A structural testing tool for MPI programs with loops

Silvia M. D. Diaz [1], **Paulo S. L. de Souza** [1], **Simone do R. S. de Souza** [1]

[1]Institute of Mathematical and Computational Sciences
University of Sao Paulo, Sao Carlos - SP, Brazil.

`smdiazdiaz@usp.br, {pssouza,srocio}@icmc.usp.br`

Resumo. *Há uma alta demanda por programas paralelos corretos, principalmente devido às arquiteturas paralelas atuais, como clusters e processadores multi/many cores. O teste estrutural permite identificar defeitos pela cobertura de estruturas internas de programas paralelos. O não determinismo em programas paralelos traz novos desafios ao teste estrutural. Ele requer ferramentas e modelos de teste específicos, capazes de cobrir primitivas de comunicação e sincronização com comportamentos dinâmicos, tais como os presentes em loops. Este artigo propõe uma nova ferramenta de software para o teste estrutural, com o objetivo de auxiliar testadores na revelação de defeitos desconhecidos associados a comunicação e presentes em estruturas de repetição de programas paralelos em C/MPI. Baseando-se na cobertura obtida, testadores podem escolher casos de teste específicos e avaliar o progresso da atividade de teste. A ferramenta de teste proposta é validada com a ingestão de defeitos no código de um programa, e com a análise do suporte dado pela ferramenta para a geração de elementos requeridos e seleção de casos de teste. A ferramenta proposta automatiza parte da atividade de teste, especificamente a geração de elementos requeridos e guia a execução dos testes, reduzindo o tempo para a aplicação da atividade de teste. Nossos resultados mostram que a ferramenta de teste é capaz de revelar defeitos desconhecidos em primitivas de comunicação presentes em iterações de loops.*

Abstract. *There is a growing demand for correct parallel programs, mainly due to nowadays parallel architectures, such as clusters and multi/many-core processors. Structural testing allows the identification of defects by covering internal structures of parallel programs. Nondeterminism in parallel programs brings new challenges to the structural testing. It requires specific test model and tools, capable to cover communication and synchronization primitives with dynamic behaviors, such as those present inside of loops. This paper proposes a novel software tool for the structural testing, aiming to help testers in revealing defects associated to communication present in repetition structures of C/MPI parallel programs. Based on the obtained coverage, testers can choose specific test cases and evaluate the progress of the testing activity. We validate the proposed testing software tool by injecting a defect in a program code, and analyzing the support for generation of required elements and selection of test cases. ValiMPI tool automates part of the test activity, specifically the generation of required elements to guide test case selection, reducing the application cost of the testing activity. Our results demonstrate that the testing tool is capable to reveal unknown defects from communication in different loop iterations.*

1. Introduction

Parallel programming is widely employed to develop applications requiring high performance, such as data analysis, climate modeling, energy research, bio-engineering processing, and other real-world problems. Due to the features of these applications, performance, reliability, and accuracy are highly required and, therefore, it is important ensuring the quality of them [Alghamdi and Eassa 2019].

Software testing criteria and tools have been adapted to the context of parallel and concurrent applications. During the testing activity, a challenge is to deal with the non-determinism present in these applications. Because of the non-determinism, the execution of a program with the same test input can generate different possible outputs, which must be tested in order to verify if the expected behavior is obtained.

Structural testing is one of the techniques for the validation and verification of concurrent programs [Delamaro et al. 2007, Ammann and Offutt 2008]. It allows revealing defects associated to concurrent events by analyzing the structure of the program, guiding the selection of test cases to cover required elements provided by testing criteria. Testing criteria establishes rules to select test data to cover required elements based on the control, data, communication and synchronization flows of concurrent programs. Analyzing loop paths in concurrent programs is imperative, since possible sub-paths can increase in number if not applying an adequate technique to select them. Another 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. The implementation of testing tools could diminish the risks and impact of threats caused by the insufficient or lack of software quality, preventing additional and unnecessary efforts in the software development process. On the other hand, the cost of test activity is high, as it requires the generation of test cases, execution of the program and analysis of the outcomes. Therefore, it is imperative to improve and automate the testing activity.

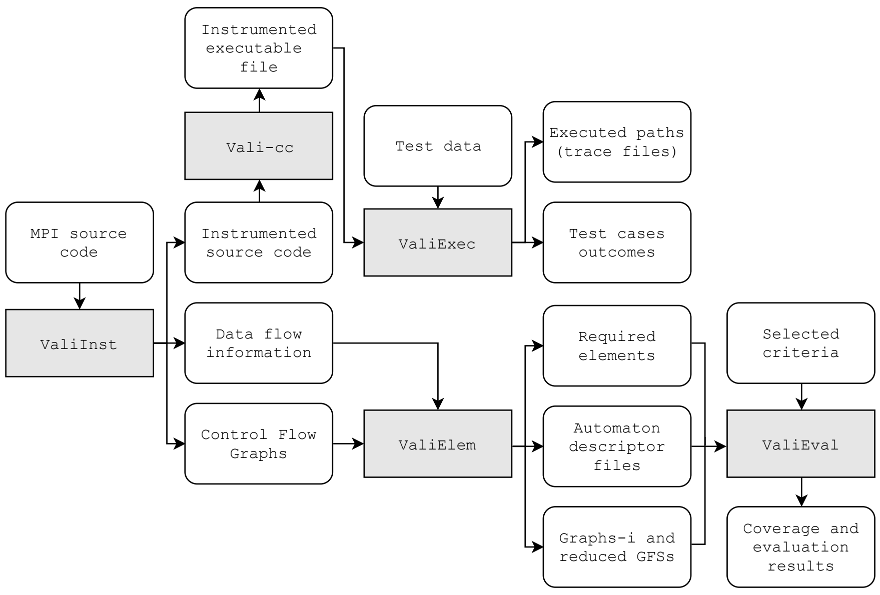
To reduce the demand for tools to support the application of testing criteria in message-passing parallel programs with loops, this paper presents ValiMPI for loops, a software testing tool oriented to test sessions, which supports the testing criteria family, introduced in [Diaz Diaz 2019]. A model that includes the main features of the parallel programs (as synchronization, communication, parallelism and concurrency) was used to support such testing criteria. ValiMPI has been designed to be independent of the environment. The test of any message-passing parallel program is possible. In fact, only one module needs to be reconfigured, which deal with source-code. We implement a new software testing tool for structural testing criteria, which considers the execution of communication events inside loops. ValiMPI solves problems as the loop factor, since structural testing is a static analysis-based technique, in the generation of required elements, and loop execution analysis, which is often seen from a dynamic perspective (the number of loop iterations can be known only on-the-fly). ValiMPI for loops also considers the possibility of nested loops, increasing the complexity in the testing activity.

The remaining of the paper is organized as follows. Section 2 introduces the architecture and main functionalities of ValiMPI tool. Section (3) presents the structural testing criteria contemplating execution of loops, that is implemented in ValiMPI. Section 4 presents the new version of ValiMPI, with implementation details. Section 5 describes a case of study using the test model and the ValiMPI tool. Section 6 presents the related work. Concluding remarks are presented in Section 7.

2. ValiMPI testing tool

ValiPar is a tool for the structural testing of concurrent programs, and it focuses on the source code of a program. ValiMPI is a version of ValiPar tool for MPI/C parallel programs, which has the following testing criteria implemented [De Souza et al. 2005]: All-nodes, All-edges, All-nodes-R, All-nodes-S, All-edges-S, All-C-uses, All-P-uses, and All-S-P-uses. There are other versions of ValiPar besides ValiMPI for different languages and memory paradigms, such as ValiPthreads [Sarmanho et al. 2008, Sarmanho 2009], ValiBPEL [Endo et al. 2008], ValiErlang [Oliveira et al. 2016], and ValiPVM [Souza et al. 2008a]. ValiMPI is an academic tool prototype that is capable of testing parallel programs with complex communication and synchronization patterns, which are common in real applications. Figure 1 shows the architecture of ValiMPI with its main modules. An overview of the four main modules and functionalities of the tool, which are common in all versions, are as follows [Ceolin Hausen 2005, Prado et al. 2015, De Souza et al. 2005]:

Figura 1. ValiMPI tool architecture



ValiInst: generates the PCFG by extracting static information. It receives as an input the program's source code and the output is the Parallel Control Flow Graph (PCFG), information regarding the data flow, flow graphs with deviations, use and definition information and data about message exchange. The instrumentation of the code is one of the essential functionalities in ValiInst. This module retrieves information from the source code to provide inputs for the other modules. 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. ValiInst generates a graph description file, representing CFGs for each function and each process, building a PCFG. A DOT file contains the control and data flows information. 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 test cases.

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. 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 CFG), the path is interrupted when finding a second occurrence of the node with the variable definition n_i. Graphs-i 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 $N^* \ n_i\text{-}p_a \ n_j\text{-}p_b \ N^*$, where N is the set of nodes of CFG^{p_b}. The regular expression for All-nodes-R is $N^* \ n_i\text{-}p \ N^*$, where we can observe the existence of one process involved. For each criterion there could be one or two automata depending on the regular expression for the required elements.

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. The execution trace file contains the intra-process paths and communication primitives exercised by each process, using the pattern $node\text{-}process$ to describe the executed node and the process. Each function is instantiated in ValiExec for the generation of trace files, so a trace file of each process function is created. After executing ValiExec, the user can see the outcome of the program to verify its correctness.

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 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 processes, since the DFA requires execution traces for processes p_a and p_b to retrieve the required elements from both trace files. The implemented automaton reads the trace 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. 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.

3. Structural testing model considering loop execution for parallel programs

The new ValiMPI for loops proposed in this paper gives the support for the test model, considering C/MPI parallel programs, with point-to-point communication patterns and both blocking or non-blocking primitives [Diaz Diaz 2019], coded inside loops (*for, while, do ... while*) for C language. The main testing criteria proposed in [Diaz Diaz 2019] are described as follow.

Control and communication flow structural criteria:

All events-s-loop: the test set must cover all sender nodes n^p where $n^p \in N_{\textit{sync-loop}}$ for each it_p iteration, such that $1 \leq it_p \leq k^d$. Required elements are represented by $n^p_{it_p}$. The set $N_{\textit{sync-loop}}$ represents all nodes with communication primitives inside loops.

All events-r-loop: the test set must cover all receiver nodes where $n^p \in N_{\textit{sync-loop}}$ for each it_p iteration, such that $1 \leq it_p \leq k^d$. Required elements are represented by $n^p_{it_p}$.

All sync-events-loop: the test set must cover all inter-process edges $(n^{p_a}, m^{p_b}_i) \in E_{\textit{s-loop}}$ for each it_p iteration, such that $1 \leq it_p \leq k^d$. Required elements are represented by $(n^{p_a}_{it_p}, m^{p_b}_{it_p})$. The set $E_{\textit{s-loop}}$ represents all edges associated to communication primitives inside loops.

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 $(n^{p_a}_{it_a}, (m^{p_a}_{it_a}, w^{p_b}_{it_b}), x)$.

All defs-recv-loop: the test set must execute paths that cover all definitions in receive primitives, where $x \in def(n^p)$ and $n^p \in N_{\textit{sync-loop}}$ for each it_p iteration such that $1 \leq it_p \leq k^d$. Required elements are represented by $n^p_{it_p}$.

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 $(n^{p_a}_{it_a}, (m^{p_b}_{it_a}, w^{p_b}_{it_b}), v^{p_b}_{it_b}, x^{p_a}, x^{p_b})$.

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 $(n^{p_a}_{it_a}, (m^{p_a}_{it_a}, w^{p_b}_{it_b}), (v^{p_b}_{it_b}, z^{p_b}_{it_b}), x^{p_a}, x^{p_b})$.

4. ValiMPI for message passing programs with loops

We implemented the new set of testing criteria presented in the previous section, which is oriented to the execution of loops in parallel programs [Diaz Diaz 2019]. The source code version of ValiMPI used is described in [Machado 2011]. The extensions in ValiMPI considers a modular approach to minimize *a)* refactoring efforts, *b)* insertion of defects in a functional software prototype testing tool, and *c)* regression test cost.

ValiInst module

ValiInst returns a file for each process, listing the nodes of each loop as $l_i = \{n_1, n_2, ..., n_j\}$, 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 are specified by their identifiers. See Table 1 for a practical case. The loop information files are used by ValiElem for the automated generation of required elements.

ValiElem module

Tabela 1. L.*function* file structure

Loop identifier	Nodes/Nested loops identifier
$l1$	2 3 4 10 $l2$ $l3$ 14 17
$l2$	5 6 7 11 15
$l3$	21 23 22 24

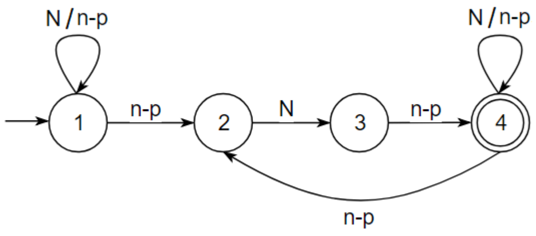
For the generation of the required elements in *ValiElem* module we consider the constant k, which establishes the number of replications of each required element depending on the depth d of the loop, whether there are nested loops or not. We contemplate the following scenarios in the communication between 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. For the execution of ValiElem the k value has been added and is given by:

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

Where, besides the k parameter, NP is the number of processes and "$function_i(p_a, p_b, ...)$" is the function executed by processes p_a, p_b, and "$function_j(p_c, p_d, ...)$" is the function executed by processes p_c, p_d. Graphs-i [Ceolin Hausen 2005, C Hausen et al. 2019] is a method to identify loop paths so that the tool is able to recognize the associations required by our criteria. Graphs-i are generated for each node with variable definitions in each *CFG* for data flow criteria in ValiMPI, creating definition-free paths for c-p and s-uses. The paths in loops are unfolded when generating each sub-graph in a Graph-i, however, the paths of the different iterations are not contemplated. We do not duplicate the paths representing each loop iteration, to avoid generating too many sub-graphs. The number of parent loops (nested loop with depth d) determines the number of times that a communication event must be executed.Therefore, the number of iterations of a determined required element is determined by k^d. The required elements for our criteria contain the iteration number, having the pattern $node\text{-}process\text{-}iteration$ that represents an event. ValiMPI uses DFAs (Deterministic Finite Automata) that recognizes a regular expression representing a required element, in order to evaluate its coverage. Since the description of the automata depends on the criteria, ValiElem implements the DFA through descriptor files containing its transition functions and states. The final state of an automaton recognizes the string of a required element that is present in the trace file [Ceolin Hausen 2005].

One descriptor is used for all required k iterations of a required element. This new approach, in relation to ValiMPI´s previous version, optimizes the process of reading the automata descriptors, avoiding the re-interpretation of each event that must be covered during the test activity. 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. Figure 2 presents an example of DFA for All-events-s-loop criterion. The regular expression representing the required element recognized by the automaton, corresponds to $[N^*\ \ n\text{-}p\ \ N^*\ \ n\text{-}p\ \ N^*]^k$. It indicates that any node of the program can be executed (excluding the node with the send primitive) followed by the send node $n\text{-}p$, subsequently expecting the execution of any node from N. Another occurrence of node $n\text{-}p$ 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.

Figura 2. DFA for All-events-s-loop criterion



ValiEval module

ValiEval reads the automata descriptor files to create a structure representing the states and transitions of an automaton. For the criteria proposed, ValiEval expects that each line of the file contains the number of required iterations for each required element in addition. This approach allows to recover the amount of loop iterations performed by a process 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 of the automaton and the number of times, in order to obtain the number of executed iterations and thus coverage percentage.

Two factors are considered 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. We compare the execution of communication events in a loop (through the trace file) with the number of required iterations (from automaton descriptor file), obtaining the coverage. 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.

5. Case of study and test model exemplification

This case study represents an test session execution in ValiMPI of a simple program. We present the elements and sets generated by the tool, and illustrate how ValiMPI supports the revelation of defects in loops in concurrent programs. For a more in depth analysis, see [Diaz Diaz 2019]. Algorithms 1 and 2 exemplify the use of this new version of the ValiMPI software testing tool, by presenting an implementation for the Greatest Common Divisor - GCD [Dourado 2015]. The algorithms are important to associate the PCFG with required elements, and to clarify the defect. The programs calculate the GCD for three integers with three processes: a Master and two Slaves. The variable *iter* controls the number of iterations, and *result* contains the algorithm's outcome. The Master process reads three values, sending the first and the second to Slave 1, and the second and third to Slave 2. Each slave calculates the GCD of its two numbers and sends back the result to the Master process. If one of the returned values in this first iteration equals to 1 (one) it means that the GCD is 1 and the algorithm finishes; otherwise, both partial results are re-sent to Slave 1 to calculate the final GCD with both the partial values. The Master process finishes the execution when sending to both Slaves the value 0 (lines 39-42 in

Algorithm 1 and line 6 in Algorithm 2). An error related to synchronization events (and labeled as an observability error according to [Delamaro et al. 2007]) can be injected if *values[0]* in line 26 is replaced by *values[1]*, making the Master to receive one of the values to calculate the GCD rather than the result of a Slave process. A test case with values $\{8, 3, 2\}$ does not reveal this defect with just one iteration of the *while* in line 08 of Algorithm 1, because the result for both Slaves is 1 and even with the injected defect, at least one of them is enough to make the *if( )* condition in line 30 *true*. However, if tester is required to cover more iterations in loops with message-passing primitives, he/she may choose a test case with values $\{3, 6, 2\}$, imposing the execution of the second iteration of the loop in line 08 of Algorithm 1. With this new test case, the results of *firstValue* and *secondValue* in the first iteration should be 3 and 2 or 2 and 3 (lines 24 and 26), depending on synchronization. But with the defect in line 26, the results in the first iteration will be 3 and 2 or even 2 and 6, depending on synchronization. In such scenario, the output (GCD result) will be 2 or 1 evidencing the observability defect, having a possible correct output (GCD = 1) despite the error in the code. The execution of different iterations in the loop -and thus, distinct paths- should reveal the defect. A different choice of test data could not reveal the defect, but this is true for every structural testing criteria based on source code coverage.

To illustrate the application of our proposed criteria we present the model exemplification for the GCD algorithms, which generates required elements for all of the proposed criteria, and illustrates with a practical and simple case the elements of our test model, showing the sets, associations and required elements. Consider the PCFG for the CGD program, with the relevant communication edges and definitions. ValiMPI generates the GFC for each functions executed by processes and the required elements. We adapted the PCFG in Figure 3 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 with sender and receiver nodes, nodes with definitions and communicational uses. iii. Present the PCFG in a legible form, representing the elements of our test model.

Dashed lines in the graph (Figure 3) represent synchronization edges ($s\text{-}edge$), and those ones having a communicational use are marked as $s\text{-}use$. Dots indicate that there are other nodes and edges in the graph but they are not relevant here. We do not include the non executable $s\text{-}edges$ in the graph. The definition sets involved in our test model are described in Table 2a. The $def(26^0) = values$ is defined in a receive primitive. Table 2b shows the elements for sets L^p, N_{sync_loop} and E_{s_loop} defined in the test model. Table 3 presents some of the required elements for our criteria, due to their extension. 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. Recapitulating the error example described for GCD program, the defect can be revealed by All-s-uses-loop and All-s-c-uses-loop criteria by covering required elements generated by ValiElem, executing the second iteration of the loop. Such required elements are $(11^1_2,\ (19^1_2,\ 26^0_2),\ values_{p^1})$, $(11^2_2,\ (19^2_2,\ 26^0_1),\ values_{p^2})$, and $(11^1_2,\ (19^1_2, 26^0_2),\ 30^0_2,\ values_{p^1},\ values_{p^0})$. Thus, with the help of ValiMPI and the guide in the selection of adequate test cases, we can reveal defects present in loops in concurrent programs.

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
```

Figura 3. Parallel Control Flow Graph for GCD program

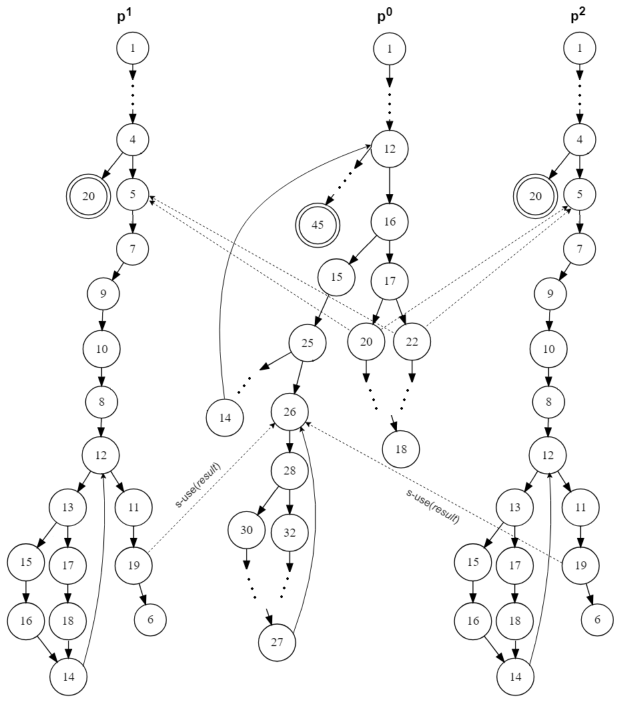


Tabela 2. Test model sets

(a) Number of required elements

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

(b) Number of adequate test cases

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

6. Related work

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 present their main features. We did not find a previous related work where a testing tool implements structural testing criteria for parallel programs considering the execution of loops. Dellapasta (Delaware Parallel Software Testing Aid) is a tool developed to analyze concurrent paths for shared memory programs [Delamaro et al. 2007]. This tool has been used to verify all-du-path coverage criteria, but one of its limitation is that it is not suitable for distributed memory programs [Souza et al. 2008b]. 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 et al. 2007] basing on model testing [Adhianto et al. 2010] as well. ConAn was implemented for a

Tabela 3. 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	$(20_1^0,\ 5_1^1), (20_2^0, 5_2^1), (20_1^0,\ 5_1^2), (20_2^0,\ 5_2^2), (22_1^0,\ 5_1^1), (22_2^0,\ 5_2^1), (22_1^0,\ 5_1^2), (22_2^0,\ 5_2^2), \ldots$
All-defs-recv-loop	$(26_1^0,\ values_{p0}), (26_2^0,\ values_{p0}), (5_1^1,\ values_{p1}), (5_1^1,\ values_{p1}), (5_1^2,\ values_{p2}), (5_2^2,\ values_{p2})$
All-s-uses-loop	$(11_1^1,\ (19_1^1,\ 26_1^0),\ values_{p1}), (11_1^1,\ (19_1^1,\ 26_2^0),\ values_{p1}), (11_2^1,\ (19_2^1,\ 26_1^0),\ values_{p1}), \ldots$
All-s-c-uses-loop	$(11_1^1, (19_1^1, 26_1^0), 30_1^0, values_{p1}, values_{p0}), (11_1^1, (19_1^1, 26_2^0), 30_2^0, values_{p1}, values_{p0}),$
	$(11_2^1, (19_2^1, 26_1^0), 30_1^0, values_{p1}, values_{p0}), (11_2^1, (19_2^1, 26_2^0), 30_2^0, values_{p1}, values_{p0}), \ldots$
All-s-p-uses-loop	$(11_1^1, (19_1^1, 5_1^2), (7_1^2, 9_1^2), values_{p1}, values_{p2}), (11_1^1, (19_1^1, 5_2^2), (7_2^2, 9_2^2), values_{p1}, values_{p2}),$
	$(11_1^1, (19_1^1, 5_1^2), (7_1^2, 8_1^2), values_{p1}, values_{p2}), \ldots$

technique to test java interrupts in another study [Wildman et al. 2004]. The Automated Concurrent Testing (AutoConTest) is a method to automatically generate and run concurrent tests for a given class [Terragni and Cheung 2016]. The generation of such tests bases on the generation sequences of threads, joining them into multi-threaded tests. 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. [DeSouza et al. 2005] propose Intel Message Checker, a tool that verifies the correctness of programs in MPI.

7. Concluding Remarks

In the current literature, there is not a testing tool that implements structural testing criteria for message-passing programs, that considers the execution of loops. Challenges must be approached when implementing a test model that combines static analysis and dynamic structures such as loops. Some considerations are the number of required loop iterations, considerations on nested loops, parallel nondeterminism, and conceptual adaptations on data and control flows and implications in the implementation for coverage. 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 programs of the benchmark. 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. We implemented the proposed criteria in ValiMPI tool, for the automated generation of required elements and coverage evaluation. We also contribute to the enhancement of

ValiMPI, which is the base for several investigations in our research group. We decide 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. The approaches, methods and design that we followed helped us to improve the developed functionalities in ValiMPI for the proposed criteria.

Referências

Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., and Tallent, N. R. (2010). HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. *Concurrency Computation Practice and Experience*, 22(6):685–701.

Alghamdi, A. and Eassa, F. (2019). Software testing techniques for parallel systems: A survey. 19:176–186.

Ammann, P. and Offutt, J. (2008). *Introduction to Software Testing*.

C Hausen, A., Vergilio, S., and Souza, S. (2019). A tool for structural testing of mpi programs.

Ceolin Hausen, A. (2005). Valimpi : uma ferramenta de teste estrutural para programas paralelos em ambiente de passagem de mensagem. *Master's thesis, Universidade Federal do Paraná*, page 92.

Chen, Q., Wang, L., Yang, Z., and Stoller, S. (2009). HAVE: Integrated Dynamic and Static Analysis for Atomicity Violations. *12th Int. Conf. on Fundamental Approaches to Software Engineering (FASE)*.

De Souza, S., Vergilio, S. R., De Souza, P. S. L., Da Silva Simão, A., Gonçalves, T. B., De Melo Lima, A., and Hausen, A. C. (2005). ValiPar: A testing tool for message-passing parallel programs. *17th Int. Conf. on Software Engineering and Knowledge Engineering, SEKE 2005*, (November 2015):386–391.

Delamaro, M., Maldonado, J., and Jino, M. (2007). *Introdução ao teste de software*. CAMPUS - RJ.

DeSouza, J., Kuhn, B., de Supinski, B. R., Samofalov, V., Zheltov, S., and Bratanov, S. (2005). Automated, scalable debugging of mpi programs with intel&reg; message checker. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, SE-HPCS '05, pages 78–82, New York, NY, USA. ACM.

Diaz Diaz, S. M. (2019). Structural testing criteria for concurrent programs considering loop executions. *Master's dissertation, Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo*.

Dourado, G. G. M. (2015). 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*.

Endo, A. T., d. S. Simão, A., d. R. S. d. Souza, S., and d. Souza, P. S. L. (2008). Web services composition testing: A strategy based on structural testing of parallel programs. In *Testing: Academic Industrial Conference - Practice and Research Techniques (taic part)*, pages 3–12.

Machado, M. C. C. (2011). Estudo e definição de mecanismos para redução do custo de aplicação do teste de programas concorrentes. *Master's thesis, USP*, page 92.

Oliveira, A., Lopes de Souza, P., and Souza, S. (2016). Valierlang: A structural testing tool for erlang programs. pages 1–10.

Prado, R. R., Souza, P. S. L., Dourado, G. G. M., Souza, S. R. S., Estrella, J. C., Bruschi, S. M., and Lourenco, J. (2015). Extracting static and dynamic structural information from java concurrent programs for coverage testing. In *CLEI 2015*, pages 1–8.

Sarmanho, F. S. (2009). Teste de programas concorrentes com memória compartilhada. *Master's thesis, USP*, page 92.

Sarmanho, F. S., Souza, P. S. L., Souza, S. R. S., and Simão, A. S. (2008). Structural testing for semaphore-based multithread programs. In Bubak, M., van Albada, G. D., Dongarra, J., and Sloot, P. M. A., editors, *Computational Science – ICCS 2008*, pages 337–346, Berlin, Heidelberg. Springer Berlin Heidelberg.

Simão, A., Maldonado, J., Vincenzi, A., and Santana, A. (2003). A language for the description of program instrumentation and automatic generation of instrumenters. *CLEI Electron. J.*, 6.

Souza, P. L., Sawabe, E. T., Silva Simão, A., Vergilio, S. R., and Rocio Senger De Souza, S. (2008a). 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*, pages 257–264, Berlin, Heidelberg. Springer-Verlag.

Souza, S. c., Vergilio, R., Souza, P., Simao, S., and Hausen, A. (2008b). Structural testing criteria for message-passing parallel programs. *Concurrency Computation Practice and Experience*, 20(16):1893–1916.

Terragni, V. and Cheung, S.-C. (2016). Coverage-driven Test Code Generation for Concurrent Classes. In *Proceedings of the 38th Int. Conf. on Software Engineering*, ICSE '16, pages 1121–1132, New York, NY, USA. ACM.

Wildman, L., Long, B., and Strooper, P. (2004). Testing Java interrupts and timed waits. In *Software Engineering Conference, 2004. 11th Asia-Pacific*, pages 438–447.