, 17^1, 18^1, 19^1, 20^1, 16^1, 11^1, 19^1, 6^1\}$
$L^{p^2} = \{3^2, 4^2, 5^2, 7^2, 8^2, 9^2, 11^2, 12^2, 10^2, 14^2, 15^2, 17^2, 18^2, 19^2, 20^2, 16^2, 11^2, 19^2, 6^2\}$
$N_{sync\_loop} = \{19^1, 5^1, 19^2, 5^2, 20^0, 22^0, 26^0\}$
$E_{s\_loop} = \{(20^0, 5^1), (22^0, 5^1), (20^0, 5^2), (22^0, 5^2), (19^1, 26^0), (19^2, 26^0)\}$

Table 6 presents the required elements for our criteria (part of All-s-c-uses-loop and All-s-p-uses-loop, due to their extension). Observe that for All-s-p-uses-loop criterion the generated required elements are non-executable, because it considers associations between processes p^1 and p^2 , which do not communicate to each other. Our criteria generates required elements for the associations found between two processes, regardless of their non-existent interaction during execution. This situation is influenced by the structure and logic of the program.

Recapitulating the observability error example described in Section 1.1 for GCD program, the defect can be revealed by All-s-uses-loop and All-s-c-uses-loop criteria by covering required elements that execute the second iteration of the loop, such as $(11_2^1, (19_2^1, 26_2^0), values_{p^1})$, $(11_2^2, (19_2^2, 26_2^0), values_{p^2})$, and $(11_2^1, (19_2^1, 26_2^0), 30_2^0, values_{p^1}, values_{p^0})$. We analyze the criteria effectiveness further in Chapter 8.

6.6 Final Considerations

In this chapter we presented the proposed testing model for structural testing criteria in the presence of loops, basing on a previous definition of criteria for message-passing programs (SOUZA *et al.*, 2008; SOUZA; SOUZA; ZALUSKA, 2014). We introduced the concept of *event* in the model, for the occurrence of communication and synchronization primitives in loops

Table 6 – Required elements for structural testing loop criteria

Criteria	Required elements
All-events-s-loop	$(20_1^0), (20_2^0), (22_1^0), (22_2^0), (19_1^1), (19_2^1), (19_1^2), (19_2^2)$
All-events-r-loop	$(26_1^0), (26_2^0), (5_1^1), (5_2^1), (5_1^2), (5_2^2)$
All-sync-events-loop	$(20_1^0, 5_1^1), (20_2^0, 5_2^1), (20_1^0, 5_1^2), (20_2^0, 5_2^2), (22_1^0, 5_1^1), (22_2^0, 5_2^1), (22_1^0, 5_1^2), (22_2^0, 5_2^2), (19_1^1, 26_1^0), (19_2^1, 26_2^0), (19_1^1, 5_1^2), (19_2^1, 5_2^2), (19_1^2, 26_1^0), (19_2^2, 26_2^0), (19_1^2, 5_1^1), (19_2^2, 5_2^1)$
All-defs-recv-loop	$(26_1^0, values_{p^0}), (26_2^0, values_{p^0}), (5_1^1, values_{p^1}), (5_2^1, values_{p^1}), (5_1^2, values_{p^2}), (5_2^2, values_{p^2})$
All-s-uses-loop	$(11_1^1, (19_1^1, 26_1^0), values_{p^1}), (11_1^1, (19_1^1, 26_2^0), values_{p^1}), (11_2^1, (19_2^1, 26_1^0), values_{p^1}), (11_2^1, (19_2^1, 26_2^0), values_{p^1}), (11_1^1, (19_1^1, 5_1^2), values_{p^1}), (11_1^1, (19_1^1, 5_2^2), values_{p^1}), (11_2^1, (19_2^1, 5_1^2), values_{p^1}), (11_2^1, (19_2^1, 5_2^2), values_{p^1}), (11_1^2, (19_1^2, 26_1^0), values_{p^2}), (11_2^2, (19_2^2, 26_1^0), values_{p^2}), (11_1^2, (19_1^2, 26_2^0), values_{p^2}), (11_2^2, (19_2^2, 26_2^0), values_{p^2}), (11_1^2, (19_1^2, 5_1^1), values_{p^2}), (11_2^2, (19_2^2, 5_1^1), values_{p^2}), (11_1^2, (19_1^2, 5_2^1), values_{p^2}), (11_2^2, (19_2^2, 5_2^1), values_{p^2}), (11_2^2, (19_2^2, 5_2^2), values_{p^2})$
All-s-c-uses-loop	$(11_1^1, (19_1^1, 26_1^0), 30_1^0, values_{p^1}, values_{p^0}), (11_1^1, (19_1^1, 26_2^0), 30_2^0, values_{p^1}, values_{p^0}), (11_2^1, (19_2^1, 26_1^0), 30_1^0, values_{p^1}, values_{p^0}), (11_2^1, (19_2^1, 26_2^0), 30_2^0, values_{p^1}, values_{p^0}), (11_1^1, (19_1^1, 26_1^0), 32_1^0, values_{p^1}, values_{p^0}), (11_1^1, (19_1^1, 26_2^0), 32_2^0, values_{p^1}, values_{p^0}), (11_2^1, (19_2^1, 26_1^0), 32_1^0, values_{p^1}, values_{p^0}), (11_2^1, (19_2^1, 26_2^0), 32_2^0, values_{p^1}, values_{p^0}), (11_1^1, (19_1^1, 5_2^2), 7_1^2, values_{p^1}, values_{p^2}), (11_1^1, (19_1^1, 5_2^2), 7_2^2, values_{p^1}, values_{p^2}), (11_2^1, (19_2^1, 5_2^2), 7_1^2, values_{p^1}, values_{p^2}), (11_2^1, (19_2^1, 5_2^2), 7_2^2, values_{p^1}, values_{p^2}) \dots$
All-s-p-uses-loop	$(11_1^1, (19_1^1, 5_2^2), (7_1^2, 9_1^2), values_{p^1}, values_{p^2}), (11_1^1, (19_1^1, 5_2^2), (7_2^2, 9_2^2), values_{p^1}, values_{p^2}), (11_2^1, (19_2^1, 5_2^2), (7_1^2, 9_1^2), values_{p^1}, values_{p^2}), (11_2^1, (19_2^1, 5_2^2), (7_2^2, 9_2^2), values_{p^1}, values_{p^2}), (11_1^1, (19_1^1, 5_2^2), (7_1^2, 8_1^2), values_{p^1}, values_{p^2}), (11_1^1, (19_1^1, 5_2^2), (7_2^2, 8_2^2), values_{p^1}, values_{p^2}), (11_2^1, (19_2^1, 5_2^2), (7_1^2, 8_1^2), values_{p^1}, values_{p^2}), (11_2^1, (19_2^1, 5_2^2), (7_2^2, 8_2^2), values_{p^1}, values_{p^2}), (11_1^1, (19_1^1, 5_2^2), (12_1^2, 11_1^2), values_{p^1}, values_{p^2}), (11_1^1, (19_1^1, 5_2^2), (12_2^2, 11_2^2), values_{p^1}, values_{p^2}), (11_2^1, (19_2^1, 5_2^2), (12_1^2, 11_1^2), values_{p^1}, values_{p^2}), (11_2^1, (19_2^1, 5_2^2), (12_2^2, 11_2^2), values_{p^1}, values_{p^2}), (11_1^1, (19_1^1, 5_2^2), (12_1^2, 13_1^2), values_{p^1}, values_{p^2}), (11_1^1, (19_1^1, 5_2^2), (12_2^2, 13_2^2), values_{p^1}, values_{p^2}), (11_2^1, (19_2^1, 5_2^2), (12_1^2, 13_1^2), values_{p^1}, values_{p^2}), (11_2^1, (19_2^1, 5_2^2), (12_2^2, 13_2^2), values_{p^1}, values_{p^2}) \dots$

between processes. Thus, it was required to re-define nodes and edges sets, as well as to extend the concepts for variables definition-communicational-use associations. To define the structural testing criteria considering loop execution, we resolved issues regarding the number of loop iterations and the possibility of nested loops.

We achieved to propose structural testing criteria that allows the generation of required elements that have not been considered in previous test models. It presumes the finding of concurrency defects that might be present in communication events that occur in loop iterations.

TESTING CRITERIA IMPLEMENTATION

7.1 Initial considerations

In this chapter, we present the modifications performed on ValiMPI to implement our proposed criteria, guided by the test model and development standards of the tool. The following sections will describe the considerations and implications in the adaptation of the modules of ValiMPI, as well as a detailed revision of the development for this project.

We recommend a previous reading of Chapter 3, Section 3.4.2 for basic notions on the functions of ValiMPI, as we will approach some technical subjects.

7.2 Modifications in ValiMPI modules

The existing structural testing criteria (SOUZA *et al.*, 2008; SOUZA; SOUZA; ZALUSKA, 2014; SOUZA *et al.*, 2013) is implemented in ValiMPI, however, it does not contemplate the execution of communication primitives in loops. Thus, we perform specific enhancements in ValiMPI to provide the required support for the extended criteria proposed in this research work.

We develop improvements in ValiMPI in a modular form to avoid major adjustments in the original code. Our objective with this approach is minimizing (1) the refactoring efforts, (2) the insertion of defects in a functional software prototype testing tool and (3) the regression test cost. Based on a software development analysis, we made modifications in three modules of ValiMPI, taking into account the limitations of the tool.

7.2.1 ValiInst

The results generated by ValiInst module (Instrumented source code, data flow information and CFGs) are essential for the correct operation of ValiMPI, as described in Section 3.4.2.

Table 7 – L.<function> file structure

Loop identifier	Nodes/Nested loops identifier
l_1	2 3 4 10 l_2 l_3 14 17
l_2	5 6 7 11 15
l_3	21 23 22 24

To provide an adequate support for our extended model proposal, ValiInst should be adapted to retrieve information from the instrumented file and GFCs to identify the nodes that are inside loops and those in nested loops.

Due to timeline limitations, such modifications were not performed in ValiInst and to this extent, the tester must recover manually the loop information. Nonetheless, ValiMPI allows the implementation of the proposed criteria, through the generation of required elements offering support to the execution of the parallel instrumented programs and their coverage analysis. Considering that the main goal of using ValiMPI in this project is to validate our structural testing criteria through an experimental study, we are accomplishing our objective. Such alterations will be contemplated for a future work, potentially related to an undergraduate research (Scientific Initiation) project.

In consideration of the existent code of ValiInst, future alterations should generate a file for each process, listing the nodes of each loop as $l_i = \{n_1, n_2, \dots, n_j\}$, where i is a sequential identifier for the loops and j the total number of loops in p . If a loop has nested loops, they must be specified by their identifiers. See Table 7 for a practical case. The aforementioned loop information files are manually created, and are used by ValiElem for the automated generation of required elements.

7.2.2 ValiElem

The generation of required elements by ValiElem takes into consideration the definitions and uses (communicational, computational and predicate) of the variables inside loops. Additionally, we acknowledge the constant k informed by ValiMPI user when the module is executed, which establishes the number of required iterations of each required element. The depth of nested loops depends on the value of k and inner loops findings in loop file information.

For the execution of ValiElem the k value has been added and is given by:

vali_elem $\langle NP \rangle$ “ $function_i(p_a, p_b, \dots)$ ” “ $function_j(p_c, p_d, \dots)$ ” $-k$ $\langle k \text{ value} \rangle$

Where, besides the k parameter, NP is the number of processes and “ $function_i(p_a, p_b, \dots)$ ” is the function executed by processes p_a, p_b , and “ $function_j(p_c, p_d, \dots)$ ” is the function executed by processes p_c, p_d .

Capturing information from loops

In addition to the graphs-i method described in Section 3.4.2, we investigate and analyze other three loop path selection methods with different perspectives, in order to identify their main features. Our aim with this brief study is to corroborate if there is another efficient mechanism to address the generation of required elements, different from the graph-i which is already implemented in ValiMPI.

We consider the following scenarios in the communication between two processes p_a and p_b : definition-use pair paths in process p_a and paths in p_b , paths inside loops of each process and the iterations of each loop. In such a way, we must define a mechanism to identify those paths so that the tool is able to recognize the associations required by new criteria, considering that those associations are inside repetition structures that might have different executable paths in each iteration.

The approach of Zanjani et al. (ZANJANI; FULADI; AGHABABA, 2011) selects paths from loops focusing on the feasibility of selected paths, using dynamic analysis to traverse the CFG and symbolic execution to determine the type of the loop and thus obtain the number of iterations. Zanjani et al. defines rules that loops must accomplish to implement the method, and presents characteristics of the CFG representation. An algorithm loads a data structure with information to evaluate a loop and establishes its execution based on path feasibility.

The proposal of Breveglieri et al. (BREVEGLIERI; REGHIZZI; GOLDWURM, 2010) represents Repeat-until loops (*do...while*) using a formal language, where integer compositions (a mathematical structure) represent loop iterations. A syntax tree is created to generate words that represent input strings, providing a solution to schedule checking problems (sequence of operations in a program).

Another investigation on capturing information from loops analyzes the paths inside a loop by "unwrapping" such paths (WANG; HUANG; GONG, 2013). It allows to find possible paths to test, establishing the execution order of loop paths. This study proposes a differentiation of loops and sub-loops and their correspondent sequence of nodes, through a loop processing model structure for path selection. By unfolding paths in a loops Wang et al. (WANG; HUANG; GONG, 2013) defines loop sub-paths that are selected using an algorithm, marking the order in which they will execute. However, the study does not consider the execution of the same path in the loop nor criteria to test programs.

Table 8 shows a comparison among the different methods investigated for path selection in loops, by evaluating the advantages, disadvantages, complexity and implementation requirements, considering the functioning of ValiMPI.

We analyzed the algorithm complexity for each method, the disadvantages regarding our requirements (some methods do not contemplate nested loops or combined constraints), the lack of accessibility to important information to implement some algorithms (BREVEG-

LIERI; REGHIZZI; GOLDWURM, 2010), and the requirements to develop more complex implementations that diverged from ValiMPI's paradigm.

Table 8 – Loop path selection methods

Subject of evaluation	Graphs-i (HAUSEN, 2005)	Improving path selection by handling loops in automatic test data generation (ZANJANI; FULADI; AGHABABA, 2011)	Efficient recognition of trace languages defined by repeat-until loops (BREVEGLIERI; REGHIZZI; GOLDWURM, 2010)	Research on Loop Path Selection in Coverage Testing (WANG; HUANG; GONG, 2013)
Algorithm	Generates a graph for each node with a variable definition, with free definition paths and without cycles.	Algorithm that selects executable paths of a program containing loops, establishing whether the loop is executed again in the next iteration.	Generates a syntax tree of a word from a language defined by a repeat-until nested loop, representing an input string. Computes the nodes of the syntax tree to execute the initial loop, and then the iterations of the nested loop.	Selects sub-paths of the loop and establishes the order of execution of those sub-paths. It traverses the program searching for all the loops and their initial nodes.
Advantages	Considers nested loops and loop constraints.	Looks for diminishing the testing cost in terms of time, avoiding non-executable paths that are related to an unappropriated selection of loop iterations.	Considers nested loops.	Looks for finding executable paths of the program that traverse loops.
Disadvantages	Verification of nodes with definitions previous to path selection, as a consequence it duplicates paths when finding node with definition.	It does not allow loops with combined constraints. Verifies which node can be executed through an algorithm (which was not referenced in the article). There are unresolved gaps in the theoretical basis of the proposed algorithm.	There is a wide theoretical basis that has to be introduced into the ValiMPI logic.	It does not consider nested loops.
Complexity	$O(nr)$ complexity, being n the number of nodes of the original graph and r the output degree of each node.	$O(n^2)$ complexity, being n the number of nodes.	$O(nm^3 \log m)$, being m the cardinality of the alphabet set, and n the word length (input strings).	Polynomial complexity. The execution depends on the recursive calls, the number of loops and number of nodes in the loops.
Implementation	Validation of the paths in the graph-i for the nodes in loops, for the identification of paths for the associations depending on the k factor.	Implementation of both mentioned algorithms. Requires symbolic execution to define the number of iterations of dependent loops. It is necessary to detect the type of loop.	Requires the implementation of algorithms for: (1) syntax tree generation, (2) compose the representation of loop execution and (3) construction of composed finite representation (labeled compositions).	Implementation of algorithms to recognize the loops, generate paths inside the loops and tagging them, and identification of the paths before and after the loops. It also presents an algorithm to select executable paths, in order to avoid selection of program paths that are infeasible.

Evaluating the studies' main features (Table 8), we concluded that the most efficient and consistent method with ValiMPI is the graph-i, already implemented in ValiElem module. The paths in loops are currently unfolded when generating each sub-graph, however, the paths of the different iterations are not contemplated. The replication of paths could be a possible solution

for loop iterations depending on the k constant, but in order to avoid a considerable growth in the size of the graphs- i , we opted for not duplicating the paths in the loop, which would generate more sub-graphs as well.

This unfolding-paths method is also applied for control flow criteria, except it disregards the conditions to validate nodes with variable definitions, as graphs- i are generated for data flow criteria in ValiMPI. In such manner, for control flow required elements, we validate the nodes and edges of each processes' CFG.

To retrieve data from the loop information file (See Table 7), a new function reads such file and saves the list of nodes in the loop into a structure. This structure is complemented through a processing that establish the following information: number of nodes, process, number of parent loops (represents the depth, if it has any nested loops), and number of sub-loops (loop identifiers, if it has any). Each loop has a unique identifier in a single process.

The number of parent loops (nested loop depth d) determines the number of times that a communication event must be executed, which allows us to control the execution of nested loops. Therefore, the number of iterations of a determined required element is given by k^d .

The loop information retrieval is performed when ValiElem creates the graph structure from the DOT file (graph description file) for each process. In this way, the performance of the module was not highly compromised by reading the graph file twice (for previous and new criteria). Thus, we took the graph generation process and improved it to read paths from loops, which have been identified in a different file.

After the loop information has been taken, the required elements are generated for all of the loop structural testing criteria.

Generation of required elements.

The novel required elements need to differentiate all events occurring during the k iterations of the loops. Graph- i constructs paths that start from a definition of a variable ensuring that there is no redefinition of such variable. The graph- i contains all paths to the computational, predicate and communicational uses of the defined variable.

The loop information file is processed obtaining the depth of nested loops, in order to determine the number of required executions for each required element. The iteration numbers of each event component depends on the depth d of the loop that the analyzed node is. For example, when looking for s-use-loop associations we first find the variable definition and then the s-use, where the send node (in sender process) can be in a single loop and the receive node (in receiving process) in a nested loop. Therefore, the depth d is different for the receiving node component in the s-use association.

We validate whether the first node of each graph- i (which corresponds to the variable definition) belongs or not to a loop, avoiding the analysis of the entire graph. After the is-in-loop

verification, we start traversing the rest of the nodes of the graph-*i*, looking for s-uses belonging to the set of nodes in the loop. We first look for communicational uses of the defined variable, to establish All-s-uses-loop associations, to soon after examine the c-uses or p-uses in the graph of the receiving process (for All-s-c-use-loop and All-s-p-use-loop criteria associations).

Since our required elements must contain the iteration number, it was necessary to modify the previous pattern for the required elements from *node-process* to *node-process-iteration*, to allow the verification of the loop repetitions that were executed and determine covered required elements.

Our solution allows the generation of the file of required elements considering the new pattern, showing each event and how many iterations each event will have. Table 9 exemplifies the content of such file where three nodes belonging to loops must be executed twice (observe they are duplicated). In this example we have $k = 2$, nodes 12 and 14 in process 0, and node 5 belongs to process 1. Figure 7 presents an example of the generated file of required elements for All-sync-events-loop criterion.

The file of required elements contains the associations and the iterations that must be covered by the test cases. Differently, the automaton file contains the descriptor of the automaton for each required element without considering its replications. That is, the automaton is able to read the k iterations of a required element and recognize them.

Table 9 – Base pattern of required elements for loop criteria

<i>node-process-iteration</i>
12 - 0 - 1
12 - 0 - 2
14 - 0 - 1
14 - 0 - 2
5 - 1 - 1
5 - 1 - 2

ValiMPI implements the automata through the descriptor files, having that for previous criteria, one descriptor is generated for each required element (refer to Section 3.4.2). For the new criteria, one descriptor is used for all of the required iterations of a required element. This approach optimizes the process of reading the automata descriptors, avoiding the re-interpretation of each event that must be covered during the test activity. Figure 8 presents an example of the descriptor file for the automata for All-sync-events-loop criterion.

The files containing the descriptors maintain the previous pattern for the early criteria. The new criteria use automata that recognize loop iterations and accept new regular expressions. Therefore, we include the number of required iterations for each new descriptor, which will be used to evaluate the coverage by ValiEval.

Figure 7 – Required elements file example for All-sync-events-loop

```

REQUIRED ELEMENTS FOR ALL-SYNC-EVENTS-LOOP CRITERIA
<node-source_process-iteration node-destination_process-iteration>

1) 20-0-1 5-1-1
2) 20-0-2 5-1-2
3) 20-0-1 5-2-1
4) 20-0-2 5-2-2
5) 22-0-1 5-1-1
6) 22-0-2 5-1-2
7) 22-0-1 5-2-1
8) 22-0-2 5-2-2
9) 19-1-1 26-0-1
10) 19-1-2 26-0-2
11) 19-1-1 5-2-1
12) 19-1-2 5-2-2
13) 19-2-1 26-0-1
14) 19-2-2 26-0-2
15) 19-2-1 5-1-1
16) 19-2-2 5-1-2

```

Figure 8 – Automata descriptor file example for All-sync-events-loop

```

1) 1 4 4 1 1:Np1/5-1 2:5-1 0 2 3:20-0 0 3 4:5-1 0 4 2:5-1 4:N 0 2 1
2) 1 4 4 1 1:Np2/5-2 2:5-2 0 2 3:20-0 0 3 4:5-2 0 4 2:5-2 4:N 0 2 1
3) 1 4 4 1 1:Np1/5-1 2:5-1 0 2 3:22-0 0 3 4:5-1 0 4 2:5-1 4:N 0 2 1
4) 1 4 4 1 1:Np2/5-2 2:5-2 0 2 3:22-0 0 3 4:5-2 0 4 2:5-2 4:N 0 2 1
5) 1 4 4 1 1:Np0/26-0 2:26-0 0 2 3:19-1 0 3 4:26-0 0 4 2:26-0 4:N 0 2 1
6) 1 4 4 1 1:Np2/5-2 2:5-2 0 2 3:19-1 0 3 4:5-2 0 4 2:5-2 4:N 0 2 1
7) 1 4 4 1 1:Np0/26-0 2:26-0 0 2 3:19-2 0 3 4:26-0 0 4 2:26-0 4:N 0 2 1
8) 1 4 4 1 1:Np1/5-1 2:5-1 0 2 3:19-2 0 3 4:5-1 0 4 2:5-1 4:N 0 2 1

```

Figures 9, 10, 11, 12, 13, 14, 15 illustrate the Deterministic Finite Automaton (DFA) for all novel criteria.

Figure 9 – DFA for All-events-s-loop criterion

As described in Section 3.4.2, criteria involving the communication among two processes consider two automata, one for each process' trace. An exception in this approach is the All-sync-events-loop (ASEL) criterion, as we are analyzing an s-edge from a send node in a process p_a to a receive node in a process p_b . In such situation, it is possible to determine in the trace of p_b (the receiving process in a communication event) the sender node and process, and in this manner we can establish the s-edge that should be covered. Therefore, the automaton for the ASEL criterion considers the trace file of the receiving process p_b , where the nodes with send and receive are specified.

Figure 10 – DFA for All-events-r-loop criterion

Figure 11 – DFA for All-sync-events-loop criterion

Figure 12 – DFA for All-defs-recv-loop criterion

The creation of the automata for our criteria is influenced by the patterns generated in the trace files, where nodes with send and receive primitives are represented as follows. When there is a send in a node n^{p_a} , the trace of the process p_a has the string: $n^{p_a} 0^{p_b} n^{p_a}$, where 0^{p_b} indicates that the receiver process is p_b . The 0^{p_b} is placed instead of the number of the receiving node because it can be a non-deterministic receive and therefore it is not known in which exact node of process p_b it received the MPI message. When a receive node is executed, the trace file has the scheme $m^{p_b} n^{p_a} m_i^{p_b}$, where n^{p_a} (noted as $n-a$ in required elements file) represents the send node of process p_a .

Exemplifying the functioning of the automaton in Figure 9 for All-events-s-loop criterion, consider that process p_a sends a message to p_b , and node $n_{it_a}^{p_a}$ with a send is in a single loop with $k = 2$ (i.e., the send must be executed twice). The state 1 represents the initial state of the automaton, N represents the set of all the nodes in the PCFG, and $n-p$ represents node $n_{it_a}^{p_a}$. The transition function $N/n-p$ determines that any node from the PCFG except for $n-p$ can be

Figure 13 – DFA for All-s-uses-loop criterion

Figure 14 – DFA for All-s-c-uses-loop criterion

executed to stay in state 1. The descriptor that defines the automaton for the events-s-loop will stay in state 1 while reading any node from N in the trace file, and only when finding node $n - p$ the automaton will change to state 2. Given that trace files have the sequence $n^{p_a} 0^{p_b} n^{p_a}$, the subsequent node that the automaton reads is 0^{p_b} (represented by $0-p$ and belongs to N), executing the transition to change to state 3. Finally, the $n-p$ pattern is found again and the automaton reaches acceptance state 4, recognizing the pattern that describes one iteration of the event-s required element.

Nodes from the PCFG $N/n-p$ can be exercised until detecting another occurrence of node $n-p$, then, the automaton returns to state 2 and the process to recognize required elements is repeated until reaching the acceptance state 4, covering the second iterations of the event-s node. The other automata follow the equivalent described functioning, and will not be described in detail in this work.

Figure 15 – DFA for All-s-p-uses-loop criterion

Take for example Figure 9 for All-events-s-loop criterion. The regular expression representing the required element recognized by the automaton, corresponds to $[N^* n-p N^* n-p N^*]^k$. It indicates that any node of the program can be executed (excluding the node with the send primitive) followed by the send node $n-p$, subsequently expecting the execution of any node from N . Another occurrence of node $n-p$ is required, considering the previously exposed pattern of the send node. The regular expression allows to recognize all k iterations present in the trace file.

Table 10 presents the all the regular expressions accepted by automata descriptors and the pattern of the required elements for all criteria. The set N represents the set of nodes where n is a specific node, p_a, p_b are processes, n, m, s are nodes with communication events and x are nodes with variable definitions. $N_p v_n$ are sets of nodes with no re-definition of $def(x)$. The node y and the tuple (u, t) represent a c-use and p-use of a variable, respectively. The *Regular expression* column describes the regular expressions that represent the patterns that the automaton accepts, and defines the implementation of the automaton descriptors. *Required elements pattern* column describes the pattern in which the required elements are generated by ValiElem.

When a required element is inside a nested loop, we followed the same pattern. A DFA representing an automaton is capable to interpret any number of repetitions for a required element. The ValiMPI loop structure contains the information required to determine the depth d of a nested loop.

The described approach for the generation of required elements could have a different execution. One of the possible solutions we contemplated regarding the required elements pattern was to simply incorporate the iteration number. However, the iteration number would also have to be added to the automaton description file (generated along with the required elements file). When adding the iteration element to the pattern, it impacts ValiEval module since the functions that compose and recognize the automata have been created to understand

the standard pattern of *node-process*. Furthermore, ValiExec generates trace files considering such standard. Thus, in favor of the patterns to match among modules, it would be necessary to modify the automaton interpretation and the functions that set nodes in trace files, which leads to change the generation of required elements for all of the previous criteria. In consequence, such approach was not implemented because of the cost of modifications required in ValiExec, for refactoring of previous criteria in ValiElem and automaton implementation in ValiEval, which goes against the premise we chose to follow to implement our criteria.

Table 10 – Loop structural testing criteria - Regular expression and required elements pattern

Criteria	Regular expression	Required elements pattern
All-events-s-loop	$N^* [n-p \ N^* \ n-p \ N^*]^k$	$n-p-it_p$
All-events-r-loop	$N^* [n-p \ N^* \ n-p \ N^*]^k$	$n-p-it_p$
All-sync-events-loop	$N^* [m-p_b \ n-p_a \ m-p_b \ N^*]^k$	$n-p_a-it_b, \ m-p_b-it_b$
All-defs-recv-loop	$N_p^* \ x-p \ [N_p^* \ x-p \ N_p^*]^k$	$x-p-it_p, \ def(x)$
All-s-uses-loop	$N_{p_a}^* \ x-p_a \ [N_{p_a}^* \ n_v^* \ 0-p_b \ n-p_a \ N_{p_a}^*]^k \text{ AND } N_{p_b}^* \ n-p_a \ [m-p_b \ N_{p_b}^*]^k$	$x-p_a-it_a, \ (n-p_a-it_a, m-p_b-it_b), \ def(x)$
All-s-c-uses-loop	$N_{p_a}^* \ x-p_a \ [N_{p_a}^* \ n_v^* \ 0-p_b \ n-p_a \ N_{p_a}^*]^k \text{ AND } N_{p_b}^* \ n-p_a \ [m-p_b \ N_{p_b}^* \ n_v^* \ y-p_b \ N^*]^k$	$x-p_a-it_a, \ (n-p_a-it_a, m-p_b-it_b), \ y-p_b-it_b, \ def(x_a), \ def(x_b)$
All-s-p-usos-loop	$N_{p_a}^* \ x-p_a \ [N_{p_a}^* \ n_v^* \ 0-p_b \ n-p_a \ N_{p_a}^*]^k \text{ AND } N_{p_b}^* \ n-p_a \ [m-p_b \ N_{p_b}^* \ n_v^* \ u-p_b \ t-p_b \ N^*]^k$	$x-p_a-it_a, \ (n-p_a-it_a, m-p_b-it_b), \ (u-p_b-it_b, t-p_b-it_b), \ def(x_a), \ def(x_b)$

Non-executable required elements.

As a consequence of structural testing techniques implementations, there will be presence of non-executable required elements and infeasible paths. In the case of our criteria, they can be given for two main reasons. First, because the function for generation of required elements considers that a process p_a with a send primitive can send a message to all the other processes with receive primitives. This occurs because it is not possible to statically identify which processes are going to receive a message, or to identify the target node of send primitives (for example, when a variable determines the receiving process). The other reason is that the required iterations of each loop for each process are also counted, combining every possible scenario even when there are not test case able to execute them.

For example, supposing two processes p_a and p_b each one with a loop, where p_a has a variable definition in the first iteration of the loop (it_1) and a send in the first and second iterations (it_1 and it_2), and p_b has an s-use in the first and second iterations of the loop (it_1 and it_2). In such scenario, there are four required elements for the criterion All-s-uses-loop, considering the permutation between those two loop iterations. The required element that describes the variable definition in first iteration of p_a and s-use in second iteration of p_b is going to be non-executable,

due to the non-existence of a definition-clear path between the first iteration of the variable definition in p_a and the s-use in it_2 of p_b . Another non-executable required element would be the variable definition in it_2 of p_a and the s-use in it_1 of p_b , because the s-use in it_1 of p_b was covered before, matching with the first iteration of p_a .

7.2.3 ValiExec

Because we did not modify the patterns for the automaton descriptor files, we did not perform any changes in ValiExec. An adjustment in this module would impact the functionality of the existing criteria, as well as the method that is followed to read the automaton descriptor file, to create the DFA structure (with states and transitions) and the function to read and accept the regular expressions from the trace file.

Additionally, if considering a new pattern that includes the iteration number for the automaton descriptor, the trace file would have had to be adapted as well, since the expression that represents required elements does not contemplate the iteration number. Thus, when generating the trace file, which is analyzed by ValiEval to calculate coverage, the pattern must match with the function reading the automaton descriptor, demanding a modification in ValiExec module and a complete transformation of the behavior of existing criteria.

Incorporating the iteration number to the existing criteria pattern would require re-writing every function to generate and read required elements, whereas they identify the nodes, edges and def-use pairs in paths and between processes with no loop paths. The ValiMPI functions created for each MPI function in ValiExec would have to be modified as well.

7.2.4 ValiEval

Automatons for criteria validation.

ValiMPI reads files with descriptors to create a structure that represents the states and transitions of the automaton. For our criteria, it is expected that each line of the file contains the number of required iterations for each required element. Therefore, we made additional validations to avoid conflicts when reading the files with descriptors for the previous criteria.

Trace files and path recognition.

The required elements of the proposed criteria consider an additional element, the iteration number. Even when maintaining the generation and the structure of the trace files intact, we are capable of recovering the amount of iterations performed from the execution trace file. Through the recognition of automatons it is possible to validate the number of occurrences of a required element and determine if it reached the acceptance state and how many times, in order to obtain the number of executed iterations.

Initially, the ValiEval function in charge of changing the automaton states through the transitions reading a string accepts a pattern if and only if with such string the automaton reaches the acceptance state (defined in descriptor file). The state transition is performed considering the conditions established by the automaton descriptor, so if the string pattern is not recognized it returns to the initial state.

In relation to our novel proposed criteria, when a trace string reaches the final or acceptance state and the automaton continues reading the trace file sequence, it could carry out a transition to another state causing it to enter into a non-acceptance state when it finishes reading the trace path. In such scenario, when finding another occurrence of the string after its first appearance, the automaton will not recognize a required element that has been executed more than once. Those are the communication events in loops we are interested in executing.

Our solution is to verify when the automaton entered to the final state for the first time, so that it is not limited to the first occurrence of the required element, and hence the strings from other iterations of a loop would also be considered and accepted. Simultaneously, this outcome represents the number of executed iterations according to the trace. We can compare this data to the number of required iterations and obtain the coverage of required elements.

Coverage evaluation.

We consider two factors to evaluate the coverage of criteria: the number of processes involved and the number of loop iterations for those processes.

The number of executed iterations by each process for each required element determines the number of re-executed communication events. In other words, if we have $k = 2$ where process p_a is not in a nested loop with $d = 1$ with $k^d = 2^1$, and process p_b has $d = 2$ with $k^d = 2^2$, then there will be a total of eight (8) required elements with the combinations between the iterations of each process.

In Table 11, each line represents a required element for the All-sync-events-loop criterion. See that process 1 has four iterations and processes 0 and 2 have two iterations each.

Table 11 – Required elements for All-sync-events-loop criteria

<i>node-process-iteration</i>		<i>node-process-iteration</i>
12 - 0 - 1	,	3 - 1 - 1
12 - 0 - 2	,	3 - 1 - 2
12 - 0 - 1	,	3 - 1 - 3
12 - 0 - 2	,	3 - 1 - 4
5 - 1 - 1	,	10 - 2 - 1
5 - 1 - 2	,	10 - 2 - 2
5 - 1 - 3	,	10 - 2 - 1
5 - 1 - 4	,	10 - 2 - 2

For loop structural testing criteria the evaluation of coverage (relation of covered required items over the number of executed elements) was done differently. ValiEval contemplates that replicated required elements are recognized by the same automaton, as long as the executed iterations have covered such required elements.

The automaton file contains the number of required iterations of each event, so that the outcome of the automaton recognition allows the verification of which required elements were covered after running the program. We compare how many times a communication event was executed in the program (through the trace file), and how many iterations were actually required (from automaton descriptor file), in order to calculate the coverage.

After validating if the automaton recognized the string of the events or not, the covered and uncovered required elements are shown to the user. This is because an automaton might be recognized for an event, but it does not mean that all of the required elements replications have been executed. For example, for four loop iterations, only two could have been actually executed and covered.

Figure 16 shows an example of the outcome for ValiEval after executing a test case for All-sync-events-loop criterion.

Figure 16 – ValiEval results for All-sync-events-loop

```
-- COVERED REQUIRED ELEMENTS --
1) 20-0-1 5-1-1, covered by valimpi/test_case0000, sync 0
7) 22-0-1 5-2-1, covered by valimpi/test_case0000, sync 0
9) 19-1-1 26-0-1, covered by valimpi/test_case0000, sync 0
13) 19-2-1 26-0-1, covered by valimpi/test_case0000, sync 0

-- NON-COVERED REQUIRED ELEMENTS --
2) 20-0-2 5-1-2
3) 20-0-1 5-2-1
4) 20-0-2 5-2-2
5) 22-0-1 5-1-1
6) 22-0-2 5-1-2
8) 22-0-2 5-2-2
10) 19-1-2 26-0-2
11) 19-1-1 5-2-1
12) 19-1-2 5-2-2
14) 19-2-2 26-0-2
15) 19-2-1 5-1-1
16) 19-2-2 5-1-2

Number of covered required elements: 4

Number of non-covered required elements: 12

Coverage for the all-sync-events-loop criterion: 25.00%
```

7.3 Final considerations

We presented the implementation process of the proposed structural testing criteria in ValiMPI. There were necessary modifications on the paradigm that has been followed for the

generation of required elements, as well as for the evaluation of covered events.

Analyzing the loop paths in concurrent programs is imperative, since possible sub-paths can increase in number if not applying an adequate technique to select them. Other aspect in this context is the complexity of the solution to find loop paths as it can reduce efficacy, for example, implementing redundant mechanisms to select paths consuming extra computational resources beyond the necessary for an effective solution.

Accordingly, we examined different techniques and selected the one that had less impact on the development and quality of the tool, regarding the results on the generation of required elements and automaton descriptors. On the other hand, the recognition of required elements based on our criteria can be represented by a regular expression, accepted by a Deterministic Finite Automaton. We defined automata for the criteria contemplating the sequence of patterns in the trace files generated by ValiEval.

The criteria implementation process allowed us to confirm that the complexity of analysis, design and development for structural criteria increases when involving loop paths and events association. We ran several tests (regression, functional and system) during and on development completion to look for defects after our modifications, checking the expected results with some of the benchmarks. We had manually generated required elements for such programs basing on the new criteria and comparing the expected results of each module. The trace files provided the information of covered elements, allowing us to verify the number of executed iterations (occurrences of required elements patterns in the file string) to review the results of ValiEval regarding the shown covered required elements.

The approaches, methods and design that we followed helped us to improve the developed functionalities in ValiMPI for the proposed criteria.

EXPERIMENTAL VALIDATION

8.1 Initial considerations

This chapter presents the experimental evaluation of the proposed criteria, with the methodology, execution, results and analysis of results. We also introduce the process of selection of benchmarks to validate the test model, showing their characteristics and purpose for the experiments. Section 8.1 presents the benchmarks and their features, Section 8.2 introduces the experimental design considering the objectives, metrics, experimental setup and threats to validity. In Section 8.3 we present the results of the experiments execution and their analysis, considering the defined metrics. Finally, Section 8.4 has a discussion of the results and concluding remarks on the performed experimental study.

8.2 Benchmarks

The objective of selecting a set of message-passing programs is to test the proposed criteria. The benchmarks must involve relevant aspects that allow the evaluation of criteria, such as diverse communication events and process interactions inside loops and/or nested loops.

We select programs developed by different sources to avoid that knowledge on the program interferes in the selection of test cases. We choose programs that match to ValiMPI prerequisites regarding source code limitations, and that also do not require modifications to be executed in our testing environment. Our benchmarks are programs designed for distributed memory architectures involving the presence of communication primitives in repetition structures, so they can be evaluated using the new criteria. The existence of nested loops allows verifying the behaviour of criteria and the impact in the generation of required elements.

We consider the characteristics of parallel benchmarks exposed in ([DOURADO, 2015](#)), to select, classify and prioritize the programs regarding complexity, synchronization and com-

Table 12 – Features of benchmarks analyzed during selection

Characteristic	Accomplishment
Easy understanding and execution	■ ■ ■
Level of interactions among processes	■ ■ ■
Simple outputs	■ ■ □
Test cases	□ □ □
Cyclomatic complexity	■ ■ ■
Flexibility	■ ■ □
Scope of communication/synchronization primitives	■ ■ □
Event based vision	■ ■ ■
Non-determinism	■ ■ ■
Standardized coding	■ □ □
Execution environment	■ ■ ■
Pseudo-code	■ □ □
Standard documentation	■ ■ □
Portability	■ ■ □
Free from defects	■ ■ ■
With known defects	□ □ □

munication events, process interaction, communication patterns, and non-determinism. We also contemplated open-source code programs. Table 12 presents the characteristics with the level of compliance of the set of benchmarks we selected.

The benchmarks are C/MPI programs solving mathematical and computational problems, focusing on the communication and synchronization events among processes. A higher complexity in the programs does not necessarily benefit the results of the evaluation of criteria, but on the other hand, having understandable and manageable programs allows a more appropriate analysis and comprehension of the criteria implementation. Considering the size of the programs and their complexity, the level of interactions among processes is high, as can be seen in Table 13 with the number of send and receive primitives, meaning there are important communications with a flexible number of processes.

The outputs of the benchmarks are readable and comprehensible, even though we have some programs requiring additional analysis during verification, such as Search number, Jacobi, Philosophers dining, and Trapezoidal rule. The set of benchmarks initially does not have a known set of test cases, which for our objective is desirable as we intend to generate test cases basing on the proposed criteria. After this empirical study execution the benchmarks will have a known set of test cases.

Most of the programs are flexible regarding the number of processes to execute test cases. However, programs with a fixed number of processes present a suitable complexity in the

interactions in loops. The scope of the MPI primitives is limited to the message-passing memory paradigm and point to point communication pattern for all programs.

Given the nature of our proposed criteria, the benchmarks are event oriented, as every iteration in the loop represents a different communication event between processes. Non-determinism is present in most of the programs as well.

As benchmarks come from different sources, there is no standardization with coding conventions neither with code documentation. However, we build documentation for the selected programs in order to guide the reproducibility of the experiments using those programs (see Appendix A). Some of the programs already have pseudo-code descriptions.

The empirical protocol for this project provides the characteristics of the execution environment for the programs, as we configured an arranged settings for the benchmarks. They were normalized to run in ValiMPI according to the requirements of the tool to instrument and execute programs.

All of the selected benchmarks are known-defect-free, functioning according to the specifications of the code. For this experimental design we consider the injection of defects in the code using a mutation-based methodology, in order to avoid bias.

Table 13 shows more detailed information of the eleven programs used in our experiments. They belong to open-source sets of testing benchmarks, as well as free access programs. The column *Developed by* corresponds to the research group or university affiliation of the author/s of the program, and we provide the links where we retrieved the code. We measured the Cyclomatic Complexity and the number of lines of code (LOC) with the tool CCCC¹. The number of processes allowed for benchmarks execution is given by the limitations of each program's source code regarding the number of processes when running with MPI.

---

¹ <http://cccc.sourceforge.net/>

¹ Paulo S. L. de Souza

² M. Quinn adaptation. Greg Andrews. Available on: <https://www2.cs.arizona.edu/greg/mpdbook/programs/jacobi.mpi.c>

³ Ian Finlayson. Available on: <http://cs.umw.edu/finlayson/class/fall14/cpsc425/notes/18-sorting.html>

⁴ Pacheco adaptation. Mario Machado

⁵ Pacheco adaptation. Available on: https://www.cs.usask.ca/spiteri/CMPT851/notes/demos/mpi_trap2.c

⁶ Blaise Barney, adapted from Ros Leibensperger, Cornell Theory Center. Available on: https://computing.llnl.gov/tutorials/mpi/samples/C/mpi_mm.c

⁷ John Burkardt. Available on: https://people.sc.fsu.edu/jburkardt/c_src/mpi/mpi.html

⁸ Greg Andrews. Available on: <https://www2.cs.arizona.edu/greg/mpdbook/programs/jacobi.mpi.c>

⁹ Alexandre Hausen

Table 13 – List of C/MPI message-passing benchmarks

Program name	No. of Send	No. of Receive	Developed by	No. Loops	Cyclomatic Complexity	LOC	No. of Processes
GCD (Two slaves)	5	2	ICMC (DOURADO, 2015) (adapted from Java)	3	17	80	3
GCD (Three slaves)	5	3	ICMC (DOURADO, 2015) (adapted from Java)	1	10	68	4
Global sum	1	1	ICMC ²	1	7	37	Flexible
Sieve of Eratosthenes	1	1	U. of Arizona ³	1	22	123	Flexible
Odd even sort	3	3	U. Mary Washington ⁴	1	32	134	Flexible
Sieve of Eratosthenes (fixed np)	3	3	ICMC ⁵	2	42	260	4
Trapezoidal rule	4	4	U. of Saskatchewan ⁶	4	9	68	Flexible
Matrix multiplication	7	7	Cornell University ⁷	2	15	87	≥ 2
Search number	5	4	Florida State University ⁸	3	13	138	Flexible
Jacobi	4	4	U. of Arizona ⁹	4	27	138	$grid\_size * strip\_size$
Philosopher's dining	3	8	ICMC ¹⁰	3	20	115	Flexible

8.3 Experimental design

This section presents the empirical protocol followed before executing the experiments with the selected benchmarks.

8.3.1 Objective

The objective of this empirical evaluation is to demonstrate that the proposed structural testing criteria contribute to identifying defects in concurrent programs, in the presence of communication events among processes that are inside loops, having possible defects in the different iterations. The criteria we are evaluating are: *All-events-r-loop* (AERL), *All-events-s-loop* (AESL), *All-sync-events-loop* (ASEL), *All-defs-r-loop* (ADRL), *All-s-uses-loop* (ASUL), *All-s-c-uses-loop* (ASCUL), and *All-s-p-uses-loop* (ASPUL).

To accomplish the objective, we verify the applicability of criteria concerning the application cost, effectiveness and strength. These metrics allow to find out how a tester can implement

the criteria and at what cost, in order to perform a better test planning. The main concern is to learn if criteria help to reveal defects and to analyze the viability of their use.

The experiments are performed from the perspective of a tester, using criteria to select test cases to a program that has been developed by another person, however knowing the logic underneath the program.

8.3.2 Metrics

We evaluate the application cost, effectiveness, and strength of the proposed criteria, in order to verify to what extent their implementation have an impact on the mentioned measurements by analyzing the experimental results.

The application cost is determined by:

- Number of adequate test cases that are necessary to satisfy a criterion (also known as adequate test set size).
- Number of required elements.
- Number of non-executable required elements.

In this case, the computational cost is not relevant since the objective is to compare the proposed criteria among each other verifying their applicability, rather than the performance of the tool. We do not consider the existent criteria (SOUZA *et al.*, 2008) in this empirical evaluation for comparison, being it a potential future work and bearing in mind that there are criteria that are not implemented yet in ValiMPI.

The effectiveness is given in terms of:

$$Effectiveness = \frac{Number\ of\ revealed\ defects}{Number\ of\ inserted\ defects} * 100 \quad (8.1)$$

It determines the capability of a criterion to detect the most significant quantity of defects.

The strength is given regarding the coverage percentage compared between two criteria, and refers to the probability of satisfying a criterion (MELO *et al.*, 2017; BRITO, 2011). The set of test cases of a given criterion C_1 is applied to another criterion C_2 , evaluating the coverage of C_2 in contrast with coverage of C_1 and its original test case set.

In other words, a set of test cases TCS_1 that is adequate to C_1 is compared to the obtained coverage when applying TCS_1 to $C_2, C_3, \dots, C_6$. If $Coverage(C_6) > Coverage(C_1)$, then C_1 has a lower strength compared to C_6 . This situation is because when the set TCS_1 achieves higher coverage with C_6 than with the criterion it was generated for (C_1), it means C_1 has the probability of revealing a fewer quantity of errors than C_6 with the same test cases.

8.3.3 Object definition

The benchmarks are the objects of the experiments. These programs belong to the set of benchmarks selected for the evaluation of the proposed criteria (see Section 8.2).

8.3.4 Hypotheses formulation

Based on the objective of the empirical evaluation, the research question for this empirical evaluation is:

RQ: *What are the effectiveness, application cost and strength of the proposed structural testing criteria for message-passing programs?*

Considering that the proposed set of criteria is composed by $C_i = \{C_1, C_2, C_3, C_4, C_5, C_6, C_7\}$ with $1 \leq i \leq 7$, and TCS_i is the adequate test case set of a criterion C_i , the formulated hypotheses are:

Null Hypothesis 1 (NH1): The application cost of the analyzed criteria is the same, in terms of the metrics defined in Subsection 8.3.2.

$$NH1: Cost(C_i) = Cost(C_{i+1})$$

Alternative Hypothesis 1 (AH1): The application cost is different for at least one of the analyzed criteria, in terms of the metrics defined in Subsection 8.3.2.

$$AH1: Cost(C_i) \neq Cost(C_{i+1})$$

Null Hypothesis 2 (NH2): The effectiveness of the analyzed criteria is the same, in terms of the metrics defined in Subsection 8.3.2.

$$NH2: Effectiveness(C_i) = Effectiveness(C_{i+1})$$

Alternative Hypothesis 2 (AH2): The effectiveness is different for at least one of the analyzed criteria, in terms of the metrics defined in Subsection 8.3.2.

$$AH2: Effectiveness(C_i) \neq Effectiveness(C_{i+1})$$

Null Hypothesis 3 (NH3): No criterion subsumes another criterion, regarding the strength of the analyzed criteria as defined in Subsection 8.3.2.

$$NH3: TCS_i \not\subset TCS_{i+1}$$

Alternative Hypothesis 3 (AH3): There is at least one criterion that subsumes another criterion, regarding the strength of the analyzed criteria as defined in Subsection 8.3.2.

$$AH3: TCS_i \subset TCS_{i+1}$$

8.3.5 Variable selection

Independent variables: structural testing criteria, approach to generate test cases, and the approach to inject errors in the code.

Dependent variables: cost, effectiveness y strength.

8.3.6 Experimental setup

This section presents the steps to execute the experiments, considering the metrics, hypotheses, variables and objects.

- i **Preparation of the experiments.** Before the execution of the experiments, it is necessary to prepare the testing environment, configuring the ValiMPI tool. Additionally, we considered the computational requirements for the execution of the programs in the tool. The configuration for ValiMPI execution comprehends:
  - Ubuntu 16.4.10 Virtual Machine, 6 GB RAM, Intel Core i7-7500U (physical host).
  - gcc 5.4.0
  - OpenMPI 1.4.4.
  - ValiMPI source code.
- ii **Injection of defects in benchmarks.** In order to verify the effectiveness of the proposed criteria, we generated faulty programs from the selected benchmarks. This approach allows us to demonstrate if our criteria can find an error that we previously know. The types of defects that we injected into the programs focus mainly on communication primitives inside loops. We also consider other common concurrency errors to verify if the proposed criteria are appropriate to find other types of errors as well. We can identify concurrency defects according to the Table 3.

The errors were manually inserted considering the bug taxonomy previously presented in this work (see Section 5).

We create a new version of each original defect-free program for each one of the inserted defects using mutation testing (CARVER, 1993), where every new defect generates a distinct program (mutant) from the original. More than one defect in the same program version would make difficult to verify which one of the defects produces a wrong output. We followed the mutation operators related to MPI communication primitives, proposed in (SILVA, 2013), such as *ReplModeSend*, *ReplModeRecv*, *ChanArg*, *ReplSource*. *ReplModeSend* operator replaces the mode of a send in the program, for example, having a blocking *Send* to a non-blocking send *Isend*, and *ReplModeRecv* operator works similarly to but for receive primitives. *ChanArg* and *ReplSource* switches arguments in and replace the source parameter in send/receive primitives respectively. There are errors that appear when specific scenarios are presented in the communications between two processes, which lead us to insert such scheme into the code. This scenario refers to inter-process data race defects (see Section 5.3).

Table 14 shows the number of defects inserted in each benchmark.

Table 14 – Number of injected defects in each program

Program	Type of error							Total
	Observability	Inter-process data race	Switched variable	Changed arguments	Any tags/source	Non-blocking send/recv	Buffer integrity	
GCD (Two slaves)	1	1	1	1	2	1	1	8
GCD (Three slaves)	1	0	1	1	1	1	0	5
Global sum	2	1	1	0	0	1	1	6
Sieve of Eratosthenes	0	0	2	1	1	1	1	6
Odd even sort	0	1	0	2	1	1	1	6
Sieve of Eratosthenes (fixed np)	0	1	0	0	1	3	0	5
Trapezoidal rule	0	0	0	0	2	2	0	4
Matrix multiplication	0	1	1	1	1	2	1	7
Search number	1	1	0	0	1	2	1	6
Jacobi	1	2	0	1	1	0	1	6
Philosopher's dining	1	1	0	1	1	2	0	6
Total	7	9	6	8	12	16	7	65

- iii **Generation of the set of test cases.** The purpose of this step is to obtain the set of adequate test cases for each program, looking for the highest coverage possible, which will depend on the quantity of non-executable required elements. From this step, we collect application cost data.

Additionally, there will be no variation in the number of processes for the generation of required elements and test cases, because changing the number of processes it is necessary to create a new test session. ValiElem, ValiExec and ValiEval modules require the number of processes to perform their functions and changing it implies a different number of required elements. Thus, evaluating the coverage for varying number of processes does not bring relevant results on criteria effectiveness.

For the programs that do not require input data, the execution of different paths to cover the remaining required elements depends on the execution of different synchronization

pairs, which should be approached by using deterministic or controlled executions. To address the lack of controlled execution in ValiMPI, we will execute each test case thirty times to give the opportunity of other synchronization sequences to execute. Furthermore, for programs with no input data, the coverage evaluation might be minor compared to the programs with input data, so we will analyze the uncovered required elements in such case.

An increase in the number of processes would be more related to the scalability evaluation, both for ValiMPI and criteria, as the application cost increases and the capacity of the tool to execute more large programs are in consideration. The scalability evaluation, however, is not part of the objectives of this experimental design.

In such a way, the following steps are required to run the experiment:

- a) *Generation of required elements*: for each criterion and for each benchmark, using the ValiElem module. This process is performed on the original defect-free programs, in order to obtain the required elements that must be covered by the test cases. We can retrieve information for cost evaluation, having the number of required elements, non-executable elements, and size of adequate test sets.
- b) *Selection of test cases*: for each criterion and for each original program, aiming to cover all of the required elements. For this step, we generated the test cases from the perspective of a tester, considering equivalence classes (DELAMARO; MALDONADO; JINO, 2007) to select the first data set basing on the required elements generated by criteria. The first test case is selected according to the criterion obtaining an initial coverage, and then new test cases (also guided by criteria) are generated aiming for increasing coverage.

Thus, the selected test cases would be adequate but not redundant.

- c) *Execution of test cases*: for each criterion and for the original programs, using the ValiExec module. We obtain the results from program executions with the test cases, verifying the correctness of the outputs (without modifying the source code) and allowing to evaluate the code coverage in next step.
- d) *Evaluation of coverage of test cases*: using ValiEval module, we verify the partial coverage for each test case. In cases with uncovered required elements, new test cases should be created and added to the set of test cases for a determined criterion. That is to say, return to step b) until obtaining the highest coverage possible. The partial coverage for each new test case is registered to verify when a test case does not increase the coverage percentage, so that it is not included in the adequate set, and other test case is created.

At this point we have the set of adequate test cases for each criterion.

- e) *Identification of non-executable required elements*: this is done manually, noting down the criterion in which these elements appear and why. Non-executable required elements should be analyzed when the coverage of the program is not increasing.
- iv **Cost evaluation.** Having executed the previous step we obtain the information necessary to assess the application cost for each criterion.
- v **Effectiveness evaluation.** We execute the faulty versions of the programs with their set of adequate test cases. Each execution is repeated fifteen (15) times on a first instance, and then another fifteen times for a total of thirty (30) executions per test case, with the purpose of verifying the results in the case that different synchronizations are executed. If there are different synchronizations and results, they will be analyzed.

To verify the effectiveness, we analyze the coverage for the faulty programs, analyzing whether the inserted error has been revealed or not using the criteria.
- vi **Strength evaluation.** Execute the set of test cases of a criterion for the rest of criteria, obtaining the coverage for the different criteria. This process must be performed for every criterion so that we can compare the coverage among criteria and evaluate their strength.

8.3.7 Threats to validity

We identify the threats to validity for this empirical study, exposing the actions to mitigate them. The methodologies we followed allow us to consider that these aspects do not compromise the results and contributions of our research.

- **Conclusion validity**

The conclusions of the empirical study are determined by the statistical methods to evaluate the dependent variables. We based on the characteristics of the data, the treatments and guidelines on statistical methods to test the hypotheses, as well as the consultation of a specialist in the area.

- **Internal validity**

It is related to the means to analyze the factors (the metrics), which is the ValiMPI tool, and the validity of its results. Since we performed modifications in the tool to implement the proposed criteria, there might be threats to the obtained results because of the adjustments in the modules. To mitigate this threat, we tested several times the modifications with all of the scenarios considered by criteria, and we verified correctness and cohesiveness of the required elements generated by ValiElem, as well as the coverage obtained with ValiEval.

The method to inject defects into the code might also introduce threats to the internal validity. To reduce this bias, we made use of mutation testing to insert concurrency defects of different types, to avoid preference of specific errors and to guide the selection of test cases.

- **Construct validity**

The knowledge over the programs could affect the application of testing criteria. We selected programs from different sources and developers, in order to guide the experiments acting as a tester. The methodology applied to select test cases makes that this activity relies only on the use of testing criteria, and not on the knowledge of the programs.

- **External validity**

This experimental study could be replicated for empirical studies focused on structural testing of concurrent programs, by following the experimental design. However, generalizing the results for different objects could not be convenient. Despite we can not ensure the use of all possible scenarios to verify criteria applicability, we contemplated message-passing concurrent programs with different characteristics and communication interactions, intending to involve situations found in real parallel applications. The principal objective of our study is to propose new structural testing criteria for concurrent programs in the presence of loops, and the experimental evaluation allows us to demonstrate the applicability of criteria.

8.4 Execution of experiments and analysis of results

This phase of the experiments presents the results of the execution of the benchmarks in the ValiMPI tool, in terms of the cost, effectiveness and strength. We obtain the coverage for criteria for each program with their adequate set of test cases. Table 15 shows the coverage obtained by criteria for each program, note that we consider the non-executable required elements, as ValiEval contemplates non-executable elements to perform coverage percentage calculation. Criteria that has coverage as "-" in Table 15 did not generate required elements for the program (due to program's structure), which is different to 0% of coverage. An analysis of non-executable required elements produced by our proposed criteria is performed further in this section. We find that for all criteria, a 100% coverage was reached when contemplating only executable required elements.

Additionally, we present the analysis of the results along with the outcomes of the experiment executions.

8.4.1 Cost results

For the original programs, we executed ValiMPI modules in order to obtain the number of required elements for each criterion, as well as the size of the set of adequate test cases. We also identified the number of non-executable required elements.

Table 15 – Coverage with adequate set of test cases

Program	All-events-r-loop	All-events-s-loop	All-sync-events-loop	All-defs-rcv-loop	All-s-uses-loop	All-s-c-uses-loop	All-s-p-uses-loop
GCD (Two slaves)	100.00%	-	-	100.00%	-	-	-
GCD (Three slaves)	100.00%	87.50%	37.50%	100.00%	37.50%	31.35%	-
Global sum	30.00%	40.00%	10.00%	30.00%	1.25%	5.00%	-
Sieve of Eratosthenes	25.00%	-	-	25.00%	-	-	-
Odd even sort	38.89%	41.67%	8.33%	38.89%	1.74%	0.00%	0.00%
Sieve of Eratosthenes (fixed np)	100.00%	100.00%	100.00%	100.00%	-	-	-
Trapezoidal rule	33.33%	33.33%	4.27%	33.33%	-	-	-
Matrix multiplication	25.00%	25.00%	1.79%	25.00%	0.00%	0.00%	-
Search number	33.33%	41.67%	5.21%	33.33%	33.33%	66.67%	-
Jacobi	77.78%	77.78%	12.96%	77.78%	6.19%	0.00%	0.00%
Philosopher's dining	100.00%	75.49%	10.33%	100.00%	100.00%	100.00%	75.00%

Table 16 shows the number of required elements for each criterion and program with the number of processes (NP), and also the number of non-executable elements that we found after executing the test cases and evaluating the remaining uncovered elements. There were criteria that did not generate required elements for a determined program, mostly for the *All-s-p-uses-loop* as can be seen in the Table 16. This situation is given because the scenario in which a *s – use* is inside a loop (in both processes), and the receiving process has a subsequent *p – use* is not present in all of the programs. As we selected the benchmarks, we identify characteristics such as communication primitives inside loops and different functions. However, we did not analyze each program for the criteria to actually generate required elements for all of them, so that we would not be inserting a bias in the results by selecting favorable programs.

Table 16 – Required elements/non-executable required elements

Program	NP	Criteria						
		<i>All-events-r-loop</i>	<i>All-events-s-loop</i>	<i>All-sync-events-loop</i>	<i>All-defs-rcv-loop</i>	<i>All-s-uses-loop</i>	<i>All-s-c-uses-loop</i>	<i>All-s-p-uses-loop</i>
GCD (Two slaves)	3	6/0	8/1	16/10	6/0	16/10	32/22	0/0
GCD (Three slaves)	4	2/0	0/0	0/0	2/0	0/0	0/0	0/0
Global sum	5	10/7	10/6	40/36	10/7	80/79	20/19	0/0
Sieve of Eratosthenes	4	8/6	0/0	0/0	8/6	0/0	0/0	0/0
Odd evensort	3	18/11	12/7	72/66	18/11	288/283	168/168	192/192
Sieve of Eratosthenes (fixed np)	4	10/0	2/0	6/0	10/0	0/0	0/0	0/0
Trapezoidal rule	3	6/4	18/12	144/138	6/4	0/0	0/0	0/0
Matrix multiplication	4	24/18	32/24	672/660	24/18	288/288	32/32	0/0
Search number	3	6/4	24/14	192/182	6/4	24/16	36/12	0/0
Jacobi	4	18/4	18/4	108/94	18/4	840/788	96/96	96/96
Philosopher's dining	6	42/0	102/25	920/825	42/0	160/0	64/0	192/48
Totals	43	150/54	226/93	2170/2011	150/54	1696/1464	448/349	480/336

Table 17 shows the number of adequate test cases for each criterion. For the criteria that did not generate required elements, we did not create test cases, which is represented by a hyphen. On the other hand, when no test case achieved any coverage for a criterion because all of the required elements were non-executable, the number of adequate test cases is zero.

Table 17 – Number of adequate test cases

Program	Criteria						
	AERL	AESL	ASEL	ADRL	ASUL	ASCUL	ASPUL
GCD (Two slaves)	1	2	2	1	2	3	-
GCD (Three slaves)	1	-	-	1	-	-	-
Global sum	1	1	1	1	1	1	-
Sieve of Eratosthenes	1	-	-	1	-	-	-
Odd even sort	1	-	-	1	-	-	-
Sieve of Eratosthenes (fixed np)	2	1	2	2	-	-	-
Trapezoidal rule	1	1	1	1	-	-	-
Matrix multiplication	1	1	1	1	0	0	-
Search number	1	4	4	1	4	4	-
Jacobi	2	2	2	2	4	0	0
Philosopher's dining	3	6	6	3	3	2	2

Most of the non-executable required elements are produced by interactions between processes that do not have communications. For example, a program with a master process and three slave processes will have required elements generated for interleavings among the slaves, even when those communication events only occur with the master process. This situation also appears when there is not a defined function for the processes, and at the time of the instrumentation, all of the processes consider only the *main* function. It generates for any criterion required elements that are non-executable.

Some programs do not have an specific function for determined processes depending on their rank, creating required elements for infeasible paths. If there is a program with only a *main* function differentiating the execution of processes with conditional statements, all processes will execute *main* function, so when generating required elements ValiElem contemplates the entire *main* structure. For example, for the *All-s-p-uses-loop* criterion with required elements, in *Odd even sort* and *Jacobi* programs no test case achieves coverage because of interactions that never occur.

In other cases, when there were zero required elements generated for the criteria involving communication between two processes (such as ASUL, ASCUL and ASPUL), we verified that it is because the send or the receive primitive was not inside a loop, and thus the required element is not created. However, if an *s – use* is not in a loop and matches a receive (in a loop of another process), our criteria does not contemplate that situation, since the existing criteria covers that association.

Figure 17 – Cost for required elements for criteria

Figure 17 shows the number of required elements generated for each program by each criterion. As expected, programs that have a higher quantity of send and receive primitives, as well as the number of loops, have a greater number of required elements. However, as seen in Figure 19 with the number of non-executable required elements, many of those generated elements belong to associations that cannot be covered by any test case. Figure 18 shows the number of test cases required for each program to obtain the highest possible coverage percentage.

8.4.2 Effectiveness results

We inject a set of defects in the original programs and generate new versions of them for each defect. The goal is to be able to determine an approximate on the effectiveness of the proposed criteria.

We executed the faulty programs in ValiMPI, having a test project session for each one. Using the adequate test cases for each criterion, we registered the outputs and the coverage after the tests set execution. Note that the obtained coverage percentage also includes non-executable required elements for all criteria. The results of executions were examined in order to verify if the error did show up. We did not consider a defect as revealed when it did not generate any required elements when applying the adequate test cases for a criterion for a program. This scenario would cause inconsistencies because when having no adequate test cases for non-existent required elements, there is no possibility of revealing any defect. However, we did contemplate a defect

Figure 18 – Cost for number of adequate test cases for criteria

Figure 19 – Cost for non-executable required elements for criteria

revelation when applying the adequate test cases and not obtaining any coverage, but the error has been shown. An example is when a deadlock occurs. It is because at the moment we use the test cases guided by criteria, we are using the adequate data to cover the required elements and thus find a possible existing error.

Table 18 shows the effectiveness of the criteria for the different types of errors we tested,

using the equation 8.1. Figure 20 shows the graphic illustration of the mentioned table. The ASPUL criterion is the one that achieved the fewest effectiveness, which happens because many of the programs did not have required elements, or generated non-executable elements for interleavings between processes with unmatched synchronizations. This condition is presented mostly in programs that do not have a function for processes.

Table 18 – Criteria effectiveness

Type of error	All-events-r-loop	All-events-s-loop	All-sync-events-loop	All-defs-rcv-loop	All-s-uses-loop	All-s-c-uses-loop	All-s-p-uses-loop
Observability	57.14%	71.43%	71.43%	42.86%	85.71%	71.43%	28.57%
Inter-process data-race	88.89%	88.89%	88.89%	88.89%	66.67%	66.67%	55.56%
Switched loop variable	71.43%	42.86%	42.86%	71.43%	42.86%	42.86%	14.29%
Change arguments	100.00%	75.00%	75.00%	100.00%	75.00%	75.00%	62.50%
Any tag/any source	66.67%	75.00%	75.00%	66.67%	50.00%	41.67%	25.00%
Non-blocking receive/send	75.00%	56.25%	68.75%	75.00%	31.25%	31.25%	18.75%
Buffer integrity	71.43%	85.71%	85.71%	71.43%	85.71%	85.71%	28.57%
Average	75.79%	70.73%	72.52%	73.75%	62.46%	59.23%	33.32%

Figure 20 – Criteria effectiveness

According to the results, the most effective criteria are AERL, AESL, and ASEL, which by contrasting with the cost results (Tables 16 and 17) we can observe that there are a fewer

number of generated required elements, but the adequate test cases accomplish to reveal different types of defects. The results are satisfying when we check the executed required elements, showing that errors are revealed in second iterations of loops.

There are more effective criteria for certain types of errors, considering not only the effectiveness percentage but also the consistency in revealing a defect regardless of the programs that did not have required elements for the criteria. For example, ADRL and ASEL show the error for all of the injected defects in the different programs, but the effectiveness percentage diminished because one of the programs did not have required elements generated. ASUL and ASCUL criteria also reveal most of the inserted errors, but the programs with no required elements and high quantity of non-executable required elements decreased the effectiveness as well.

The effectiveness for the ASCUL and ASPUL criteria is minor, given that they represent not very common scenarios, and in our benchmarks there are few programs that generate required elements for them. ASCUL and ASPUL criteria also have a greater quantity of non-executable elements. However, we analyze the results of the executions, and the criteria are effective to reveal particular types of defects mostly related to synchronization events and non-determinism. Examples of such defects are inter-process data race, observability, and non-blocking send/receive switching. We explain this situation in that the associations of ASCUL and ASPUL criteria are not common cases, and the percentage of errors that they can reveal, despite being fewer, is significant for concurrent programs because defects are related to data integrity and non-determinism.

8.4.3 Strength results

Tables 19 to 29 show the strength of each program with the criteria that generated required elements. We did not consider criteria with zero required elements, and the obtained coverage percentages include the non-executable required elements.

Table 19 – Strength for GCD (Two slaves)

	All-events-r-loop	All-events-s-loop	All-sync-events-loop	All-defs-rcv-loop	All-s-uses-loop	All-s-c-uses-loop	All-s-p-uses-loop
tcs-all-event-r-loop	100.00%	75.00%	25.00%	100.00%	25.00%	12.50%	-
tcs-all-event-s-loop	100.00%	87.50%	37.50%	100.00%	37.50%	25.00%	-
tcs-all-sync-events-loop	100.00%	87.50%	37.50%	100.00%	37.50%	25.00%	-
tcs-all-defs-rcv-loop	100.00%	75.00%	25.00%	100.00%	25.00%	12.50%	-
tcs-all-s-uses-loop	100.00%	87.50%	37.50%	100.00%	37.50%	25.00%	-
tcs-all-s-c-uses-loop	100.00%	87.50%	37.50%	100.00%	37.50%	31.35%	-

For the programs where the strength does not vary from one criterion to another, we can observe that it is given by the lack of input data, so the selection of test cases depend merely on the number of processes. Since we do not change the number of processes for a test session, we execute one test case with a defined number of processes, and thus the coverage for the rest of the criteria does not change.

There is a relation between the number of adequate test cases and the coverage that set is able to obtain for other criteria. For example, in the Search number program (Table 28) we can observe that the AESL, ASEL, ASUL and ASCUL criteria gets a higher coverage for the others, but a fewer for the AERL and ADRL, which have only one test case.

In the Philosopher's dining program (Table 29), we can observe that the adequate test cases for the criterion All-sync-events-loop achieve a higher percentage for other criteria. ASEL criterion has a more significant number of test cases to obtain the highest coverage possible, considering the non-executable required elements. This situation leaves the other criteria with better strength than the ASEL itself.

Table 20 – Strength for GCD (Three slaves)

Strength	All-events-r-loop	All-events-s-loop	All-sync-events-loop	All-defs-rcv-loop	All-s-uses-loop	All-s-c-uses-loop	All-s-p-uses-loop
tcs-all-event-r-loop	100.00%	-	-	100.00%	-	-	-
tcs-all-defs-rcv-loop	100.00%	-	-	100.00%	-	-	-

Table 21 – Strength for Global sum

	All-events-r-loop	All-events-s-loop	All-sync-events-loop	All-defs-rcv-loop	All-s-uses-loop	All-s-c-uses-loop	All-s-p-uses-loop
tcs-all-event-r-loop	30.00%	40.00%	10.00%	30.00%	1.25%	5.00%	-
tcs-all-event-s-loop	30.00%	40.00%	10.00%	30.00%	1.25%	5.00%	-
tcs-all-sync-events-loop	30.00%	40.00%	10.00%	30.00%	1.25%	5.00%	-
tcs-all-defs-rcv-loop	30.00%	40.00%	10.00%	30.00%	1.25%	5.00%	-
tcs-all-s-uses-loop	30.00%	40.00%	10.00%	30.00%	1.25%	5.00%	-
tcs-all-s-c-uses-loop	30.00%	40.00%	10.00%	30.00%	1.25%	5.00%	-

Table 22 – Strength for Sieve of Eratosthenes

	All-events-r-loop	All-events-s-loop	All-sync-events-loop	All-defs-rcv-loop	All-s-uses-loop	All-s-c-uses-loop	All-s-p-uses-loop
tcs-all-event-r-loop	25.00%	-	-	25.00%	-	-	-
tcs-all-defs-rcv-loop	25.00%	-	-	25.00%	-	-	-

Table 23 – Strength for Sieve of Eratosthenes (fixed processes)

	All-events-r-loop	All-events-s-loop	All-sync-events-loop	All-defs-rcv-loop	All-s-uses-loop	All-s-c-uses-loop	All-s-p-uses-loop
tcs-all-event-r-loop	100.00%	100.00%	100.00%	100.00%	-	-	-
tcs-all-event-s-loop	70.00%	100.00%	50.00%	70.00%	-	-	-
tcs-all-sync-events-loop	100.00%	100.00%	100.00%	100.00%	-	-	-
tcs-all-defs-recv-loop	100.00%	100.00%	100.00%	100.00%	-	-	-

Table 24 – Strength for Jacobi

	All-events-r-loop	All-events-s-loop	All-sync-events-loop	All-defs-rcv-loop	All-s-uses-loop	All-s-c-uses-loop	All-s-p-uses-loop
tcs-all-event-r-loop	77.78%	77.78%	12.96%	77.78%	4.17%	0.00%	0.00%
tcs-all-event-s-loop	77.78%	77.78%	12.96%	77.78%	4.17%	0.00%	0.00%
tcs-all-sync-events-loop	77.78%	77.78%	12.96%	77.78%	4.17%	0.00%	0.00%
tcs-all-defs-recv-loop	77.78%	77.78%	12.96%	77.78%	4.17%	0.00%	0.00%
tcs-all-s-uses-loop	77.78%	77.78%	12.96%	77.78%	6.19%	0.00%	0.00%
tcs-all-s-c-uses-loop	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
tcs-all-s-p-uses-loop	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

Table 25 – Strength for Matrix multiplication

	All-events-r-loop	All-events-s-loop	All-sync-events-loop	All-defs-rcv-loop	All-s-uses-loop	All-s-c-uses-loop	All-s-p-uses-loop
tcs-all-event-r-loop	25.00%	25.00%	1.79%	25.00%	0.00%	0.00%	-
tcs-all-event-s-loop	25.00%	25.00%	1.79%	25.00%	0.00%	0.00%	-
tcs-all-sync-events-loop	25.00%	25.00%	1.79%	25.00%	0.00%	0.00%	-
tcs-all-defs-recv-loop	25.00%	25.00%	1.79%	25.00%	0.00%	0.00%	-
tcs-all-s-uses-loop	25.00%	25.00%	1.79%	25.00%	0.00%	0.00%	-
tcs-all-s-c-uses-loop	25.00%	25.00%	1.79%	25.00%	0.00%	0.00%	-

Table 26 – Strength for Trapezoidal rule

	All-events-r-loop	All-events-s-loop	All-sync-events-loop	All-defs-rcv-loop	All-s-uses-loop	All-s-c-uses-loop	All-s-p-uses-loop
tcs-all-event-r-loop	33.33%	33.33%	4.27%	33.33%	-	-	-
tcs-all-event-s-loop	33.33%	33.33%	4.27%	33.33%	-	-	-
tcs-all-sync-events-loop	33.33%	33.33%	4.27%	33.33%	-	-	-
tcs-all-defs-recv-loop	33.33%	33.33%	4.27%	33.33%	-	-	-

Table 27 – Strength for Odd even sort

	All-events-r-loop	All-events-s-loop	All-sync-events-loop	All-defs-rcv-loop	All-s-uses-loop	All-s-c-uses-loop	All-s-p-uses-loop
tcs-all-event-r-loop	38.89%	41.67%	8.33%	38.89%	1.74%	0.00%	0.00%
tcs-all-event-s-loop	38.89%	41.67%	8.33%	38.89%	1.74%	0.00%	0.00%
tcs-all-sync-events-loop	38.89%	41.67%	8.33%	38.89%	1.74%	0.00%	0.00%
tcs-all-defs-rcv-loop	38.89%	41.67%	8.33%	38.89%	1.74%	0.00%	0.00%
tcs-all-s-uses-loop	38.89%	41.67%	8.33%	38.89%	1.74%	0.00%	0.00%
tcs-all-s-c-uses-loop	38.89%	41.67%	8.33%	38.89%	1.74%	0.00%	0.00%
tcs-all-s-p-uses-loop	38.89%	41.67%	8.33%	38.89%	1.74%	0.00%	0.00%

Table 28 – Strength for Search number

	All-events-r-loop	All-events-s-loop	All-sync-events-loop	All-defs-rcv-loop	All-s-uses-loop	All-s-c-uses-loop	All-s-p-uses-loop
tcs-all-event-r-loop	33.33%	29.17%	3.65%	33.33%	4.17%	8.33%	-
tcs-all-event-s-loop	33.33%	41.67%	5.21%	33.33%	33.33%	66.67%	-
tcs-all-sync-events-loop	33.33%	41.67%	5.21%	33.33%	33.33%	66.67%	-
tcs-all-defs-rcv-loop	33.33%	29.17%	3.65%	33.33%	4.17%	8.33%	-
tcs-all-s-uses-loop	33.33%	41.67%	5.21%	33.33%	33.33%	66.67%	-
tcs-all-s-c-uses-loop	33.33%	41.67%	5.21%	33.33%	33.33%	66.67%	-

Table 29 – Strength for Philosopher's dining

	All-events-r-loop	All-events-s-loop	All-sync-events-loop	All-defs-rcv-loop	All-s-uses-loop	All-s-c-uses-loop	All-s-p-uses-loop
tcs-all-event-r-loop	100.00%	68.63%	9.57%	95.24%	95.00%	75.00%	47.92%
tcs-all-event-s-loop	100.00%	75.49%	9.78%	100.00%	100.00%	75.00%	47.92%
tcs-all-sync-events-loop	100.00%	75.49%	10.33%	100.00%	100.00%	100.00%	75.00%
tcs-all-defs-rcv-loop	100.00%	68.63%	9.57%	95.24%	95.00%	75.00%	47.92%
tcs-all-s-uses-loop	100.00%	70.49%	9.78%	100.00%	100.00%	75.00%	47.92%
tcs-all-s-c-uses-loop	100.00%	73.53%	10.11%	100.00%	100.00%	100.00%	75.00%
tcs-all-s-p-uses-loop	100.00%	73.53%	10.33%	100.00%	100.00%	100.00%	75.00%

8.5 Hypotheses testing

We tested each one of the hypotheses with statistical methods, in order to accept or reject Null Hypothesis and provide a conclusion regarding the results of the experiments. We used the Action Stat plugin for Microsoft Office Excel to perform the selected statistical methods for each analysis.

8.5.1 Cost analysis

We consider each one of the measurements to determine the application cost of criteria to test the related hypothesis: number of required elements, number of adequate test cases and

number of non-executable required elements.

For the number of required elements, we have the results for each tested program from the selected benchmarks. Using the Shapiro-Wilk test, we verified that the data does not present a normal distribution evidenced by a p-value = 0, which is < 0.5 . Then, we selected a non-parametric method being it the Kruskal-Wallis test using multiple comparisons with Bonferroni. For this test the p-value = 0.249992, allowing to accept the null hypothesis (NH1). That is to say that there is no statistical difference between at least one pair of criteria considering the cost for the number of required elements, which is also illustrated in the table 30 of Kruskal-Wallis test, showing a single group for mean value grouping.

For the cost analysis considering the number of adequate test cases for the proposed criteria, the Shapiro-Wilk test for normal distribution evaluation had a p-value = 0, which is < 0.5 , which means it is also not normal. The data contemplated criteria that did not generate required elements for any program, and criteria that had no test cases covering any of the required elements. We used Kruskal-Wallis test using multiple comparisons with Bonferroni as well, obtaining a p-value = 0.008834482 rejecting the null hypothesis (NH1). In other words, there is statistical evidence that there exists at least a pair of criteria for which the number of adequate test cases is different, see Table 31 for the factors, mean value and grouping information.

Table 30 – Grouping table for number of required elements - Kruskal-Wallis

Factors	Mean value (Rank)	Groups
ASEL	51.09091	a
ASUL	44.22727	a
ASCUL	40.59091	a
AESL	37.13636	a
ADRL	36.81818	a
AERL	36.81818	a
ASPUL	26.31818	a

Table 31 – Grouping table for number of adequate test cases - Kruskal-Wallis

Factors	Mean value (Rank)	Groups
ASEL	46.72727	a
ADRL	46.68182	a
AERL	46.68182	a
AESL	44.90909	a
ASUL	38.31818	ab
ASCUL	32.27273	ab
ASPUL	17.40909	b

The results of the hypothesis tests for number of adequate test cases shows that ASPUL criterion presents a difference in the application cost compared to ASUL and ASCUL criteria, and the group of ASEL, ADRL, AERL and AESL criteria. Observing the results of the experiments execution in Table 17, ASPUL criterion has not generated required elements for most of the programs, and thus it does not have data for number of adequate test cases. ASUL and ASCUL criteria present a similar scenario.

Finally, for the number of non-executable required elements, the Shapiro-Wilk test for normal distribution evaluation had a p-value < 0.5 , also evidencing that it is not normal. The data set considered criteria that generated required elements at least for one program and with zero non-executable required elements as well (which means that all of the required elements were covered by a test case). The Kruskal-Wallis test using multiple comparisons with Bonferroni obtained a p-value = 0.000321733, rejecting the null hypothesis (NH1) and allowing us to conclude that there is statistical evidence that the number of non-executable required elements is different for at least one pair of criteria. See Table 32 for mean and grouping data for the number of non-required elements.

It is desirable to analyze the three measurements for cost as a whole, for example, lower cost values considering non-executable elements is beneficial if the number of test cases is low as well. This would show that a criterion is able to cover a good quantity (if not all of them) of required elements with an adequate set of test cases.

Table 32 – Grouping table for number of non-executable required elements - Kruskal-Wallis

Factors	Mean (Rank)	Groups
ASPUL	47.5	a
ASEL	42	a
ASUL	39.42857	a
ASCUL	35.92857	ab
AESL	25.22222	ab
ADRL	17.18182	b
AERL	17.18182	b

In such a manner, considering the results of the statistical evaluation and the results of the experimental execution, the All-sync-events-loop (ASEL) and All-s-uses-loop (ASUL) criteria have the highest cost in terms of number of generated required elements, but present statistical difference in the number of test cases. This is explained in the lack of test cases for ASUL criterion due to the non generation of required elements for some programs, which is not the case of ASEL. However, ASUL and ASCUL (All-s-c-uses-loop) criteria have statistical similarity in the number of test cases as well as All-s-p-uses-loop (ASPUL), as a result of the generation of no required elements for some programs. This situation is discussed in depth in Section 8.6.

8.5.2 Effectiveness analysis

The data set for effectiveness analysis consisted of the results for the repetitions of test case execution for faulty programs. We obtained different results for each program and each set of test cases for all criteria. There were different results due to the programs' non-determinism, which provided data to statistically test the effectiveness of criteria to reveal errors.

Table 33 – Grouping table for number of criteria effectiveness - Kruskal-Wallis

Factors	Mean value (Rank)	Groups
ASEL	197.1019	a
AESL	186.9815	ab
AERL	185	ab
ADRL	182.8154	ab
ASUL	172.5729	ab
ASPUL	151.375	ab
ASCUL	139.7188	b

The Shapiro-Wilk test showed a p-value = 0, meaning it is not normal. We used the non-parametric test of Kruskal-Wallis, since the test to verify normal distribution of data with Shapiro-Wilk failed. The p-value = 0.021155904, rejecting the null hypothesis (NH2) and accepting the alternative hypothesis (AH2). In other words, there is statistical evidence showing that there is at least a criterion that has differences in the effectiveness to reveal defects, which we could infer as that there is a criterion that is more effective. Table 33 presents the grouping results for criteria. Kruskal-Wallis test showed that criteria could be arranged in different groups, considering the effectiveness percentage obtained for each faulty-program execution.

8.5.3 Strength analysis

For the analysis of the third hypotheses related to the strength of criteria, we used a multivariate analysis method, specifically the cluster analysis. The objective of this test is to identify possible relationships among the criteria to expose any inclusion relation, which makes it suitable for strength analysis. Dendograms are figures that represent inclusion relations among criteria, taking the mean value of the data set. We used the mean value of the coverage obtained by criteria with the set of adequate test cases of the rest of criteria. Each level in the dendogram represents a cluster with similar mean values. When a criterion includes another one, both should be seen at the same last level, meaning that there is no statistical difference in their coverage.

Figure 21 shows the dendogram for the adequate set of test cases for criterion AERL (All-events-r-loop), where we can observe an indication of inclusion relation between AERL and ADRL criteria and between ASEL and ASUL criteria. Such mean values similarities in the same group can be influenced by the fact that ASEL criterion considers s-edges (synchronization

Figure 21 – Dendrogram for the strength of AERL test case set

edges), while ASUL contemplates the association definition-s-use, which has a representation of an s-edge.

For ADRL criterion test cases set, the resulting dendrogram is equivalent to the AERL strength evaluation and for this reason it is not shown. We can find in all of the dendrograms for every criteria an inclusion relation between AERL and ADRL, belonging to the same cluster. Such result might be caused by the similarity in the required elements of both criteria, however, AERL is a control flow criterion and ADRL analyzes data flow.

Figure 22 presents dendrogram for AESL (All-events-s-loop), where AERL and ADRL are grouped in the same cluster, ASEL and ASPUL in another group and ASUL and ASCUL also in a different cluster. We observe a relation inclusion between criteria coinciding in a cluster.

Figure 23 shows the dendrogram for ASEL (All-sync-events-loop) test set, ASUL and ASPUL criteria present a similarity in the mean value of the obtained coverage, when applying the adequate set of test cases for ASEL criterion.

Figure 24 for ASUL criterion presents a small difference with AESL dendrogram in the criteria clusters, where the group for ASUL and ASCUL is more differentiated from ASEL and ASPUL cluster.

In dendrogram for ASCUL criterion (Figure 25), ASUL and ASPUL show a indication of inclusion, differently from the ASUL criterion dendrogram. We noted that this variation is given by the lack of required elements generated for some programs for the ASCUL and ASPUL criteria.

Figure 22 – Dendrogram for the strength of AESL test case set

Dendrogram for ASPUL criterion in Figure 26 also shows an inclusion relation in the coverage between ASUL and ASCUL criteria.

Given the results of clustering analysis with dendrograms and the previously exposed criteria test set inclusion, we can reject the Null hypothesis NH3 and accept Alternative hypothesis

Figure 23 – Dendrogram for the strength of ASEL test case set

Figure 24 – Dendrogram for the strength of ASUL test case set

AH3, as there are inclusion relations between at least one pair of criteria.

Figure 25 – Dendrogram for the strength of ASCUL test case set

Figure 26 – Dendrogram for the strength of ASPUL test case set

8.6 Discussion of results

The cost results show that when having nested loops, more processes and more communication primitives inside loops, there will be greater quantity of required elements generated. However, for the *All-s-c-uses-loop* and *All-s-p-uses-loop* criteria there are more non-executable required elements compared to the other criteria, which might increase the testing cost time. However, ASCUL and ASPUL criteria help to reveal specific types of defects, such as observability issues and inter-process data races. Basing on the results of application cost with programs having different numbers of processes, we infer that the cost will increase when considering a greater number of processes, which will be impacted by the depth of the loop and quantity of communication primitives.

With the test environment that we use for the experiments there are no computational resources limitations in the execution of test cases. In case of having a greater quantity of required elements with larger programs, it would be applicable as a complementary work to evaluate if more significant computational resources are required.

The proposed criteria show a considerable percentage of effectiveness to reveal non-determinism and communication related errors. With the test cases that we select based on criteria, we verify that we obtain a hundred percent of coverage for all criteria when considering executable required elements, and that they are events that corresponded to different loop iterations, rather than the execution of only the first iteration. We are able to demonstrate that there can be concurrency defects in loop iterations, and our criteria helps to reveal them.

We disregard the obtained coverage percentage for effectiveness evaluation, as in some programs a deadlock error is shown but the associations of the required elements are not able to complete the execution of paths in the trace files. In such case, we consider the criterion as effective since the test cases are selected based on our criteria, and with those test cases we achieve to reveal defects, even when program execution is not finished due to dead locks. We also observe in the outputs of the programs that the results of the re-execution of test cases for 30 times do not differ with the results of the first 15 repetitions.

Regarding the strength of criteria, we analyze the coverage results for each program and we observe that the coverage obtained by All-events-r-loop and All-defs-recv-loop criteria is similar, since their required elements consider the execution of receives in a loop. The programs with no input data do not allow us to analyze deeply the strength among criteria, however, in those cases the All-events-r-loop, All-events-s-loop and All-defs-recv-loop criteria obtained the higher coverage for the execution of the test cases. This situation is explained in the fact that the other criteria requires an association with the other process in a communication event. For the rest of the programs we observe that the All-sync-events-loop and All-s-c-uses-loop criteria obtain the higher coverage percentage for the other criteria, applying their adequate set of test cases.

For strength hypotheses statistical evaluation, we consider the coverage of criteria that generate required elements for a determined program. It does not involve the capacity of criteria to generate required elements for a program, it is related to the program's structure. However, we contemplate criteria that obtained no coverage because of non-executable required elements, which implicates an analysis of the executable events provided by the criteria.

It is pertinent to perform experiments for a different number of processes for each program and a larger sample of programs and defects, to analyze in depth the outcomes of the benchmarks that do not have input data to be executed. It is also relevant to evaluate how the cost of the criteria increases.

The experiments presented in this section show that the proposed criteria are relevant to reveal concurrent defects related to synchronization events and non-determinism, as we were able to obtain positive conclusions and deductions that support the validity of our test model. The benchmarks that we used for the experiments are relevant in terms of complexity and non-deterministic behavior (see Table 12), our focus in this project. Additionally, the number of required elements for the programs is considerable, showing that there are communication events (see Table 16) that should be contemplated when testing concurrent programs with interactions inside loops.

We accomplished our objective of verifying the applicability of the proposed criteria with this empirical study. We presented the advantages of the structural testing criteria in a practical environment, considering the cost, effectiveness, and strength. The results allow us to analyze the implications and benefits of applying the proposed criteria, for testing concurrent programs with

a message-passing memory paradigm when there is a presence of loops. The communication events that occur inside repetition structures present non-deterministic behavior, hiding defects that could not be identified with criteria that consider only going over one path of the loop.

8.7 Final considerations

In this chapter we analyzed and validated the application of the proposed structural testing criteria, in respect of cost, effectiveness and strength. First, we presented the process for selection of the tested programs in ValiMPI tool, along with their main features and significance for our experimental study.

The experimental design and execution allowed us to obtain the necessary data to perform statistical tests, and also to analyze the results of benchmarks execution in ValiMPI. We found that our criteria do generate required elements that help testers to include test cases that cover communication events in loops, effectively finding specific types of concurrency defects. Despite the application cost of our proposed criteria is increased due to the k iterations of loops and nested loops, whenever there are non-executable required elements they can be manually detected (i.e. the different iterations of a required element in a loop).

CONCLUSION

9.1 Research characterization

The main objective and contribution of this project is the proposition of structural testing criteria, contemplating loop execution in message-passing programs.

In order to determine the generation of required elements established by our criteria, we design and define criteria in a test model. We contemplate concurrent program elements that are related to non-determinism, communication and synchronization events that occur when executing repetition structures. In such a manner, we focus the construction of criteria considering the analysis of four main factors:

- i the types of loops and the paths traversing them,
- ii the concurrency types of defects,
- iii the number of iterations of a loop in a determined process, and
- iv the presence of nested loop and their impact on the generation of required elements.

An additional side-contribution of the modeling of our testing criteria is the proposal of a Concurrency Defects Taxonomy (Section 5.3), describing the main issues in shared and distributed memory programs.

Before validating the application of our criteria through an experimental evaluation, we implemented them in ValiMPI tool, enhancing the necessary modules to accomplish the test activity. We expose the details in Section 7.2. Our objective through the development of criteria is to preserve and improve when possible the modularity, usability, maintainability, reusability and functionality of the tool.

Our experimental evaluation allows us to verify the application of criteria, as well as the viability of their use in message-passing programs. The results show that our proposed criteria

reveals concurrency errors, accomplishing the objectives of this project. We demonstrate that there can be non-determinism related defects in loop iterations, and our criteria helps to reveal them.

9.2 Contributions

We describe the results obtained from the development of this project and the corresponding contributions to the area. We identify, consolidate and classify concurrency defects for shared and distributed memory programs, presenting a Concurrent Defects Taxonomy. Our classification serves as a basis for future studies that focus on the analysis of concurrency defects to define criteria, develop testing tools or perform empirical studies. The categories that we define are based on the main characteristics of the defects previously identified in other studies, we also consider the behavior of concurrent programs in the presence of such defects. We merge several taxonomies in the found literature with a complementary perspective, avoiding redundancy in each category.

The main contribution of this project is related to the definition of structural testing criteria of message-passing programs considering the execution of loops. We propose a test model establishing new sets of nodes, synchronization edges and loop nodes, along with loop def-use associations. The conceptualization behind our criteria constitutes a structured methodology to generate required elements, facilitating the testing activity and favoring tests automation. Our structural testing loop criteria focus on the revelation of defects related to communication events occurring in different loop iterations, which are generally difficult to identify. Thus, we contribute to the improvement of the quality and reliability of message-passing programs.

We implemented the proposed criteria in ValiMPI tool, for the automated generation of required elements and coverage evaluation. By means of this implementation we allow the evaluation of the proposed criteria through the execution of experimental studies. We also contribute to the enhancement of ValiMPI, which is the base for several investigations in our research group. We achieve to maintain the functionality of the previous criteria intact, incorporating proposed criteria conforming the test model definitions. This work describes in a detailed form the previous and actual behavior of ValiMPI tool, and we make available its source code with proper documentation, which can be used as a guide for future implementations.

We design and execute an empirical study to validate the set of proposed criteria, evaluating and comparing their application cost, effectiveness and strength. We demonstrate that our criteria is effective to reveal synchronization, communication and resource management defects, with an application cost that depends on the number of processes used for the program instrumentation. The identification of non-executable required elements can be of low-cost since replicated required elements can be marked as infeasible. Through the results of the execution of the experimental study, we prove that criteria considering the execution of communication

events in loops helps to identify concurrency defects that cannot be discovered by applying previous criteria. We present results on the comparison of application cost, effectiveness to reveal concurrency defects and inclusion relations among criteria, establishing a foundation for future experimental replications. It also supports studies that wish to have a guide for designing and executing empirical studies to validate structural testing criteria. Statistical analyses provide accuracy to our conclusions that are based on the results and hypotheses validation.

We intend in this work to illustrate and describe concepts, methods, tools and proceedings in an appropriate level of detail that allows future researchers to guide their investigations.

9.3 Challenges and limitations

One of the challenges we found in the development of this project is related to the identification of concurrency types of defects that our criteria address. Before its definition, we analyzed defects in existing taxonomies for message-passing programs. However, we did not find a unified classification, making difficult the selection and analysis of such defects in C/MPI programs (as we were evaluating them in ValiMPI). For this reason we proposed a consolidated Concurrent Defects Taxonomy.

The development in ValiMPI modules also implicated a challenge, as we required to do a deep analysis to have an understanding of the previous behavior before implementing our criteria. We approached the analysis with a reverse engineering method, along with the information provided in the papers that present ValiMPI tool. Accordingly, we completed a comprehensive code documentation.

Another challenge we overcome is in the selection of statistical methods to test the hypotheses, as it is not our speciality, we reached to previous studies and researchers that had performed empirical studies for testing criteria evaluation. The orientation in statistical methods allowed us to accomplish the experimental study objectives.

One of the limitations in the implementation of criteria is the automated recognition of nodes inside loops, due to constraints of ValiInst module that should be addressed to comply such requirement. We employ a solution that achieves the objective of this project, as our focus is criteria definition.

9.4 Future work

The contributions of this project provide the possibility of continuing with research involving structural testing criteria for loop execution of concurrent programs. We identify and leave open future work that can be based on our work:

- Extension of the ValiInst module: the implementation of an algorithm after program

instrumentation to retrieve the nodes in loops from the instrumented source code file and the DOT files. It would allow the automated identification of loops and nested loops, with an algorithm that analyzes the GFC files to identify cyclic paths, and using the instrumented file to obtain the node number from each loop.

- Structural testing criteria for shared memory programs: extending the test model proposed in (SARMANHO *et al.*, 2008) considering the execution of loops. The program representation and paradigm differs from distributed memory programs, so distinct analysis and developments are required to propose such criteria.
- Structural testing criteria considering recursive functions and collective communication primitives: recursivity in concurrent programs is also a gap in structural testing. It would require an analysis of the paths structures in calling such functions, the types of defects and the depth of recursion. Our project set a base for a study contemplating recursivity.

Another extension is related to the definition of loop-based criteria for collective communication primitives, where there are more than one process involved and thus the number of interactions increases. Following our research line using ValiMPI, it is also possible to implement criteria for collective MPI functions.

- Comparison of cost, effectiveness, and strength of the proposed criteria in contrast with the existing structural testing criteria for concurrent programs: it would allow comparing both sets of criteria and conclude how they can be applied and in which cases, in order to improve the quality and reliability of concurrent programs. Additionally, it is proper to evaluate if when increasing the number of required elements the size of the set of adequate test cases also increases, which depends on the synchronization events relation among processes.
- The scalability of the criteria considering real applications: using bigger programs and other testing and development environments. It would also help to test the scalability of the ValiMPI tool and its use of computational resources.
- Replications of the experimental study: in Software Experimental Engineering the reproduction of empirical studies is important to provide more reliability to their results and conclusions. Thus, it is important to contribute with the required information for replications. Experiments following the same or different design, for example, using a different set of benchmarks, injected types of defects, and testing environment.

BIBLIOGRAPHY

ADHIANTO, L.; BANERJEE, S.; FAGAN, M.; KRENTTEL, M.; MARIN, G.; MELLOR-CRUMMEY, J.; TALLENT, N. R. HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. **Concurrency Computation Practice and Experience**, v. 22, n. 6, p. 685–701, 2010. ISSN 15320626. Citation on page 58.

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. [s.n.], 2008. 21–22 p. ISBN 0521880386, 9780521880381. Available: <http://books.google.com/books?id=leokXF8pLY0C{&}pgis=1>. Citations on pages 29, 30, 31, and 32.

ARORA, V.; BHATIA, R.; SINGH, M. A systematic review of approaches for testing concurrent programs. **Concurr. Comput. : Pract. Exper.**, John Wiley and Sons Ltd., Chichester, UK, v. 28, n. 5, p. 1572–1611, Apr. 2016. ISSN 1532-0626. Available: <http://dx.doi.org/10.1002/cpe.3711>. Citations on pages 34, 66, and 67.

ARTHO, C.; BARRINGER, H.; GOLDBERG, A.; HAVELUND, K.; KHURSHID, S.; LOWRY, M.; PASAREANU, C.; ROŞU, G.; SEN, K.; VISSER, W.; WASHINGTON, R. Combining test case generation and runtime verification. **Theoretical Computer Science**, v. 336, n. 2-3, p. 209–234, 2005. ISSN 0304-3975. Citation on page 59.

BECHINI, A.; TAI, K.-C. Design of a toolset for dynamic analysis of concurrent Java programs. **Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)**, p. 190–197, 1998. ISSN 1092-8138. Available: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=693354>. Citations on pages 37, 55, and 58.

BRADBURY, J. **Using Program Mutation for the Empirical Assessment of Fault Detection Techniques: A Comparison of Concurrency Testing and Model Checking**. Phd Thesis (PhD Thesis), 01 2007. Citations on pages 61, 62, 67, and 69.

BREVEGLIERI, L.; REGHIZZI, S. C.; GOLDWURM, M. Efficient recognition of trace languages defined by repeat-until loops. **Inf. Comput.**, Academic Press, Inc., Duluth, MN, USA, v. 208, n. 8, p. 969–981, Aug. 2010. ISSN 0890-5401. Available: <http://dx.doi.org/10.1016/j.jc.2010.03.001>. Citations on pages 83 and 84.

BRITO, M.; FELIZARDO, K. R.; SOUZA, P.; SOUZA, S. Concurrent Software Testing: A Systematic Review. **Proc. 22nd IFIP International Conference on Testing Software and Systems**, p. 79–84, 2010. Citations on pages 35, 36, and 61.

BRITO, M. A. S. Avaliação da efetividade dos critérios de teste estruturais no contexto de programas concorrentes [dissertação]. **Master's thesis, Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo**, São Carlos, SP. Brazil, 2011. Citations on pages 69 and 101.

BRITO, M. A. S.; SOUZA, S. R. S.; SOUZA, P. S. L. An empirical evaluation of the cost and effectiveness of structural testing criteria for concurrent programs. **Procedia Computer Science**, Elsevier B.V., v. 18, p. 250–259, 2013. ISSN 18770509. Available: <http://dx.doi.org/10.1016/j.procs.2013.05.188>. Citations on pages 50 and 51.

BRUEGGE, B.; GOTTSCHALK, T.; LUO, B. A framework for dynamic program analyzers. In: **Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications**. New York, NY, USA: ACM, 1993. (OOPSLA '93), p. 65–82. ISBN 0-89791-587-9. Available: <http://doi.acm.org/10.1145/165854.165870>. Citation on page 59.

CARVER, R. H. Mutation-based testing of concurrent programs. In: **Proceedings of the IEEE International Test Conference on Designing, Testing, and Diagnostics - Join Them**. Washington, DC, USA: IEEE Computer Society, 1993. p. 845–853. ISBN 0-7803-1430-1. Available: <http://dl.acm.org/citation.cfm?id=648015.746077>. Citation on page 103.

CHEBARO, O.; CUOQ, P.; KOSMATOV, N.; MARRE, B.; PACALET, A.; WILLIAMS, N.; YAKOBOWSKI, B. Behind the scenes in SANTE: A combination of static and dynamic analyses. **Automated Software Engineering**, v. 21, n. 1, p. 107–143, mar 2014. ISSN 0928-8910. Available: <http://link.springer.com/10.1007/s10515-013-0127-xhttps://www.scopus.com/inward/record.uri?eid=2-s2.0-84898930835{&}partnerID=40{&}md5=d4c712976677827d9eea669ecbf726e2>. Citations on pages 58 and 59.

CHEN, J.; MACDONALD, S. Towards a better collaboration of static and dynamic analyses for testing concurrent programs. **Proc. of the 6th workshop on Parallel and distributed systems (PADTAD '08)**, p. 1–9, 2008. Citation on page 55.

CHEN, Q.; WANG, L.; YANG, Z.; STOLLER, S. D. HAVE: Integrated Dynamic and Static Analysis for Atomicity Violations. **12th International Conference on Fundamental Approaches to Software Engineering (FASE)**, 2009. Citations on pages 59, 60, and 73.

CHOI, S.-E.; LEWIS, E. C. A study of common pitfalls in simple multi-threaded programs. In: **Proceedings of the Thirty-first SIGCSE Technical Symposium on Computer Science Education**. New York, NY, USA: ACM, 2000. (SIGCSE '00), p. 325–329. ISBN 1-58113-213-1. Available: <http://doi.acm.org/10.1145/330908.331879>. Citations on pages 63, 67, and 69.

De Souza, S.; VERGILIO, S. R.; De Souza, P. S. L.; Da Silva Simão, A.; GONÇALVES, T. B.; De Melo Lima, A.; HAUSEN, A. C. ValiPar: A testing tool for message-passing parallel programs. **17th International Conference on Software Engineering and Knowledge Engineering, SEKE 2005**, n. November 2015, p. 386–391, 2005. Available: <http://www.scopus.com/inward/record.uri?eid=2-s2.0-84886872501{&}partnerID=40{&}md5=d6144d8f032511a504e84f37fe5a4fb4>. Citations on pages 44 and 45.

DELAMARO, M.; MALDONADO, J.; JINO, M. **Introdução ao teste de software**. CAM-PUS - RJ, 2007. ISBN 9788535226348. Available: <https://books.google.com.br/books?id=7R6XPgAACAAJ>. Citations on pages 24, 29, 30, 31, 32, 33, 35, 36, 58, 59, 60, and 105.

DESOUZA, J.; KUHN, B.; SUPINSKI, B. R. de; SAMOFALOV, V.; ZHELTOV, S.; BRATANOV, S. Automated, scalable debugging of mpi programs with intel® message checker. In: **Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications**. New York, NY, USA: ACM, 2005. (SE-HPCS '05), p. 78–82. ISBN 1-59593-117-1. Available: <http://doi.acm.org/10.1145/1145319.1145342>. Citations on pages 59, 60, 64, 67, and 69.

DIAZ, S. M. D.; SOUZA, P. S. L. de. Systematic mapping on structural testing criteria for concurrent programs considering dynamic aspects. In: **ERAD SP 2017**. Sao Carlos, SP, Brasil: SBC, 2017. Available: <http://portaldeconteudo.sbc.org.br/index.php/fpgeradsp/article/view/3085>. Citations on pages 25 and 66.

DOURADO, G. G.; SOUZA, P. S.; PRADO, R. R.; BATISTA, R. N.; SOUZA, S. R.; ESTRELLA, J. C.; BRUSCHI, S. M.; LOURENCO, J. A suite of java message-passing benchmarks to support the validation of testing models, criteria and tools. **Procedia Computer Science**, v. 80, p. 2226 – 2230, 2016. ISSN 1877-0509. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA. Available: <<http://www.sciencedirect.com/science/article/pii/S1877050916308638>>. Citations on pages 40 and 41.

DOURADO, G. G. M. Contribuindo para a Avaliação do Teste de Programas Concorrentes : uma abordagem usando benchmarks. **Master's thesis, Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo**, São Carlos, SP. Brazil, 2015. Citations on pages 24, 40, 41, 97, and 100.

ENDO, A. T.; SIMÃO, A. d. S.; SOUZA, S. d. R. S. d.; SOUZA, P. S. L. d. Web services composition testing: A strategy based on structural testing of parallel programs. In: **Testing: Academic Industrial Conference - Practice and Research Techniques (taic part 2008)**. [S.l.: s.n.], 2008. p. 3–12. Citation on page 45.

Farchi, E.; Nir, Y.; Ur, S. Concurrent bug patterns and how to test them. In: **Proceedings International Parallel and Distributed Processing Symposium**. [S.l.: s.n.], 2003. p. 7 pp.–. ISSN 1530-2075. Citations on pages 63 and 67.

FLANAGAN, C.; FREUND, S. N.; YI, J. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 43, n. 6, p. 293–303, 2008. ISSN 0362-1340. Available: <<http://doi.acm.org/10.1145/1379022.1375618>>. Citation on page 59.

GONG, X.; WANG, Y.; ZHOU, Y.; LI, B. On Testing Multi-threaded Java Programs. In: **Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing. SNPD 2007**. [S.l.: s.n.], 2007. v. 1, p. 702–706. Citations on pages 55 and 58.

HAUSEN, A. C. Valimpi : uma ferramenta de teste estrutural para programas paralelos em ambiente de passagem de mensagem. **Master's thesis, Universidade Federal do Paraná**, São Carlos, SP. Brazil, p. 92, 2005. Citations on pages 44, 45, 47, and 84.

HAUSEN, A. C.; VERGILIO, S.; SOUZA, S. A tool for structural testing of mpi programs. 02 2019. Citation on page 46.

HOWDEN, W. E. Methodology for the generation of program test data. **IEEE Trans. Comput.**, IEEE Computer Society, Washington, DC, USA, v. 24, n. 5, p. 554–560, May 1975. ISSN 0018-9340. Available: <<http://dx.doi.org/10.1109/T-C.1975.224259>>. Citation on page 72.

HWANG, G.-H.; LIN, H.-Y.; LIN, S.-Y.; LIN, C.-S. Statement-coverage testing for concurrent programs in reachability testing. **Journal of Information Science and Engineering**, v. 30, n. 4, p. 1095–1113, 2014. Available: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-84940243977{&}partnerID=40{&}md5=37fed2492197ee1fd70fc1c4ac88231a>>. Citations on pages 55 and 58.

KOJIMA, H.; TAKAHASHI, J.; OHTA, T.; KAKUDA, Y. A model for concurrent states and its coverage criteria. In: **2009 International Symposium on Autonomous Decentralized Systems**. [S.l.: s.n.], 2009. p. 1–6. ISSN 1541-0056. Citations on pages 56, 57, and 58.

KOSMATOV, N.; WILLIAMS, N.; BOTELLA, B.; ROGER, M.; CHEBARO, O. A lesson on structural testing with PathCrawler-online.com. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, v. 7305 LNCS, p. 169–175, 2012. Citation on page 59.

KRAWCZYK, H.; WISZNIEWSKI, B.; MORK, P. Classification of software defects in parallel programs. **Control**, p. 89–111. Citations on pages 36, 65, 66, and 67.

LETKO, Z. Sophisticated Testing of Concurrent Programs. In: **Search Based Software Engineering (SSBSE), 2010 Second International Symposium on**. [S.l.: s.n.], 2010. p. 36–39. Citation on page 59.

LI, B. H.; SHIEH, S. RELEASE: Generating Exploits Using Loop-Aware Concolic Execution. In: **Fifth International Conference on Secure Software Integration and Reliability Improvement (SSIRI)**. [S.l.: s.n.], 2011. p. 165–173. Citations on pages 54 and 58.

LONG, B.; STROOPER, P.; WILDMAN, L. A method for verifying concurrent Java components based on an analysis of concurrency failures. **Concurrency Computation Practice and Experience**, v. 19, n. 3, p. 281–294, 2007. Available: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-34548726308&partnerID=40&md5=84cb54728d095406f813b84b08d8c2e1>>. Citation on page 59.

LOURENÇO, J.; CUNHA, J. C.; KRAWCZYK, H.; LOURENCO, J.; CUNHA, J. C.; KRAWCZYK, H.; KUZORA, P.; NEYMAN, M.; WISZNIEWSKI, B. An integrated testing and debugging environment for parallel and distributed programs. In: **EUROMICRO 97. New Frontiers of Information Technology., Proceedings of the 23rd EUROMICRO Conference**. [s.n.], 1997. p. 291–298. ISSN 1089-6503. Available: <http://ieeexplore.ieee.org/xpls/abs/_all.jsp?arnumber=617289>. Citations on pages 58, 59, 60, and 67.

LOURENÇO, J. a.; CUNHA, G. Testing patterns for software transactional memory engines. In: **Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging**. New York, NY, USA: ACM, 2007. (PADTAD '07), p. 36–42. ISBN 978-1-59593-748-3. Available: <<http://doi.acm.org/10.1145/1273647.1273655>>. Citation on page 62.

LU, S.; JIANG, W.; ZHOU, Y. A study of interleaving coverage criteria. In: **The 6th Joint Meeting on European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers**. New York, NY, USA: ACM, 2007. (ESEC-FSE companion '07), p. 533–536. ISBN 978-1-59593-812-1. Available: <<http://doi.acm.org/10.1145/1295014.1295034>>. Citations on pages 57 and 58.

LU, S.; PARK, S.; SEO, E.; ZHOU, Y. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In: . [S.l.: s.n.], 2008. v. 36, p. 329–339. Citations on pages 62, 63, 67, and 69.

MACHADO, M. C. C. Estudo e definição de mecanismos para redução do custo de aplicação do teste de programas concorrentes. **Master's thesis, Universidade de São Paulo**, São Carlos, SP, Brazil, p. 92, 2011. Citation on page 46.

MALDONADO, J. C. **Crítérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software**. Phd Thesis (PhD Thesis) — DCA/FEEC/UNICAMP, Campinas, SP, Jul. 1991. Citation on page 33.

MARCOZZI, M.; DELAHAYE, M.; BARDIN, S.; KOSMATOV, N.; PREVOSTO, V. Generic and effective specification of structural test objectives. In: **2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)**. [S.l.: s.n.], 2017. Citations on pages 56 and 58.

MELO, S. M.; SOUZA, S. d. R. S. de; SARMANHO, F. S.; SOUZA, P. S. L. d. Contributions for the structural testing of multithreaded programs: coverage criteria, testing tool, and experimental evaluation. **Software Quality Journal**, Jul 2017. ISSN 1573-1367. Available: <<https://doi.org/10.1007/s11219-017-9376-4>>. Citations on pages 51, 59, 69, and 101.

OLIVEIRA, A.; SOUZA, P. Lopes de; SOUZA, S. Valierlang: A structural testing tool for erlang programs. In: . [S.l.: s.n.], 2016. p. 1–10. Citation on page 46.

PACHECO, P. **An Introduction to Parallel Programming**. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 9780123742605. Citations on pages 23 and 34.

PEDERSEN, J. B. Classification of programming errors in parallel message passing systems. In: **Communicating Process Architectures - CPA**. [S.l.: s.n.], 2006. Citations on pages 65 and 67.

PERSON, S.; YANG, G.; RUNGTA, N.; KHURSHID, S. Directed incremental symbolic execution. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 46, n. 6, p. 504–515, Jun. 2011. ISSN 0362-1340. Available: <<http://doi.acm.org/10.1145/1993316.1993558>>. Citation on page 59.

POLURI, S. V.; RAMANATHAN, M. K. Deterministic dynamic race detection across program versions. In: **IEEE International Conference on Software Maintenance and Evolution (ICSME)**. [S.l.: s.n.], 2015. p. 181–190. Citation on page 59.

PRADO, R. R.; SOUZA, P. S. L.; DOURADO, G. G. M.; SOUZA, S. R. S.; ESTRELLA, J. C.; BRUSCHI, S. M. Extracting Static and Dynamic Structural Information from Java Concurrent Programs for Coverage Testing. 2015. Citations on pages 40, 44, 58, and 59.

PRADO, R. R.; SOUZA, P. S. L.; SOUZA, S. R. S.; DOURADO, G. G. M.; BATISTA, R. N. Valipar service: Structural testing of concurrent programs as a web service composition. In: LATIFI, S. (Ed.). **Information Technology: New Generations**. Cham: Springer International Publishing, 2016. p. 581–590. ISBN 978-3-319-32467-8. Citation on page 46.

RAMAMOORTHY, C. V.; HO, S. B. F.; CHEN, W. T. On the automated generation of program test data. **IEEE Transactions on Software Engineering**, SE-2, n. 4, p. 293–300, Dec 1976. ISSN 0098-5589. Citations on pages 72 and 73.

RAPPS, S.; WEYUKER, E. J. Data flow analysis techniques for test data selection. In: **Proceedings of the 6th International Conference on Software Engineering**. Los Alamitos, CA, USA: IEEE Computer Society Press, 1982. (ICSE '82), p. 272–278. Available: <<http://dl.acm.org/citation.cfm?id=800254.807769>>. Citations on pages 33, 43, and 77.

ROTHERMEL, G.; HARROLD, M. J.; ORSO, A. Representation and Analysis of Software. **Control**, p. 1–19, 2005. ISSN 1097024X. Available: <<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.122.5211&rep=rep1&type=pdf>>. Citations on pages 31 and 32.

SAHOO, B.; RAY, M. Concurrent defects and test coverage criteria. **Journal of Software**, v. 11, 10 2016. Citations on pages 66 and 67.

SANTHIAR, A.; KALEESWARAN, S.; KANADE, A. Efficient Race Detection in the Presence of Programmatic Event Loops. In: **Proceedings of the 25th International Symposium on Software Testing and Analysis**. New York, NY, USA: ACM, 2016. (ISSTA 2016), p. 366–376. ISBN 978-1-4503-4390-9. Available: <http://doi.acm.org/10.1145/2931037.2931068>. Citations on pages 58 and 59.

SARMANHO, F. S. Teste de programas concorrentes com memória compartilhada. **Master's thesis, Universidade de São Paulo**, São Carlos, SP, Brazil, p. 92, 2009. Citation on page 45.

SARMANHO, F. S.; SOUZA, P. S. L.; SOUZA, S. R. S.; SIMÃO, A. S. Structural testing for semaphore-based multithread programs. In: BUBAK, M.; ALBADA, G. D. van; DONGARRA, J.; SLOOT, P. M. A. (Ed.). **Computational Science – ICCS 2008**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 337–346. ISBN 978-3-540-69384-0. Citations on pages 37, 45, 51, and 130.

SCHIMMEL, J.; MOLITORISZ, K.; JANNESARI, A.; TICHY, W. F. Combining Unit Tests for Data Race Detection. In: **Proceedings of the 10th International Workshop on Automation of Software Test**. Florence, Italy: IEEE Press, 2015. (AST '15), p. 43–47. Available: <http://dl.acm.org/citation.cfm?id=2819261.2819275>. Citations on pages 56 and 58.

SIEGEL, S. F.; ZIRKEL, T. K. Loop invariant symbolic execution for parallel programs. In: **Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation**. Berlin, Heidelberg: Springer-Verlag, 2012. (VMCAI'12), p. 412–427. ISBN 978-3-642-27939-3. Available: http://dx.doi.org/10.1007/978-3-642-27940-9_27. Citations on pages 53, 54, 58, and 59.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. **Operating System Concepts**. 8th. ed. [S.l.]: Wiley Publishing, 2008. ISBN 0470128720. Citation on page 34.

SILVA, R. A. Teste de mutação aplicado a programas concorrentes em mpi. **Master's thesis, Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo**, São Carlos, SP, Brazil, 2013. Citations on pages 40 and 103.

SIMÃO, A.; MALDONADO, J.; VINCENZI, A.; SANTANA, A. A language for the description of program instrumentation and automatic generation of instrumenters. **CLEI Electron. J.**, v. 6, 01 2003. Citations on pages 44 and 46.

SOUZA, P. L.; SAWABE, E. T.; aO, A. S. S.; VERGILIO, S. R.; SOUZA, S. R. S. D. Valipvm - a graphical tool for structural testing of pvm programs. In: **Proceedings of the 15th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface**. Berlin, Heidelberg: Springer-Verlag, 2008. p. 257–264. ISBN 978-3-540-87474-4. Available: http://dx.doi.org/10.1007/978-3-540-87475-1_35. Citation on page 45.

SOUZA, P. S.; SOUZA, S. R.; ZALUSKA, E. Structural testing for message-passing concurrent programs: an extended test model. **Concurrency and Computation: Practice and Experience**, v. 26, n. 1, p. 21–50, 2014. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.2937>. Citations on pages 24, 25, 39, 41, 42, 43, 72, 74, 79, and 81.

SOUZA, P. S. L.; SOUZA, S. S.; ROCHA, M. G.; PRADO, R. R.; BATISTA, R. N. Data flow testing in concurrent programs with message passing and shared memory paradigms. **Procedia Computer Science**, Elsevier B.V., v. 18, p. 149–158, 2013. ISSN 18770509. Available: <http://dx.doi.org/10.1016/j.procs.2013.05.178>. Citations on pages 39, 40, 41, and 81.

SOUZA, S. c.; VERGILIO, R.; SOUZA, P.; SIMÃO, S.; HAUSEN, A. Structural testing criteria for message-passing parallel programs. **Concurrency Computation Practice and Experience**, v. 20, n. 16, p. 1893–1916, 2008. Available: <<https://www.scopus.com/inward/record.uri?eid=2-s2.0-55349094707{&}partnerID=40{&}md5=54d7cf3855ab501c10fb09909b61b2c8>>. Citations on pages 24, 25, 36, 37, 39, 40, 42, 43, 50, 51, 58, 59, 72, 79, 81, and 101.

SOUZA, S. R. S.; SOUZA, P. S. L.; BRITO, M. A. S.; SIMAO, A. S.; ZALUSKA, E. J. Empirical evaluation of a new composite approach to the coverage criteria and reachability testing of concurrent programs. **Software Testing, Verification and Reliability**, v. 25, n. 3, p. 310–332, 2015. ISSN 1099-1689. Stvr.1568. Available: <<http://dx.doi.org/10.1002/stvr.1568>>. Citation on page 51.

TAKAHASHI, J.; KOJIMA, H.; FURUKAWA, Z. Coverage Based Testing for Concurrent Software. In: **2008 The 28th International Conference on Distributed Computing Systems Workshops**. [S.l.: s.n.], 2008. p. 533–538. ISSN 1545-0678. Citations on pages 56 and 58.

TANENBAUM, A. S. **Modern Operating Systems**. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 9780136006633. Citations on pages 13, 34, and 35.

TERRAGNI, V.; CHEUNG, S.-C. Coverage-driven Test Code Generation for Concurrent Classes. In: **Proceedings of the 38th International Conference on Software Engineering**. New York, NY, USA: ACM, 2016. (ICSE '16), p. 1121–1132. ISBN 978-1-4503-3900-1. Available: <<http://doi.acm.org/10.1145/2884781.2884876>>. Citation on page 59.

VERGILIO, S. R.; MALDONADO, J. C.; JINO, M. Infeasible paths in the context of data flow based testing criteria: identification, classification and prediction. **Journal of the Brazilian Computer Society**, scielo, v. 12, p. 71 – 86, 06 2006. ISSN 0104-6500. Citations on pages 56 and 58.

VO, A.; AANANTHAKRISHNAN, S.; GOPALAKRISHNAN, G.; SUPINSKI, B. R. de; SCHULZ, M.; BRONEVETSKY, G. A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In: **Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis**. Washington, DC, USA: IEEE Computer Society, 2010. (SC '10), p. 1–10. ISBN 978-1-4244-7559-9. Available: <<http://dx.doi.org/10.1109/SC.2010.7>>. Citation on page 59.

WANG, Q.; HUANG, J.; GONG, Y. Research on loop path selection in coverage testing. In: **2013 Fourth World Congress on Software Engineering**. [S.l.: s.n.], 2013. p. 99–104. Citations on pages 53, 58, 83, and 84.

WILDMAN, L.; LONG, B.; STROOPER, P. Testing Java interrupts and timed waits. In: **Software Engineering Conference, 2004. 11th Asia-Pacific**. [S.l.: s.n.], 2004. p. 438–447. ISSN 1530-1362. Citations on pages 58 and 59.

WOHLIN, C.; RUNESON, P.; HST, M.; OHLSSON, M. C.; REGNELL, B.; WESSLN, A. **Experimentation in Software Engineering**. [S.l.]: Springer Publishing Company, Incorporated, 2012. ISBN 3642290434, 9783642290435. Citations on pages 25, 50, and 51.

WONG, W. E.; LEI, Y. Reachability Graph-Based Test Sequence Generation for Concurrent Programs. **International Journal of Software Engineering and Knowledge Engineering**, v. 18, n. 6, p. 803–822, 2008. ISSN 0218-1940. Citations on pages 55 and 58.

YANG, C.-s. D.; POLLOCK, L. L. Program-Based, Structural Testing of Shared Memory Parallel Programs. **Information Sciences**, 1999. Citations on pages 36, 37, 55, and 58.

YANG, C.-S. D.; SOUTER, A. L.; POLLOCK, L. L. All-du-path coverage for parallel programs. **ACM SIGSOFT Software Engineering Notes**, v. 23, n. 2, p. 153–162, 1998. ISSN 01635948. Citations on pages 37, 54, and 58.

YANG, R.-D.; CHUNG, C.-G. Path analysis testing of concurrent programs. **Information and Software Technology**, v. 34, n. 1, p. 43–56, 1992. Citation on page 32.

YOGA, A.; NAGARAKATTE, S. Atomicity Violation Checker for Task Parallel Programs. In: **Proceedings of the 2016 International Symposium on Code Generation and Optimization**. New York, NY, USA: ACM, 2016. (CGO 2016), p. 239–249. ISBN 978-1-4503-3778-6. Available: <<http://doi.acm.org/10.1145/2854038.2854063>>. Citations on pages 55, 56, and 58.

YU, T.; SRISA-AN, W.; ROTHERMEL, G. An automated framework to support testing for process-level race conditions. **Software Testing, Verification and Reliability**, v. 27, n. 4-5, p. e1634–n/a, 2017. ISSN 1099-1689. E1634 stvr.1634. Available: <<http://dx.doi.org/10.1002/stvr.1634>>. Citation on page 59.

ZANJANI, S. N.; FULADI, M. D. T.; AGHABABA, A. B. Improving path selection by handling loops in automatic test data generation. In: **2011 IEEE 14th International Multitopic Conference**. [S.l.: s.n.], 2011. p. 273–278. Citations on pages 72, 73, 83, and 84.

ZHANG, X.; YANG, Z.; ZHENG, Q.; LIU, P.; CHANG, J.; HAO, Y.; LIU, T. Automated testing of definition-use data flow for multithreaded programs. In: **2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)**. [S.l.: s.n.], 2017. p. 172–183. Citation on page 59.

ZHAO, Q.; SIM, J. E.; WONG, W.-F.; RUDOLPH, L. DEP: Detailed Execution Profile. In: **Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques**. New York, NY, USA: ACM, 2006. (PACT '06), p. 154–163. ISBN 1-59593-264-X. Available: <<http://doi.acm.org/10.1145/1152154.1152180>>. Citation on page 59.

BENCHMARKS DOCUMENTATION

This appendix presents a short documentation of the selected parallel programs, considering an overview description of the program, compilation, execution and input data (required for adequate program execution) and benchmark objective. Further information regarding the program features can be found in [Section 8](#).

GCD Two slaves:

- *Overview.* Calculates the GCD for three numbers x , y , z using one master process and two slaves. Each slave processes two numbers (slave one calculates GCD for x and y , and slave two for y and z), both sent by the master process. When both slaves return one as a result, the program prints the answer, otherwise, the master sends both results to slave one to calculate the final GCD. The slaves end their execution when master process sends zero (0).
- *Objective.* The objective of this benchmark is to verify if concurrent program tools are able to identify errors regarding synchronization and data races among processes, in small size message passing programs.
- *Compilation.* `mpicc -o gcd gcdtwoslaves.c`
- *Execution.* `mpirun -n 3 gcd <  $n_1$   $n_2$   $n_3$  >`
- *Input data.* The input of the program are three integer numbers, and expected result is the GCD.

GCD Three slaves:

- *Overview.* Calculates the GCD for three numbers x , y , z using one master process and three slaves. Slave one calculates GCD for x and y , and slave two for y and z . When both slaves

return one as a result, the program prints the answer and slave three finishes its execution; otherwise, the master sends both results to slave three one to calculate the final GCD.

- *Objective.* The objective of this benchmark is to verify if concurrent program tools are able to identify errors when a message is sent to the wrong process, in small size message passing programs.
- *Compilation.* `mpicc -o gcd gcdthreeslaves.c`
- *Execution.* `mpirun -n 4 gcd < n1 n2 n3 >`
- *Input data.* The input of the program are three integer numbers, and expected result is the GCD.

Global sum:

- *Overview.* Sums the values of the ranks of the processes in a tree-sum mode, by determining with an iteration factor which process is going to send or receive. The process with rank 0 prints the result.
- *Objective.* The objective of this benchmark is to verify if concurrent program tools are able to identify errors when the number of iterations depends on the number of processes and the buffer that is sent has a modification.
- *Compilation.* `mpicc globalsummpi.c -o globalsummpi -Wall`
- *Execution.* `mpirun -n < np > globalsummpi`
- *Input data.* The input of the program are the number of processes, and expected result is the sum of all the processes' ranks.

Sieve of Eratosthenes:

- *Overview.* Calculates the number of primes less than or equal to the input. The algorithm divides in blocks the array of the numbers, and then it marks the multiples of the i-th number; following the sequential algorithm in Michael Quinn.
- *Objective.* The objective of this benchmark is to verify the behavior of concurrent programs when there are modifications of a variable inside a loop, and also when switching arguments in a reduce primitive, in medium size message passing programs.
- *Compilation.* `mpicc -o sieve sieve.c -lm`
- *Execution.* `mpirun -n < np > sieve < input >`

- *Input data.* The input of the program is an integer number, and expected result is the number of primes less than or equal to that number.

Odd even sort:

- *Overview.* Orders a set of numbers; each process generates its own 10 values to order, and the master process (rank 0) prints the result of the merged array. The size of the array depends on the number of processes.
- *Objective.* The objective of this benchmark is to verify the behavior of concurrent programs when there are modifications of a variable inside a loop, and also when there is a interprocess data race, in medium size message passing programs.
- *Compilation.* `mpicc -o odd odd.c`
- *Execution.* `mpirun -n < np > odd`
- *Input data.* The input of the program is an integer number, and expected result is the number of primes less than or equal to that number.

Trapezoidal rule:

- *Overview.* Implements a parallel version of the trapezoidal rule. Each process calculates the interval of integration and estimates the integral of the function using the trapezoidal rule. Process 0 receives the integral estimate from the other processes and sums it to print the result. The function is hardwired.
- *Objective.* The objective of this benchmark is to verify the behavior of concurrent programs when the values of the buffer in a send primitive are changed inside a loop, in small size message passing programs.
- *Compilation.* `mpicc -o trap trap.c`
- *Execution.* `mpirun -n < np > ./trap`
- *Input data.* The input of the program is an integer number, and expected result is the number of primes less than or equal to that number.

Matrix multiplication:

- *Overview.* Implements a parallel version of a matrix multiplication. The master task distributes a matrix multiply operation to numtasks-1 worker tasks. The master process initializes the matrices.

- *Objective.* The objective of this benchmark is to verify the behavior of concurrent programs when a variable inside a loop is changed, in small size message passing programs.
- *Compilation.* `mpicc -o matrix matrix.c`
- *Execution.* `mpirun -np <np> matrix`
- *Input data.* There is not input for the program, and expected result is the number of primes less than or equal to that number.

Search number:

- *Overview.* The purpose of the algorithm is to find a number (input by the user) in an array of numbers. A master process generates the array and then distributes the information among a set of worker processes, and waits for them to communicate back the (global) index values at which occurrences of the target value were found.
- *Objective.* The objective of this benchmark is to verify the behavior of a concurrent program when a variable that controls the number of workers that are done with their execution is modified during a loop execution, in small size message passing programs.
- *Compilation.* `mpicc -o search search.c`
- *Execution.* `mpirun -n <np> search <targetvalue>`
- *Input data.* The input is the target value, and expected result is to print the process rank (or ranks) in which the value was found, and the its position in the whole array (all occurrences of the target value).

Jacobi MPI:

- *Overview.* Implements a parallel algorithm of the Jacobi iteration problem. Each Worker computes values in one strip, communicating with neighboring workers until the specified number of iterations. Each then sends its results to the Coordinator (master process) for printing.
- *Objective.* The objective of this benchmark is to verify the behavior of a concurrent program when there is a missing mutex present in a loop structure, in medium size message passing and shared memory programs with both collective and point to point communications.
- *Compilation.* `mpicc -o jacobi jacobi.c -lm`
- *Execution.* `mpirun -n <np> jacobi <gridsize> <numiterations>`

- *Input data.* The input is the target value, and expected result is to print the process rank (or ranks) in which the value was found, and the its position in the whole array (all occurrences of the target value).