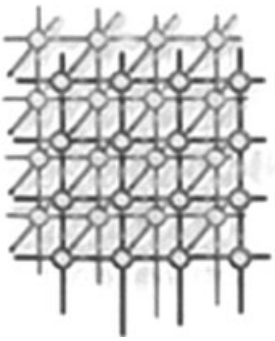
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2008; **20**:1893–1916
Published online 17 March 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.1297



Structural testing criteria for message-passing parallel programs

S. R. S. Souza[1,*,†], S. R. Vergilio[2], P. S. L. Souza[1], A. S. Simão[1] and A. C. Hausen[2]

[1]*Departamento de Sistemas de Computação, ICMC/USP, São Carlos, SP, Brazil*
[2]*Departamento de Informática, UFPR, Curitiba, PR, Brazil*

SUMMARY

Parallel programs present some features such as concurrency, communication and synchronization that make the test a challenging activity. Because of these characteristics, the direct application of traditional testing is not always possible and adequate testing criteria and tools are necessary. In this paper we investigate the challenges of validating message-passing parallel programs and present a set of specific testing criteria. We introduce a family of structural testing criteria based on a test model. The model captures control and data flow of the message-passing programs, by considering their sequential and parallel aspects. The criteria provide a coverage measure that can be used for evaluating the progress of the testing activity and also provide guidelines for the generation of test data. We also describe a tool, called ValiPar, which supports the application of the proposed testing criteria. Currently, ValiPar is configured for parallel virtual machine (PVM) and message-passing interface (MPI). Results of the application of the proposed criteria to MPI programs are also presented and analyzed. Copyright © 2008 John Wiley & Sons, Ltd.

Received 22 March 2007; Revised 25 November 2007; Accepted 3 December 2007

KEY WORDS: parallel software testing; coverage criteria; testing tool; PVM; MPI

1. INTRODUCTION

Parallel computing is essential to reduce the execution time in many different applications, such as weather forecast, dynamic molecular simulation, bio-informatics and image processing. According to Almasi and Gottlieb [1], there are three basic approaches to build parallel software: (i) automatic

*Correspondence to: S. R. S. Souza, Instituto de Ciências Matemáticas e de Computação, USP Av. Trabalhador São-carlense, 400-Centro Caixa Postal: 668-CEP: 13560-970, São Carlos, SP, Brazil.
†E-mail: srocio@icmc.usp.br

Contract/grant sponsor: CNPq; contract/grant number: 552213/2002-0

environments that generate parallel code from sequential algorithms; (ii) concurrent programming languages such as CSP and ADA; and (iii) extensions for traditional languages, such as C and Fortran, implemented by message-passing environments. These environments include a function library that allows the creation and communication of different processes and, consequently, the development of parallel programs, usually running in a cluster of computers. The most known and used message-passing environments are parallel virtual machine (PVM) [2] and message-passing interface (MPI) [3]. Such environments have gained importance in the last decade and they are the focus of our work.

Parallel software applications are usually more complex than sequential ones and, in many cases, require high reliability levels. Thus, the validation and test of such applications are crucial activities. However, parallel programs present some features that make the testing activity more complex, such as non-determinism, concurrence, synchronization and communication. In addition, the testing teams are usually not trained for testing this class of applications, which makes the test of parallel programs very expensive. For sequential programs, many of the testing problems were reduced with the introduction of testing criteria and the implementation of supporting tools. A testing criterion is a predicate to be satisfied by a set of test cases and can be used as a guideline for the generation of test data. Structural criteria utilize the code, the implementation and structural aspects of the program to select test cases. They are usually based on a control-flow graph (CFG) and definitions and uses of variables in the program [4].

Yang [5] describes some challenges to test parallel programs: (1) developing static analysis; (2) detecting unintentional races and deadlock situations in non-deterministic programs; (3) forcing a path to be executed when non-determinism might exist; (4) reproducing a test execution using the same input data; (5) generating the CFG of non-deterministic programs; (6) providing a testing framework as a theoretical base for applying sequential testing criteria to parallel programs; (7) investigating the applicability of sequential testing criteria to parallel program testing; and (8) defining test coverage criteria based on control and data flows.

There have been some initiatives to define testing criteria for shared memory parallel programs [6–11]. Other works have investigated the detection of race conditions [12–14] and mechanisms to replay testing for non-deterministic programs [15,16]. However, few works are found that investigate the application of the testing coverage criteria and supporting tools in the context of message-passing parallel programs. For these programs, new aspects need to be considered. For instance, data-flow information must consider that an association between one variable definition and one use can occur in different addressing spaces. Because of this different paradigm, the investigation of challenges mentioned above, in the context of message-passing parallel programs, is not a trivial task and presents some difficulties. To overcome these difficulties, we present a family of structural testing criteria for this kind of programs, based on a test model, which includes their main features, such as synchronization, communication, parallelism and concurrency. Testing criteria were defined to exploit the control and data flows of these programs, considering their sequential and parallel aspects. The main contribution of the testing criteria proposed in this paper is to provide a coverage measure that can be used for evaluating the progress of the testing activity. This is important to evaluate the quality of test cases as well as to consider that a program has been tested enough.

The practical application of a testing criterion is possible only if a tool is available. Most existent tools for message-passing parallel programs aid only the simulation, visualization and

DOI: 10.1002/cpe

debugging [16–21]. They do not support the application of testing criteria. To fulfill the demand for tools to support the application of testing criteria in message-passing parallel programming and to evaluate the proposed criteria, we implemented a tool, called ValiPar, which supports the application of the proposed testing criteria and offers two basic functionalities: the selection and evaluation of test data. ValiPar is independent of the message-passing environment and can be configured to different environments and languages. Currently, ValiPar is configured for PVM and MPI programs, in C language. ValiPar was used in an experiment with MPI programs to evaluate the applicability of the proposed criteria, whose results are presented in this paper.

The remainder of this paper is organized as follows. In Section 2, we present the basic concepts and the test model adopted for the definition of the testing criteria. We also introduce the specific criteria for message-passing programs and show an example of usage. In Section 3, the main functionalities of ValiPar are presented and some implementation aspects are discussed. In Section 4, the results of the testing criteria application are presented. In Section 5, related work is presented. Concluding remarks are presented in Section 6.

2. STRUCTURAL TESTING CRITERIA FOR MESSAGE-PASSING PROGRAMS

In this section, we introduce a set of testing criteria defined based on a model that represents the main characteristics of the message-passing parallel programs. This test model is first presented. In order to illustrate the application of the proposed testing criteria, an example of use is presented in Section 2.3.

2.1. Test model and basic concepts

A test model is defined to capture the control, data and communication information of the message-passing parallel programs. This model is based on Yang and Chung's work [11]. The test model considers that a fixed and known number n of processes is created at the initialization of the parallel application. These processes may execute different programs. However, each one executes its own code in its own memory space.

The communication between processes uses two basic mechanisms. The first one is the *point-to-point* communication. A process can send a message to another one using primitives such as *send* and *receive*. The second one is named *collective* communication; a process can send a message to all processes in the application (or to a particular group of them). In our model the collective communication happens in only one pre-defined domain (or context) that includes all the processes in the parallel application. The primitives for collective communication are represented in terms of several basic *sends*.

The parallel program is given by a set of n parallel processes $Prog = \{p^0, p^1, \ldots, p^{n-1}\}$. Each process p has its own control flow graph, CFG^p, which is built by using the same concepts of traditional programs [4]. In short, a CFG of a process p is composed by a set of nodes N^p and a set of edges E^p. These edges that link two nodes of a same process is called intra-process. Each node n in the process p is represented by the notation n^p and corresponds to a set of commands that are sequentially executed or can be associated with a communication primitive (*send*

DOI: 10.1002/cpe

or *receive*). The communication primitives are associated with separate nodes and are represented by the notations `send(p,k,t)` (respectively, `receive(p,k,t)`), meaning that the process p sends (respectively, receives) a message with tag t to (respectively, from) the process k. Note that the model considers blocking and non-blocking receives, such that all possible interleaving between send–receive pairs are represented. The path analysis, described next, permits one to capture the send–receive matching during the parallel program execution.

Each CFG^p has two special nodes: the entry and exit nodes, which correspond to the first and last statements in p, respectively. An edge links a node to another one.

A parallel program *Prog* is associated with a parallel control-flow graph (*PCFG*), which is composed of CFG^p (for $p = 0 \ldots n-1$) and of the representation of the communication between the processes. N and E represent the set of nodes and edges of the *PCFG*, respectively.

Two subsets of N are defined: N_s and N_r, composed of nodes that are associated with *send* and *receive* primitives, respectively. With each $n_i^p \in N_s$, a set R_i^p is associated, such that

$$R_i^p = \{n_j^k \in N_r | \exists \, (\texttt{send(p,k,t)} \text{ at node } n_i^p \text{ and } \texttt{receive(k,p,t)} \, atnode n_j^k), \ \forall k \neq p \wedge k = 0 \ldots n-1\}$$

i.e. R_i^p contains the nodes that can receive a message sent by node n_i^p.

Using the above definitions, we also define the following sets:

- *set of inter-processes edges* (E_s): contains edges that represent the communication between two processes, such that

$$E_s = \{(n_j^{p1}, n_k^{p2}) \mid n_j^{p1} \in N_s, n_k^{p2} \in R_j^{p1}\}$$

- *set of edges* (E): contains all edges, such that

$$E = E_s \cup \bigcup_{p=0}^{n-1} E^p$$

A path π^p in a CFG^p is called an intra-process path. It is given by a finite sequence of nodes, $\pi^p = (n_1^p, n_2^p, \ldots, n_m^p)$, where $(n_i^p, n_{i+1}^p) \in E^p$. $\Pi = (\pi^0, \pi^1, \ldots, \pi^k, S)$ is an inter-processes path of the concurrent execution of *Prog*, where S is the set of synchronization pairs that were executed, such that $S \subseteq E_s$. Observe that the synchronization pairs of S can be used to establish a conceptual path $(n_1^{p1}, n_2^{p1}, \ldots, n_i^{p1}, k_j^{p2} \ldots n_m^{p1})$ or $(k_1^{p2}, k_2^{p^2}, \ldots, n_i^{p1}, k_j^{p2} \ldots k_l^p p2)$. Such paths contain inter-processes edges.

An intra-processes path $\pi^p = (n_1, n_2, \ldots, n_m)$ is simple if all its nodes are distinct, except possibly the first and the last ones. It is loop free if all its nodes are distinct. It is complete if n_1 and n_m are the entry and exit nodes of CFG^p, respectively. We extend these notions to inter-processes paths. An inter-processes path $\Pi = (\pi^0, \pi^1, \ldots, \pi^{n-1}, S)$ is simple if all π^i are simple. It is loop free if all π^i are loop free. It is complete if all π^i are complete. Only complete paths are executed by the test cases, i.e. all the processes execute complete paths. A node, edge or a sub-path is covered (or exercised) if a complete path that includes them is executed.

DOI: 10.1002/cpe

A variable x is defined when a value is stored in the corresponding memory position. Typical definition statements are assignment and input commands. A variable is also defined when it is passed as an output parameter (reference) to a function. In the context of message-passing environments, we need to consider the communication primitives. For instance, the primitive *receive* sets one or more variables with the value t received in the message; thus, this is considered a definition. Therefore, we define:

$$def(n^p) = \{x \mid x \text{ is defined in} n^p\}$$

The use of variable x occurs when the value associated with x is referred. The uses can be:

1. a *computational use* (*c-use*): occurs in a computation statement, related to a node n^p in the *PCFG*;
2. a *predicate use* (*p-use*): occurs in a condition (predicate) associated with control-flow statements, related to an intra-processes edge (n^p, m^p) in the *PCFG*; and
3. a *communication use* (*s-use*): occurs in a communication statement (communication primitives), related to an inter-processes edge $(n^{p1}, m^{p2}) \in E_s$.

A path $\pi = (n_1, n_2, \ldots, n_j, n_k)$ is definition clear with respect to (w.r.t.) a variable x from node n_1 to node n_k or edge (n_j, n_k), if $x \in def(n_1)$ and $x \notin def(n_i)$, for $i = 2 \ldots j$.

Similar to traditional testing, we establish pairs composed of definitions and uses of the same variables to be tested [4]. Three kinds of associations are introduced:

c-use association is defined by a triple (n^p, m^p, x), such that $x \in def(n^p)$, m^p has a c-use of x and there is a definition-clear path w.r.t. x from n^p to m^p.

p-use association is defined by a triple $(n^p, (m^p, k^p), x)$, such that $x \in def(n^p)$, (m^p, k^p) has a p-use of x and there is a definition-clear path w.r.t. x from n^p to (m^p, k^p).

s-use association is defined by a triple $(n^{p1}, (m^{p1}, k^{p2}), x)$, such that $x \in def(n^{p1})$, (m^{p1}, k^{p2}) has an s-use of x and there is a definition-clear path w.r.t. x from n^{p1} to (m^{p1}, k^{p2}).

Note that p-use and c-use associations are intra-processes, i.e. the definition and the use of x occur in the same process p. These associations are usually required if we apply the traditional testing criteria to each process separately. An s-use association supposes the existence of a second process and it is an inter-processes association; s-use associations allow the detection of communication faults (in the use of *send* and *receive* primitives). Considering this context, we propose another kind of inter-processes associations to discover communication and synchronization faults:

s–c-use association is given by $(n^{p1}, (m^{p1}, k^{p2}), l^{p2}, x^{p1}, x^{p2})$, where there is an s-use association $(n^{p1}, (m^{p1}, k^{p2}), x^{p1})$ and a c-use association (k^{p2}, l^{p2}, x^{p2}).

s–p-use association is given by $(n^{p1}, (m^{p1}, k^{p2}), (n^{p2}, m^{p2}), x^{p1}, x^{p2})$, where there is an s-use association $(n^{p1}, (m^{p1}, k^{p2}), x^{p1})$ and a p-use association $(k^{p2}, (n^{p2}, m^{p2}), x^{p2})$.

2.2. Structural testing criteria

In this section, we propose two sets of structural testing criteria for message-passing parallel programs, based on test model and definitions presented in previous section. These criteria allow the testing of sequential and parallel aspects of the programs.

DOI: 10.1002/cpe

2.2.1. *Testing criteria based on the control and communication flows*

Each CFG^p (for $p = 0...n-1$) can be tested separately by applying the traditional criteria all-edges and all-nodes. Our objective, however, is also to test the communications in the *PCFG*. Thus, the testing criteria introduced below are based on the types of edges (inter- and intra-processes edges).

- *all-nodes-s criterion*: The test sets must execute paths that cover all the nodes $n_i^p \in Ns$.
- *all-nodes-r criterion*: The test sets must execute paths that cover all the nodes $n_i^p \in Nr$.
- *all-nodes criterion*: The test sets must execute paths that cover all the nodes $n_i^p \in N$.
- *all-edges-s criterion*: The test sets must execute paths that cover all the edges $(n_j^{p1}, n_k^{p2}) \in Es$.
- *all-edges criterion* the test sets must execute paths that cover all the edges $(n_j, n_k) \in E$.

Other criteria could be proposed such as all-paths in the CFG^p and in the *PCFG* (intra- and inter-processes paths). These criteria generally require an infinite number of elements, due to loops in the program. Thus, in such cases, only loop-free paths should be required or selected.

2.2.2. *Testing criteria based on data and message-passing flows*

These criteria require associations between definitions and uses of variables. The objective is to validate the data flow between the processes when a message is passed.

- *all-defs criterion*: For each node n_i^p and each $x \in def(n_i^p)$, the test set must execute a path that covers an association (c-use, p-use or s-use) w.r.t. x.
- *all-defs-s criterion*: For each node n_i^p and each $x \in def(n_i^p)$, the test set must execute a path that covers an inter-processes association (s–c-use or s–p-use) w.r.t. x. In the case where such association does not exist, another one should be selected to exercise the definition of x.
- *all-c-uses criterion*: The test set must execute paths that cover all the c-use associations.
- *all-p-uses criterion*: The test set must execute paths that cover all the p-use associations.
- *all-s-uses criterion*: The test set must execute paths that cover all the s-use associations.
- *all-s–c-uses criterion*: The test set must execute paths that cover all the s–c-use associations.
- *all-s–p-uses criterion*: The test set must execute paths that cover all the s–p-use associations.

Required elements are the minimal information that must be covered to satisfy a testing criterion. For instance, the required elements for the criterion *all-edges-s* are all possible synchronization between parallel processes. However, satisfying a testing criterion is not always possible, due to infeasible elements. An element required by a criterion is infeasible if there is no set of values for the parameters, the input and global variables of the program that executes a path that cover that element. The determination of infeasible paths is an undecidable problem [22].

Non-determinism is another issue that makes the testing activity difficult. An example is presented in Figure 1. Suppose that the nodes 8^1 and 9^1 in p^1 have non-deterministic *receives* and in the nodes 2^0 (p^0) and 2^2 (p^2) have *sends* to p^1. The figure illustrates the possible synchronizations between these processes. These synchronizations represent correct behavior of the application. Therefore, during the testing activity it is essential to guarantee that these synchronizations are executed.

DOI: 10.1002/cpe

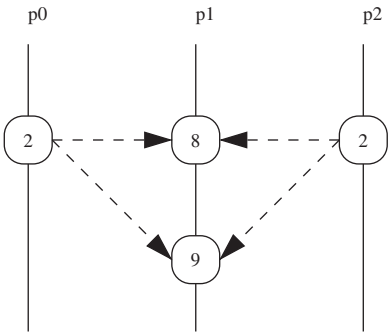


Figure 1. Example of non-determinism.

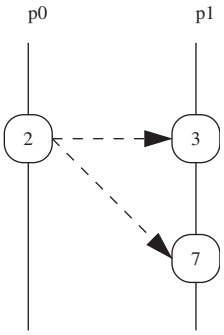


Figure 2. Example of non-blocking receive.

Controlled execution is a mechanism used to achieve deterministic execution, i.e. two executions of the program with the same input are guaranteed to execute the same instruction sequence [15] (and the same synchronization sequence). This mechanism is implemented in ValiPar tool and is described in Section 3.

Figure 2 illustrates an example with *non-blocking receive*. Suppose that the nodes 3^1 and 7^1 in p^1 have *non-blocking receive*. Two synchronization edges are possible, but only one is exercised in each execution. During the path analysis, it is possible to determine the edges that were covered. This information is available in path Π, which is obtained by instrumentation of the parallel program. This instrumentation is described in Section 3.

2.3. An example

In order to illustrate the introduced definitions, consider the GCD program in PVM (Figure 3), described in [23]. This program uses four parallel processes (p^m, p^0, p^1, p^2) to calculate the maximum common divisor of three numbers. The master process p^m (Figure 3(a)) creates the

DOI: 10.1002/cpe

```
/* Master program GCD - mgcd.c */
#include<stdio.h>
#include "pvm3.h"
extern void pack(int);
extern int unpack();
int main(){
/*1*/ int x,y,z, S[3];
/*1*/   scanf("%d%d%d",&x,&y,&z);
/*1*/   pvm_spawn("gcd",(char**)0,0,"",3,S);
/*2*/   pack(&x);
/*2*/   pack(&y);
/*2*/   pvm_send(S[0],1);
/*3*/   pack(&y);
/*3*/   pack(&z);
/*3*/   pvm_send(S[1],1);
/*4*/   pvm_recv(-1,2);
/*4*/   x = unpack();
/*5*/   pvm_recv(-1,2);
/*5*/   y = unpack();
/*6*/   if ((x>1)&&(y>1))        {
/*7*/         pack(&x);
/*7*/         pack(&y);
/*7*/         pvm_send(S[2],1);
/*8*/         pvm_recv(-1,2);
/*8*/         z = unpack();    }
/*9*/   else { pvm_kill(S[2]);
/*9*/          z = 1;    }
/*10*/  printf("%d", z);
/*10*/  pvm_exit();      }
```

(a)

```
/* Slave program GCD - gcd.c */
#include<stdio.h>
#include"pvm3.h"
extern void pack(int);
extern int unpack();
int main(){
/*1*/   int tid,x,y;
/*1*/   tid = pvm_parent();
/*2*/   pvm_recv(tid,-1);
/*2*/   x = unpack();
/*2*/   y = unpack();
/*3*/   while (x != y){
/*4*/     if (x<y)
/*5*/           y = y-x;
/*6*/     else
/*6*/           x = x-y;
/*7*/   }
/*8*/   pack(&x);
/*8*/   pvm_send(tid,2);
/*9*/   pvm_exit();}
```

(b)

Figure 3. GCD program in PVM: (a) master process and (b) slave process.

slave processes p^0, p^1 and p^2, which run 'gcd.c' (Figure 3(b)). Each slave waits (blocked *receive*) two values sent by p^m and calculates the maximum divisor for these values. To finish, the slaves send the calculated values to p^m and terminate their executions. The computation can involve p^0, p^1 and p^2 or only p^0 and p^1, depending on the input values. In p^m, the *receive* commands (nodes 4^m, 5^m and 8^m) are non-deterministic; thus which message will be received in each *receive* command depends on the execution time of each process.

The *PCFG* is presented in Figure 4. The numbers on the left of the source code (Figure 3) represent the nodes in the graph. Inter-processes edges are represented by dotted lines. For simplification reasons, in this figure, only some inter-processes edges (and related *s-use*) are represented. Table I presents the sets $def(n_i^p)$. Table II contains the values of all sets introduced in Section 2.1.

In Table III, we present some elements required by the structural testing criteria introduced in Section 2.2. Test inputs must be generated in order to exercise each possible required element. For example, considering the test input $\{x = 1, y = 2, z = 1\}$, the execution path is $\Pi = (\pi^m, \pi^0, \pi^1, S)$, where $\pi^m = \{1^m, 2^m, 3^m, 4^m, 5^m, 6^m, 9^m, 10^m\}$, $\pi^0 = \{1^0, 2^0, 3^0, 4^0, 5^0, 7^0, 3^0, 8^0, 9^0\}$, $\pi^1 = \{1^1, 2^1, 3^1, 4^1, 6^1, 7^1, 3^1, 8^1, 9^1\}$, $S = \{(2^m, 2^0), (3^m, 2^1), (8^0, 4^m), (8^1, 5^m)\}$. Note that p^2 does not execute any path because the result has been already produced by p^0 and p^1. Owing to the *receive* non-deterministic in nodes 4^m and 5^m, four synchronization edges will be possible: $(8^0, 4^m)$, $(8^0, 5^m)$, $(8^1, 4^m)$, $(8^1, 5^m)$ and only two of them are exercised for each execution of path Π depending on the execution time ($(8^0, 4^m)$ or $(8^0, 5^m)$, $(8^1, 4^m)$ or $(8^1, 5^m)$). In each program execution, it is necessary to determine the inter-processes edges that were executed. This aspect is related to the evaluation of the test cases and was considered in the implementation of ValiPar, described in Section 3.

DOI: 10.1002/cpe

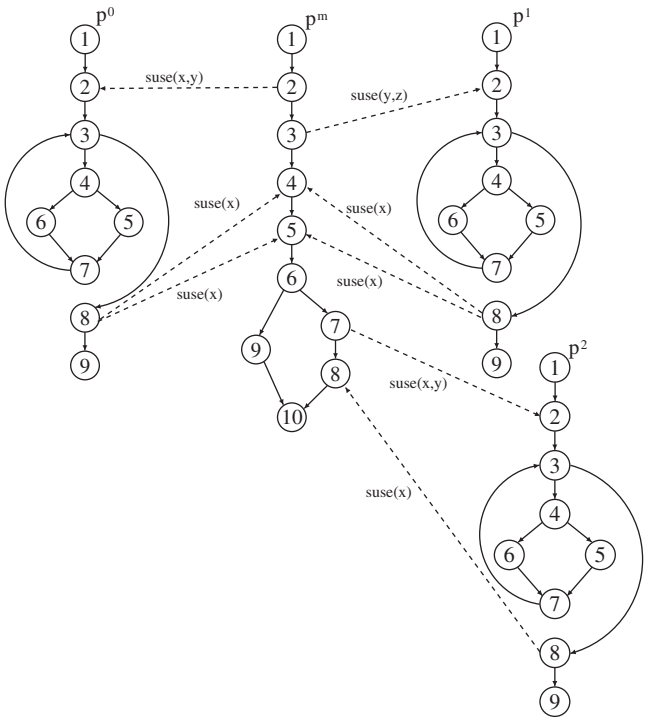


Figure 4. Parallel control-flow graph for GCD program.

Table I. Definition sets for GCD program.

$def(1^m) = \{x, y, z, S\}$	$def(1^0) = \{tid\}$
$def(4^m) = \{x\}$	$def(2^0) = \{x, y\}$
$def(5^m) = \{y\}$	$def(5^0) = \{y\}$
$def(8^m) = \{z\}$	$def(6^0) = \{x\}$
$def(9^m) = \{z\}$	
$def(1^1) = \{tid\}$	$def(1^2) = \{tid\}$
$def(2^1) = \{x, y\}$	$def(2^2) = \{x, y\}$
$def(5^1) = \{y\}$	$def(5^2) = \{y\}$
$def(6^1) = \{x\}$	$def(6^2) = \{x\}$

2.4. Revealing faults

The efficacy (in terms of fault revealing) of the proposed criteria can be illustrated by some kinds of faults that could be present in program GCD (Figure 3) and showing how the criteria contribute to reveal these kinds of faults. The fault situations are based on the works of Howden [24] and

DOI: 10.1002/cpe

Table II. Sets of the test model for GCD program.

$n = 4$
$Prog = \{p^m, p^0, p^1, p^2\}$
$N = \{1^m, 2^m, 3^m, 4^m, 5^m, 6^m, 7^m, 8^m, 9^m, 10^m, 1^0, 2^0, 3^0, 4^0, 5^0, 6^0, 7^0,$ $8^0, 9^0, 1^1, 2^1, 3^1, 4^1, 5^1, 6^1, 7^1, 8^1, 9^1, 1^2, 2^2, 3^2, 4^2, 5^2, 6^2, 7^2, 8^2, 9^2\}$
$N_s = \{2^m, 3^m, 7^m, 8^0, 8^1, 8^2\}$ (nodes with *pvm_send()*)
$N_r = \{4^m, 5^m, 8^m, 2^0, 2^1, 2^2\}$ (nodes with *pvm_recv()*)
$R_2^m = \{2^0, 2^1, 2^2\}$
$R_3^m = \{2^0, 2^1, 2^2\}$
$R_7^m = \{2^0, 2^1, 2^2\}$
$R_8^0 = \{4^m, 5^m, 8^m\}$
$R_8^1 = \{4^m, 5^m, 8^m\}$
$R_8^2 = \{4^m, 5^m, 8^m\}$
$E = E_i^p \cup E_s$
$E_i^m = \{(1^m, 2^m), (2^m, 3^m), (3^m, 4^m), (4^m, 5^m), (5^m, 6^m), (6^m, 7^m),$ $(7^m, 8^m), (8^m, 10^m), (6^m, 9^m), (9^m, 10^m)\}$
$E_i^0 = \{(1^0, 2^0), (2^0, 3^0), (3^0, 4^0), (4^0, 5^0), (4^0, 6^0), (5^0, 7^0), (6^0, 7^0),$ $(7^0, 3^0), (3^0, 8^0), (8^0, 9^0)\}$
$E_i^1 = \{(1^1, 2^1), (2^1, 3^1), (3^1, 4^1), (4^1, 5^1), (4^1, 6^1), (5^1, 7^1), (6^1, 7^1),$ $(7^1, 3^1), (3^1, 8^1), (8^1, 9^1)\}$
$E_i^2 = \{(1^2, 2^2), (2^2, 3^2), (3^2, 4^2), (4^2, 5^2), (4^2, 6^2), (5^2, 7^2), (6^2, 7^2),$ $(7^2, 3^2), (3^2, 8^2), (8^2, 9^2)\}$
$E_s = \{(2^m, 2^0), (2^m, 2^1), (2^m, 2^2), (3^m, 2^0), (3^m, 2^1), (3^m, 2^2), (7^m, 2^0),$ $(7^m, 2^1), (7^m, 2^2), (8^0, 4^m), (8^0, 5^m), (8^0, 8^m), (8^1, 4^m), (8^1, 5^m), (8^1, 8^m),$ $(8^2, 4^m), (8^2, 5^m), (8^2, 8^m)\}$

Krawczyk and Wiszniewski [23], which describe typical faults in traditional and parallel programs, respectively.

Howden [24] introduces two types of faults in traditional programs: computation and domain faults. The first one occurs when the result of a computation for an input of the program domain is different from the expected result. The second one occurs when a path that is different from the expected one is executed. For example, in the process slave (gcd.c), replacing the command of node 5^1 '$y = y - x$' by the incorrect command '$y = y + x$' corresponds to a computation fault. A domain fault can be illustrated by changing the predicate $(x < y)$ in edge $(4^1, 5^1)$ by the incorrect predicate $(x > y)$, taking a different path during the execution. These faults are revealed by applying traditional criteria, all-edges, all-nodes, etc., and testing each *CFG* separately. Executing the test input $\{x = 1, y = 2, z = 1\}$ the node 5^1 is covered and the first fault is revealed. Considering the second fault, the test input $\{x = 2, y = 3, z = 2\}$ executes a path that covers the edge $(4^1, 5^1)$ and reveals the fault. For both inputs, the program executes the loop of node 3 (gcd.c) forever, and a failure is produced. These situations illustrate the importance of investigating the application of criteria for sequential testing in parallel software.

In the context of parallel programs, a computation fault can be related to a communication fault. To illustrate this fact, consider that in slave process (Figure 3(b)) the variable y is mistakenly

DOI: 10.1002/cpe

Table III. Some elements required by the proposed testing criteria for GCD program.

all-nodes-*s*	$2^m, 3^m, 7^m, 8^0, 8^1, 8^2$
all-nodes-*r*	$4^m, 5^m, 8^m, 2^0, 2^1, 2^2$
all-nodes	$1^m, 2^m, 3^m, 4^m, 5^m, 6^m, 7^m, 8^m, 9^m, \ldots, 1^0, 2^0, 3^0, \ldots, 1^1, 2^1, 3^1, \ldots$
all-edges-*s*	$(2^m, 2^0), (2^m, 2^1), (2^m, 2^2), (3^m, 2^0), (3^m, 2^1), (3^m, 2^2), (7^m, 2^2), (8^0, 4^m), (8^0, 5^m),$ $(8^0, 8^m), (8^1, 4^m), (8^1, 5^m), (8^1, 8^m), 8^2, 4^m), (8^2, 5^m), (8^2, 8^m) \ldots$
all-edges	$(1^m, 2^m), (2^m, 3^m), \ldots, (1^0, 2^0), (2^0, 3^0), \ldots, (1^1, 2^1), (2^1, 3^1), \ldots (2^m, 2^0), (2^m, 2^1) \ldots$
all-defs	$(8^m, 10^m, z), (2^0, 5^0, x), (2^0, 6^0, x), (2^0, (3^0, 4^0), x), (2^0, 6^0, y) \ldots$
all-defs-*s*	$(1^m, (2^m, 2^0), 5^0, x, x), (1^m, (2^m, 2^0), 6^0, y, y),$ $(1^m, (2^m, 2^0), (4^0, 5^0), y, y), (1^m, (3^m, 2^0), 5^0, z, y), \ldots$
all-*c*-uses	$(1^m, 10^m, z), (8^m, 10^m, z), (2^0, 8^0, x) \ldots$
all-*p*-uses	$(4^m, (6^m, 7^m), x), (4^m, (6^m, 9^m), x), (5^m, (6^m, 7^m), y),$ $(5^m, (6^m, 9^m), y), (2^0, (3^0, 4^0), x), (2^0, (3^0, 8^0), y) \ldots$
all-*s*-uses	$(1^m, (2^m, 2^0), x, y), (1^m, (2^m, 2^1), x, y), (1^m, (3^m, 2^0), y, z),$ $(4^m, (7^m, 2^2), x), (5^m, (7^m, 2^0), y), (5^m, (7^m, 2^1), y), \ldots$
all-*s*–*c*-uses	$(1^m, (2^m, 2^0), 5^0, x, x), (1^m, (2^m, 2^0), 6^0, x, x), (1^m, (2^m, 2^0), 5^0, y, y),$ $(1^m, (2^m, 2^0), 6^0, y, y), (1^m, (2^m, 2^1), 6^1, x, x),$ $(1^m, (3^m, 2^1), 6^1, x, x), (2^0, (8^0, 8^m), 10^m, x, z), \ldots$
all-*s*–*p*-uses	$(1^m, (2^m, 2^0), (3^0, 4^0), x, x), (1^m, (2^m, 2^0), (3^0, 8^0), x, x),$ $(1^m, (2^m, 2^0), (4^0, 5^0), x, x), (1^m, (3^m, 2^0), (3^0, 4^0), z, y),$ $(5^m, (7^m, 2^0), (3^0, 4^0), y, x), (2^0, (8^0, 4^m), (6^m, 7^m), x, y), \ldots$

replaced by the variable x in communication statement $y = unpack()$ (node 5^m). The received value is written in the same variable received previously (variable x). Some test inputs, such as $\{x = 1, y = 2, z = 1)\}$, do not reveal this fault. However, this fault can be revealed when we apply, for example, the all-defs-*s* criterion. The test input $\{x = 2, y = 8, z = 4\}$, which covers the association $(5^m, (7^m, 2^2), 5^2, y, y)$, reveals this fault.

Krawczyk and Wiszniewski [23] present two kinds of faults related to parallel programs: observability and locking faults. The observability fault is a special kind of domain fault, related to synchronization faults. These faults can be observed or not during the execution of a same test input; the observation depends on the parallel environment and on the execution time (non-determinism). Locking faults occur when the parallel program does not finish its execution, staying locked, waiting forever. To illustrate this fault, consider again the execution of the program GCD with the test input $\{x = 7, y = 14, z = 28\}$. The expected output is (7) and the expected matching points between *send–receive* pairs are $(2^m, 2^0)$, $(3^m, 2^1)$, $(8^0, 4^m)$ or $(8^0, 5^m)$, $(8^1, 5^m)$ or $(8^1, 4^m)$, $(7^m, 2^2)$, $(8^2, 8^m)$. It is important to point out that nodes 4^m and 5^m have non-deterministic *receive* primitives (Section 2.3).

Without loss of generality, let us consider that the matching points reached are $(8^0, 4^m)$ and $(8^1, 5^m)$. Suppose that in node 5^m the statement *pvm_recv*() has been mistakenly changed to *pvm_nrecv*(), a non-blocking primitive. In this case, the message sent by slave p^1 may be not reached by non-blocking *receive* in node 5^m, before the execution of this node. This is a synchronization fault. Thus, variable y is not updated with the value sent from slave p^1. This fact could appear irrelevant here, since the value of y (14) is equal to the value that must be received

DOI: 10.1002/cpe

from p^1. However, this fault makes the node 8^m to receive the message from 8^1 instead of the message from 8^2. This fault can be revealed by the all-s-uses criterion. To cover the s-use association $(6^2, (8^2, 8^m), x)$, the tester has to provide a test input that executes the slave process p^2, for instance, $\{x = 3, y = 9, z = 4\}$. The expected output (1) is obtained, but the s-use association is not covered (due to the fault related to the non-blocking *receive*). This test case did not reveal the fault, but it indicated an unexpected path. The tester must try to select a test input that covers the s-use association. The test input $\{x = 7, y = 14, z = 28\}$ covers the association and also produces an unexpected output. The tester can conclude that the program has a fault. ValiPar (discussed in Section 3) provides support in this case, allowing the analysis of the execution trace. By analyzing the execution trace, the tester can observe that a wrong matching point was reached.

This fault is related to non-determinism and the occurrence of the illustrated matching points is not guaranteed. For example, if the slave process p^1 is fast enough to execute, the sent message reaches the node 5^m and the fault will not be observed. Notwithstanding, the synchronizations illustrated previously are more probable, considering the order of the processes creation.

A special type of the locking error is deadlock [25], a classical problem in parallel programs. Ideally, it must be detected before the parallel program execution. It is not the focus of the testing criteria proposed in this work; nonetheless, the information extracted from the parallel programs during the application of the coverage criteria may be used to statically detect deadlock situations.

3. ValiPar TESTING TOOL

To support the effective application of the testing criteria defined in the previous section, we have implemented ValiPar. ValiPar works with the concept of test sessions, which can be set up to test a given parallel program and allows one to stop testing activity and resume it later. Basically, the tool provides functionalities to (i) create test sessions, (ii) save and execute test data and (iii) evaluate the testing coverage w.r.t. a given testing criterion.

The implementation of the tool follows the architecture shown in Figure 5. This architecture was also described in [26]. ValiPar has four main modules: *ValiInst* performs all static analysis of parallel program; *ValiElem* generates the list of required elements; *ValiEval* performs test case evaluation (coverage computation); and *ValiExec* involves the parallel program execution (virtual machine creation) and generation of the executed paths.

ValiPar is able to validate parallel programs in different message-passing environments with a fixed number of processes. It is currently instanced for PVM and MPI parallel programs in C language. To adapt this tool for another message-passing environment or programming language, it is required to instance the modules *ValiInst* and *ValiExec*.

3.1. ValiInst

The ValiInst module is responsible for extracting flow information of the parallel program and for instrumenting the program with statements that will register the actual paths of execution. These tasks are accomplished mostly using the `idelgen` system, which is a compiler for the IDeL language (*Instrumentation Description Language*) [27]. IDeL is a meta-language that can

DOI: 10.1002/cpe

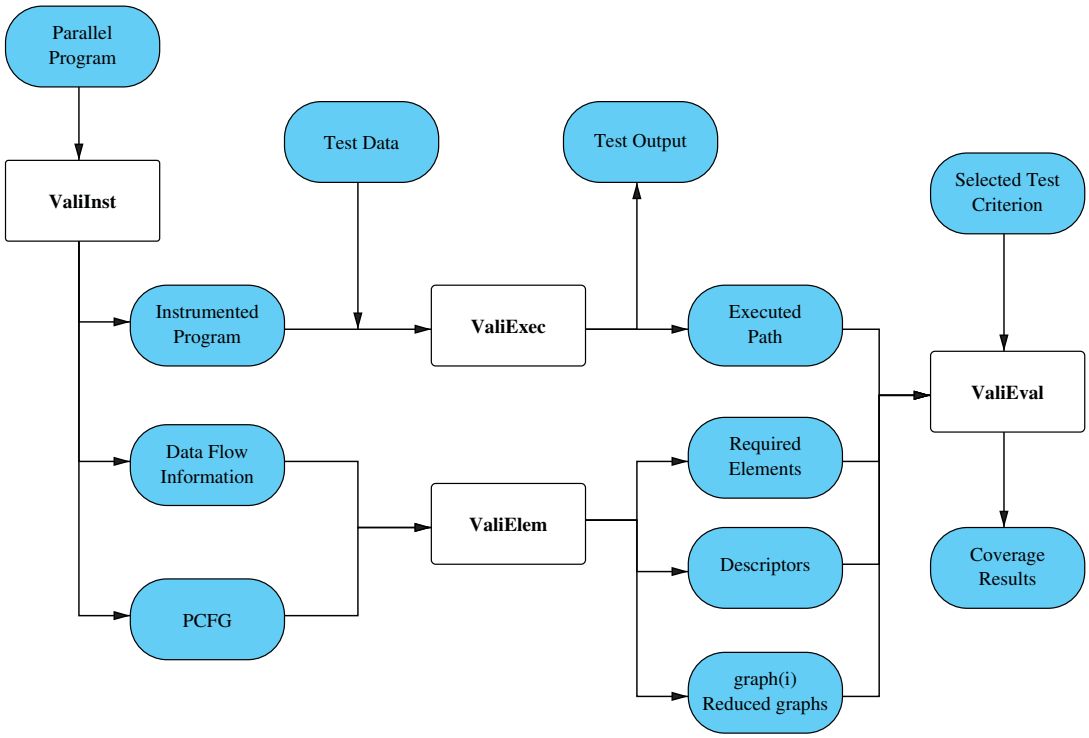


Figure 5. ValiPar tool architecture.

be instanced for different languages. In the context of this work, the instantiation of IDeL for C language was used and it was extended to treat specific aspects of PVM and MPI.

The *PCFG* is generated with information about nodes, edges, definitions and uses of variables in the nodes, as well as the presence of *send* and *receive* primitives[‡]. In this version of ValiPar the primitives for collective communication were not implemented. They need to be mapped in terms of *send* and *receive* basics.

This information set is generated for each process. The `idelgen` accomplishes the syntactic and semantic analysis of the program, according to the grammar of a given language, extracting the necessary information for instrumentation. The instrumented program is obtained by inserting check-point statements in the program being tested. These statements do not change the program semantics. They only write necessary information in a trace file, by registering the node and the process identifier in the *send* and *receive* commands. The instrumented program will produce the paths executed in each process, as well as the synchronization sequence produced within a test case.

[‡]The following primitives were considered. For MPI: *MPI_send()*, *MPI_Isend()*, *MPI_recv()* and *MPI_Irecv()*; for PVM: *pvm_send()*, *pvm_recv()* and *pvm_nrecv()*.

DOI: 10.1002/cpe

3.2. ValiElem

The ValiElem module generates the required elements by the coverage testing criteria defined in this paper. These elements are generated from *PCFGs* and data-flow information, generated by ValiInst. For that purpose, two other graphs are used: the heirs reduced graph, proposed by Chusho [28], and the graph(i), used by the testing tool Poketool [29].

In a reduced graph of heirs, all the branches are primitive. The algorithm is based on the fact that there are edges inside a *PCFG*, which are always executed when another one is executed. If each complete path that includes the edge a also includes the edge b, then b is called heir of a, and a is called ancestral of b, because b inherits information about execution of a. In other words, an edge that is always executed when another one is executed is called heir edge. An edge is called primitive, if it is not heir of any other one. ValiPar adapted the algorithm for the parallel programs context. The concept of synchronization edge was included to the concept of primitive edge. Minimizing the number of edges required by ValiPar is possible by the use of both concepts.

A graph(i) is built for each node that contains a variable definition. The purpose of this is to obtain all definition-clear paths w.r.t. a variable $x \in def(n_i^p)$. Hence, a given node k will belong to a graph(i) if at least one path from i to k exists and this path does not redefine at least one variable $\mathtt{x}$, defined in i. A node k can generate several different images in the graph because just one graph(i) is built for all defined variables in node i. However, the paths in the graph(i) are simple. To do this and to avoid infinite paths, caused by the existence of loops in the *CFG*, in the same path of the graph(i) only a node can contain more than one image, and its image is the last node of the path. The graph(i) is used to establish associations between definitions and uses of variables, generating the elements required by the data-flow testing criteria introduced in Section 2.

For each required element, ValiElem also produces a descriptor, which is a regular expression that describes a path that exercises a required element. For instance, the descriptor for the elements required by all-nodes criterion is given by the expression:

$$N * n_i^p N *$$

where N is the set of nodes in CFG^p. A required node n_i^p will be exercised by the path π^p, if π^p includes n_i. In the same way, a regular expression is defined for each element required by all testing criteria.

The descriptor describes all the paths in the graph that exercise the corresponding element and is used by ValiEval module. Figure 6 shows the required elements generated for the all-edges-s criterion, considering the program in Figure 3

Note that, in this section, we follow the notation that is adopted in the tool. For instance, `2–0` means node 2 in process 0. Moreover, the master process is always represented by process 0 and the slave processes are appropriately named 1, 2, 3, ... and so on.

3.3. ValiExec

ValiExec executes the instrumented program with the test data provided by the user. A script is used to initialize the message-passing environment before parallel program execution. ValiExec stores the test case, the execution parameters and the respective execution trace. The execution

DOI: 10.1002/cpe

1) 2–0 2–1	7) 7–0 2–1	13) 8–1 2–2	19) 8–2 2–3
2) 2–0 2–2	8) 7–0 2–2	14) 8–1 2–3	20) 8–3 4–0
3) 2–0 2–3	9) 7–0 2–3	15) 8–2 4–0	21) 8–3 5–0
4) 3–0 2–1	10) 8–1 4–0	16) 8–2 5–0	22) 8–3 8–0
5) 3–0 2–2	11) 8–1 5–0	17) 8–2 8–0	23) 8–3 2–1
6) 3–0 2–3	12) 8–1 8–0	18) 8–2 2–1	24) 8–3 2–2

Figure 6. Required elements of all-edges-s criterion.

```
tracePO :
1–0  2–0  3–0  4–0  8–1  4–0  5–0  8–2  5–0  6–0  9–0
  10–0

tracePl :
1–1  2–1  2–0  2–1  3–1  4–1  5–1  3–1  4–1  5–1  3–1
  7–1  8–1  9–1

tracePZ :
1–2  2–2  3–0  2–2  3–2  4–2  5–2  3–2  4–2  6–2  3–2
  4–2  5–2  3–2  7–2  8–2  9–2

tracePS :
```

Figure 7. Trace file.

trace includes the executed path of each parallel process, as well as the synchronization sequences. It will be used by ValiEval to determine the elements that were covered.

After the execution, the tester can visualize the outputs and the execution trace to determine whether the obtained output is the same as that expected. If it is not, a fault was identified and may be corrected before continuing the test.

A trace of a parallel process is represented by a sequence of nodes executed in this process. A synchronization from n_i^a to m_j^b is represented at the trace of the sender process of the message by the sequence $n_{i-1}^a\ n_i^a\ m_j^b\ n_i^a\ n_{i+1}^a$. Note that process a is unable to know to which node j of process b the message was sent. The same synchronization is represented at the trace of the receiver process by the sequence $m_{j-1}^b\ m_j^b\ n_i^a\ m_j^b\ m_{j+1}^b$. In this way, it is possible to determine whether the inter-processes edge (n_i^a, m_j^b) was covered. The produced traces are used to evaluate the test cases and they provide a way for debugging the program. To illustrate, Figure 7 shows the traces generated for GCD program, executed with the test input: $\{x = 1, y = 3, z = 5\}$. For this test, process 3 was not executed.

ValiExec also enables the controlled execution of the parallel program under test. This feature is useful for replaying the test activity. Controlled execution guarantees that two executions of the parallel program with the same input will produce the same paths and the same synchronization sequences. The implementation of controlled execution is based on the work of Carver and Tai [15], adapted to message-passing programs. Synchronization sequences of each process are gathered in runtime by the instrumented check-points of blocking and non-blocking *sends* and *receives*. The latter is also subject to non-determinism; hence, each request is associated with the number of times it has been evaluated. This information and other program inputs are used to achieve the deterministic execution and, thus, to allow test case replay.

DOI: 10.1002/cpe

3.4. ValiEval

ValiEval evaluates the coverage obtained by a test case set w.r.t. a given criterion. ValiEval uses the descriptors, the required elements generated by ValiElem and the paths executed by the test cases to verify which elements required for a given testing criterion are exercised. The module implements the automata associated with the descriptors. Thus, a required element is covered if an executed path is recognized by its corresponding automaton. The coverage score (percentage of covered elements) and the list of covered and not covered elements for the selected test criterion is provided as output. Figure 8 shows this information considering the all-edges-*s* criterion and the GCD program (Figure 3). These results were generated after the execution of test inputs in Figure 9.

```
Required elements not covered - criterion all_edges_s:
Required element 2 not covered: 2) 2-0 2-2
Required element 3 not covered: 3) 2-0 2-3
Required element 4 not covered: 4) 3-0 2-1
Required element 6 not covered: 6) 3-0 2-3
Required element 8 not covered: 8) 7-0 2-2
Required element 9 not covered: 9) 7-0 2-3
Required element 11 not covered: 11) 8-1 4-0
Required element 13 not covered: 13) 8-1 2-2
Required element 14 not covered: 14) 8-1 2-3
Required element 15 not covered: 15) 8-2 4-0
Required element 17 not covered: 17) 8-2 8-0
Required element 18 not covered: 18) 8-2 2-1
Required element 19 not covered: 19) 8-2 2-3
Required element 20 not covered: 20) 8-3 5-0
Required element 21 not covered: 21) 8-3 8-0
Required element 23 not covered: 23) 8-3 2-1
Required element 24 not covered: 24) 8-3 2-2

Coverage: 29.17%
```

Figure 8. Informations about coverage of the all-edges-s criterion.

```
input1.tes:
1 3 5
output1.tes:
1
input2.tes:
2 8 4
output2.tes:
2
input3.tes:
5 1 2
output3.tes:
1
input4.tes:
4 4 4
output3.tes:
4
```

Figure 9. Test cases executed for GCD program.

DOI: 10.1002/cpe

3.5. Testing procedures with ValiPar

ValiPar tool and proposed criteria can be applied following two basic procedures: (1) to guide the selection of test cases to the program and (2) to evaluate the test set quality, in terms of code and communication coverage.

1. *Test data selection with ValiPar*: Suppose that the tester uses ValiPar for supporting the test data selection. For this, the following steps must be conducted:

(a) Choose a testing criterion to guide the test data selection.
(b) Identify test data that exercise the elements required by the testing criterion.
(c) For each test case, analyze if the output is correct; otherwise, the program must be corrected.
(d) While uncovered required elements exist, identify new test cases that exercise each one of them.
(e) The tester proceeds with this method until the desired coverage is obtained (ideally 100%). In addition, other testing criteria may be selected to improve the quality of the generated test cases.

In some cases, the existence of infeasible elements does not allow a 100% coverage of a criterion. The determination of infeasible elements is an undecidable problem [22]. Because of this, the tester has to manually determine the infeasibility of the paths and required elements.

2. *Test data evaluation with ValiPar*: Suppose that the tester has a test set T and wishes to know how good it is, considering a particular testing criterion. Another possible scenario is that the tester wishes to compare two test sets T_1 and T_2. The coverage w.r.t. a testing criterion can be used in both cases. The tester can use ValiPar in the following way:

(a) Execute the program with all test cases of T (or T_1 and T_2) to generate the execution traces or executed paths.
(b) Select a testing criterion and evaluate the coverage of T (or the coverage of T_1 and T_2).
(c) If the coverage obtained is not the expected, the tester can improve this coverage by generating new test data.
(d) To compare sets T_1 and T_2, the tester can proceed as before, creating a test session for each test set and then comparing the coverage obtained. The greater the coverage obtained, the better the test set.

Note that these procedures are not exclusive. If an *ad hoc* test set is available, it can be evaluated according to Procedure 2. If the obtained coverage is not adequate, this set can be improved by using Procedure 1. The use of such an initial test set allows effort reduction in the application of the criteria. In this way, our criteria can be considered complementary to *ad hoc* approaches. They can improve the efficacy of the test cases generated by *ad hoc* strategies and offer a coverage measure to evaluate them. This measure can be used to know whether a program has been tested enough and to stop testing.

4. APPLICATION OF TESTING CRITERIA

In this section, we present the results of the application of the criteria for message-passing parallel programs. The objective is to evaluate the proposed criteria costs in terms of the test set sizes and

DOI: 10.1002/cpe

number of required elements. Although this issue would need a broader range of studies to achieve statistically significant results, the current work provides evidences of the applicability of the testing criteria proposed herein.

Five programs implemented in MPI were used: (1) *gcd*, which calculates the greatest common divisor of three numbers (example used in Figure 4); (2) *phil*, which implements the dining philosophers problem (five philosophers); (3) *prod-cons*, which implements a multiple-producer single-consumer problem; (4) *matrix*, which implements multiplication of matrix; (5) *jacobi*, which implements the iterative method of the Gauss–Jacobi for solving a linear system of equations. These programs represent concurrent-programming classical problems. Table IV shows the complexity of the programs, in terms of the number of parallel processes and the number of *receive* and *send* commands.

For each program, an initial test set (T_i) was randomly generated. Then, T_i was submitted to ValiPar (version MPI) and an initial coverage was obtained for all the criteria. After this, additional test cases (T_a) were generated to cover the elements required by each criterion and not covered by T_i. The final coverage was then obtained. In this step, the infeasible elements were detected with support of the controlled execution. Table V presents the number of covered and infeasible elements for the testing criteria. The adequate set was obtained from $T_i \cup T_a$ by taking only the test cases that really contributed to cover elements in the executed order. The size of the adequate sets is presented in Table VI.

Table IV. Characteristics of the case studies.

Programs	Processes	Sends	Receives
gcd	4	7	7
phil	6	36	11
prod-cons	4	3	2
matrix	4	36	36
jacobi	4	23	31

Table V. Number of covered and infeasible elements for the case studies.

	Covered elements/infeasible elements				
Testing criteria	gcd	phil	prod-cons	matrix	jacobi
all-nodes	62/0	176/0	60/0	368/200	499/19
all-nodes-*r*	7/0	11/0	2/0	36/15	31/2
all-nodes-*s*	7/0	36/0	3/0	36/21	23/2
all-edges	41/20	356/280	21/0	1032/982	652/499
all-edges-*s*	30/20	325/280	6/0	972/945	531/492
all-*c*-uses	29/0	50/0	43/2	572/337	608/77
all-*p*-uses	40/0	148/27	42/2	304/206	514/118
all-*s*-uses	66/47	335/280	6/0	1404/1375	768/729

DOI: 10.1002/cpe

Table VI. Size of effective test case sets.

	Size of adequate test sets				
Testing criteria	gcd	phil	prod-cons	matrix	jacobi
all-nodes	6	2	2	2	7
all-nodes-*r*	2	1	2	1	3
all-nodes-*s*	2	2	1	1	3
all-edges	3	2	2	2	7
all-edges-*s*	3	2	2	1	3
all-*c*-uses	6	2	2	2	9
all-*p*-uses	9	4	3	2	9
all-*s*-uses	10	2	2	3	6

By analyzing the results, we observe that the criteria are applicable. In spite of the great number of required elements for the programs *phil*, *matrix* and *jacobi*, the number of test cases does not grow proportionally. The size of the adequate test sets is small.

In fact, some effort is necessary to identify infeasible elements. In this study, the controlled execution was used to aid in the identification of the infeasible elements. A good strategy is to analyze the required elements to decide infeasibility only when the addition of new test cases does not contribute to improve coverage. In this case, paths are identified to cover the remaining elements and, if possible, specific test cases are generated. Other strategy is to use infeasible patterns for classification of the paths. Infeasible patterns are structures composed of sequence of nodes with inconsistent conditions [30]. The use of patterns is an important mechanism to identify infeasibility in traditional programs. If a path contains such patterns it will be infeasible. In order to reduce the problem of infeasible paths, we intend to implement in ValiPar a mechanism for automatically discarding infeasible paths according to a pattern provided by the tester.

We observed, in the results of the experiment, that many infeasible elements are related to the *s*-uses (*all-edges-s* and *all-s-uses* criteria). This situation occurs because we adopted a conservative position by generating all the possible inter-processes edges, even when the communication may not be possible in the practice. This was adopted with the objective of revealing faults related to missing communications. We are now implementing a mechanism to disable the generation of all the combinations, if desired by the tester. Another idea is to generate all possible communication uses (*s*-uses) during the static analysis and, during the program execution, to obtain which *s*-uses tried to synchronize (race situation). These *s*-uses that participate in the race have high probability of being feasible; otherwise, *s*-uses have major probability of being infeasible. This investigation is inspired on the work of Damodaran-Kamal and Francioni [16].

5. RELATED WORK

Motivated by the fact that traditional testing techniques are not adequate for testing features of concurrent/parallel programming, such as non-determinism and concurrency, many researchers have developed specific testing techniques addressing these issues.

DOI: 10.1002/cpe

Lei and Carver [14] present a method that guarantees that every partially ordered synchronization will be exercised exactly once without saving any sequences that have already been exercised. The method is based on the reachability testing. By definition, the approach avoids generation of unreachable testing requirements. Their method is complementary to our approach. On the one hand, the authors employ a reachability schema to calculate the synchronization sequence automatically. They do not address how to select the test case which will be used for the first run. On the other hand, we use the static analysis of the program to indicate the test cases that are worth selecting. Therefore, the coverage metrics we proposed can be used to derive the test case suite that will be input to the reachability-based testing, as argued by the authors.

Wong *et al.* [31] propose a set of methods to generate test sequences for structural testing of concurrent programs. The reachability graph is used to represent the concurrent program and to select test sequences to the all-node and all-edge criteria. The methods aim the generation of a small test sequences set that covers all the nodes and the edges in a reachability graph. For this, the methods provide information about which parts of the program should be covered first to effectively increase the coverage of these criteria. The authors stress that the major advantage of the reachability graph is that only feasible paths are generated. However, the authors do not explain how to generate the reachability graph from the concurrent program or how to deal with the state space explosion.

Yang and Chung [11] introduce the path analysis testing of concurrent programs. Given a program, two models are proposed: (1) *task flow graph*, which corresponds to the syntactical view of the task execution behavior and models the task control flow, and (2) *rendezvous graph*, which corresponds to the runtime view and models the possible rendezvous sequences among tasks. An execution of the program will traverse one concurrent path of the rendezvous graph (*C-route*) and one concurrent path of the flow graph (*C-path*). A method called *controlled execution* to support the debugging activity of concurrent programs is presented. They pointed out three research issues to be addressed to make their approach practical: *C*-path selection, test generation and test execution.

Taylor *et al.* [8] propose a set of structural coverage criteria for concurrent programs based on the notion of concurrent states and on the concurrency graph. Five criteria are defined: *all-concurrency-paths*, *all-proper-cc-histories*, *all-edges-between-cc-states*, *all-cc-states* and *all-possible-rendezvous*. The hierarchy (subsumption relation) among these criteria is analyzed. They stress that every approach based on reachability analysis would be limited in practice by state space explosion. They mentioned some alternatives to overcome the associated constraints.

In the same vein of Taylor and colleagues' work, Chung *et al.* [6] propose four testing criteria for Ada programs: *all-entry-call*, *all-possible-entry-acceptance*, *all-entry-call-permutation* and *all-entry-call-dependency-permutation*. These criteria focus the rendezvous among tasks. They also present the hierarchy among these criteria.

Edelstein *et al.* [12,13] present a multi-threaded bug detection architecture called *ConTest* for Java programs. This architecture combines a replay algorithm with a seeding technique, where the coverage is specific to race conditions. The seeding technique seeds the program with *sleep* statements at shared memory access and synchronization events and heuristics are used to decide when a *sleep* statement must be activated. The replay algorithm is used to re-execute a test when race conditions are detected, ensuring that all accesses in race will be executed. The focus of the work is the non-determinism problem, not dealing with code coverage and testing criteria.

DOI: 10.1002/cpe

Yang *et al.* [9,10] extend the data-flow criteria [4] to shared memory parallel programs. The parallel program model used consists of multiple threads of control that can be executed simultaneously. A *parallel program-flow graph* is constructed and is traversed to obtain the paths, variable definitions and uses. All paths that have definition and use of variables related with parallelism of threads constitute test requirements to be exercised. The *Della Pasta Tool* (*Delaware Parallel Software Testing Aid*) automates their approach. The authors presented the foundations and theoretical results for structural testing of parallel programs, with definition of the all-du-path and all-uses criteria for shared memory programs. This work inspired the test model definition for message-passing parallel programs, described in Section 2.

The previous works stress the relevance of providing coverage measures for concurrent and parallel programs, considering essentially shared memory parallel programs. They do not address coverage criteria that consider the main features of the message-passing programs. Our work is based on the works mentioned above, but differently we explore control and data-flow concepts to introduce criteria specific for the message-passing environment paradigm and describe a supporting tool.

A related, but orthogonal, approach to testing is the use of model checking methods to provide evidences of the correctness of an algorithm, by suitably exploring the state space of all possible executions [32]. Improvements in model checking theory and algorithms allow handling huge state space. When effectively done, model checking can provide a slightly stronger assertion on the correctness of parallel programs than testing with selected test cases. There exist some initiatives of model checking of parallel programs [33–36]. These approaches suffer from several drawbacks, though. Firstly, the program cannot usually be model-checked directly, requiring instead the conversion into a suitable model. This conversion is rarely automated and must be made manually [36]. However, in this case, it is the correction of the model that is analyzed, not of the actual program. It remains to be demonstrated that the model correctly represents the program. Sometimes, the model is difficult to obtain, since important primitives of parallel program may not be directly represented in the model. This problem has been recently tackled in [34], where an extension to the model checker SPIN, called MPI-SPIN, is proposed. Although the gap between the program and the model is reduced, a direct translation is far from being feasible. Another drawback of model checking is the awkward handling of user inputs. There exist some approaches that use symbolic execution in order to represent all possible user inputs symbolically, e.g. [33]. Nonetheless, symbolic execution is a long-term research topic and brings its own problems, since the expression obtained along the paths grows intractable. Then, even if model checking is used in the verification of some model of the program, the testing of the program is still important, and the problem of measuring the quality of the test cases used to test the program still remains.

In relation to parallel testing tools, most tools available aid only the simulation, visualization and debugging; they do not support the application of testing criteria. Examples of these tools are TDC Ada [20] and ConAn [19], respectively, for ADA and Java. For message-passing environments, we can mention Xab [17], Visit [18] and MDB [16] for PVM, and XMPI [37] and Umpire [21] for MPI.

When we consider testing criteria support, we can mention the tool Della Pasta [9], based on threads, and the tool STEPS [38]. This last one works with PVM programs and generates paths to cover some elements in the control-flow graphs of PVM programs. We could not find in the

DOI: 10.1002/cpe

Table VII. Existent testing tools.

Tool	Data flow	Control flow	Test replay	Debug	Language
TDC Ada			√		Ada
ConAn			√		Java
Della Pasta	√		√	√	C
Xab				√	PVM
Visit				√	PVM
MDB			√	√	PVM
STEPS		√	√	√	PVM
Astral		√		√	PVM
XMPI				√	MPI
Umpire				√	MPI
ValiPar	√	√	√	√	PVM and MPI

literature a tool, which implements criteria based on control, data and communication flows, as the one presented in this paper. Table VII shows the main facilities of ValiPar, compared with the existing tools.

6. CONCLUDING REMARKS

Testing parallel programs is not a trivial task. As mentioned previously, to perform this activity some problems need to be investigated. This paper contributes in this direction by addressing some of them in the context of message-passing programs: definition of a model to capture relevant control and data-flow information and to statically generate the corresponding graph; proposition of specific testing coverage criteria; development of a tool to support the proposed criteria, as well as, sequential testing; implementation of mechanisms to reproduce a test execution and to force the execution of a given path in the presence of non-determinism; and evaluation of the criteria and investigation of the applicability of the criteria.

The proposed testing criteria are based on models of control and data flows and include the main features of the most used message-passing environments. The model considers communication, concurrency and synchronization faults between parallel processes and also fault related to sequential aspects of each process.

The use of the proposed criteria contributes to improve the quality of the test cases. The criteria offer a coverage measure that can be used in two testing procedures. The first one for the generation of test cases, where these criteria can be used as guideline for test data selection. The second one is related to the evaluation of a test set. The criteria can be used to determine when the testing activity can be ended and also to compare test sets. This work also showed that the testing criteria can contribute to reveal important faults related with parallel programs.

The paper described ValiPar, a tool that supports the proposed criteria. ValiPar is independent of the message-passing environment and is currently configured for PVM (ValiPVM) and MPI (ValiMPI). These versions are configured for language C. We intend to configure other versions of ValiPar, considering others languages used for message-passing parallel programs, e.g. Fortran.

DOI: 10.1002/cpe

Non-determinism is very common in parallel programs and causes problems for validation activity. To minimize these problems, we implemented in ValiPar mechanisms to permit controlled execution of parallel programs. With these mechanisms, synchronization sequences can be re-executed, repeating the test and, thus, contributing for the revalidation and regression testing of the parallel programs.

Using the MPI version of ValiPar, we carried out a case study that showed the applicability of the proposed criteria. The results showed a great number of required elements mainly for the communication-flow-based criteria. This should be evaluated in future experiments and some refinements may be proposed to the criteria. We intend to conduct other experiments to explore efficacy aspects to propose changes in the way of generating the required elements and to avoid a large number of infeasible ones.

The advantage of our coverage criteria, comparing with another techniques for testing parallel programs, is to systematize the testing activity. In fact, there exists an amount of cost and time associated with the application of the coverage criteria. However, the criteria provide a coverage measure that can be used to assess the quality of the tests conducted. In the case of critical applications, this evaluation is fundamental. In addition, ValiPar reduces this cost, by automating most of the activities related on parallel program testing.

The evolution of our work on this subject is directed to several lines of research: (1) development of experiments to refine and evaluate the testing criteria; (2) use of ValiPar for real and more complex parallel programs; (3) implementation of mechanisms to validate parallel programs that dynamically create processes and other ones to help the tester in identifying infeasible elements; (4) conduction of an experiment to evaluate the efficacy of the generated test data against *ad hoc* test sets; and (5) definition of a strategy that synergistically combines model checking methods and the testing criteria.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous referees for their helpful comments and Felipe Santos Sarmanho for his assistance in the experiments. This work was supported by Brazilian funding agency CNPq, under Process number 552213/2002-0.

REFERENCES

1. Almasi GS, Gottlieb A. *Highly Parallel Computing* (2nd edn). The Benjamin Cummings Publishing Company: Menlo Park, CA, 1994.
2. Geist GA, Kohl JA, Papadopoulos PM, Scott SL. Beyond PVM 3.4: What we've learned what's next, and why. *Fourth European PVM–MPI Conference—Euro PVM/MPI'97*, Cracow, Poland, 1997; 116–126.
3. Snir M, Otto S, Steven H, Walker D, Dongarra J. MPI: The complete reference. *Technical Report*, MIT Press, Cambridge, MA, 1996.
4. Rapps S, Weyuker EJ. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering* 1985; **11**(4):367–375.
5. Yang C-SD. Program-based, structural testing of shared memory parallel programs. *PhD Thesis*, University of Delaware, 1999.
6. Chung C-M, Shih TK, Wang Y-H, Lin W-C, Kou Y-F. Task decomposition testing and metrics for concurrent programs. *Fifth International Symposium on Software Reliability Engineering* (*ISSRE'96*), 1996; 122–130.
7. Koppol PV, Carver RH, Tai K-C. Incremental integration testing of concurrent programs. *IEEE Transactions on Software Engineering* 2002; **28**(6):607–623.

DOI: 10.1002/cpe

8. Taylor RN, Levine DL, Kelly C. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering* 1992; **18**(3):206–215.
9. Yang C-S, Souter AL, Pollock LL. All-du-path coverage for parallel programs. *International Symposium on Software Testing and Analysis* (*ISSTA'98*), *ACM-Software Engineering Notes*, 1998; 153–162.
10. Yang C-SD, Pollock LL. All-uses testing of shared memory parallel programs. *Software Testing*, *Verification and Reliability* (*STVR*) 2003; **13**(1):3–24.
11. Yang R-D, Chung C-G. Path analysis testing of concurrent programs. *Information and Software Technology* 1992; **34**(1).
12. Edelstein O, Farchi E, Goldin E, Nir Y, Ratsaby G, Ur S. Framework for testing multi-threaded java programs. *Concurrency and Computation*: *Practice and Experience* 2003; **15**(3–5):485–499.
13. Edelstein O, Farchi E, Nir Y, Ratsaby G, Ur S. Multithreaded java program test generation. *IBM System Journal* 2002; **41**(1):111–125.
14. Lei Y, Carver RH. Reachability testing of concurrent programs. *IEEE Transactions on Software Engineering* 2006; **32**(6):382–403.
15. Carver RH, Tai K-C. Replay sand testing for concurrent programs. *IEEE Software* 1991; 66–74.
16. Damodaran-Kamal SK, Francioni JM. Nondeterminacy: Testing and debugging in message passing parallel programs. *Third ACM/ONR Workshop on Parallel and Distributed Debugging*. ACM Press: New York, 1993; 118–128.
17. Beguelin AL. XAB: A tool for monitoring PVM programs. *Workshop on Heterogeneous Processing—WHP03*. IEEE Press: New York, April 1993; 92–97.
18. Ilmberger H, Thürmel S, Wiedemann CP. Visit: A visualization and control environment for parallel program debugging. *Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, May 1993; 199–201.
19. Long B, Hoffman D, Strooper P. Tool support for testing concurrent java components. *IEEE Transactions on Software Engineering* 2003; **29**(6):555–565.
20. Tai KX, Carver RH, Obaid EE. Debugging concurrent ada programs by deterministic execution. *IEEE Transactions on Software Engineering* 1991; **17**(1):45–63.
21. Vetter JS, Supinski BR. Dynamic software testing of MPI applications with Umpire. *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE Press (electronic publication): New York, 2000.
22. Frankl FG, Weyuker EJ. Data flow testing in the presence of unexecutable paths. *Workshop on Software Testing*, Banff, Canada, July 1986; 4–13.
23. Krawczyk H, Wiszniewski B. Classification of software defects in parallel programs. *Technical Report 2*, Faculty of Electronics, Technical University of Gdansk, Poland, 1994.
24. Howden WE. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering* 1976; **2**:208–215.
25. Tanenbaum AS. *Modern Operating Systems* (2nd edn). Prentice-Hall: Englewood Cliffs, NJ, 2001.
26. Souza SRS, Vergilio SR, Souza PSL, Simão AS, Bliscosque TG, Lima AM, Hausen AC. Valipar: A testing tool for message-passing parallel programs. *International Conference on Software Knowledge and Software Engineering* (*SEKE05*), Taipei, Taiwan, 2005; 386–391.
27. Simão AS, Vincenzi AMR, Maldonado JC, Santana ACL. A language for the description of program instrumentation and the automatic generation of instrumenters. *CLEI Electronic Journal* 2003; **6**(1).
28. Chusho T. Test data selection and quality estimation based on concept of essential branches for path testing. *IEEE Transactions on Software Engineering* 1987; **13**(5):509–517.
29. Chaim MJ. Poketool—uma ferramenta para suporte ao teste estrutural de programas baseados em anlise de fluxo de dados. *Master's Thesis*, DCA/FEE/UNICAMP, Campinas, SP, 1991.
30. Vergilio SR, Maldonado JC, Jino M. Infeasible paths in the context of data flow based testing criteria: Identification, classification and prediction. *Journal of the Brazilian Computer Society* 2006; **12**(1).
31. Wong WE, Lei Y, Ma X. Effective generation of test sequences for structural testing of concurrent programs. *Tenth IEEE International Conference on Engineering of Complex Systems* (*ICECCS'05*), 2005; 539–548.
32. Clarke EM, Emerson EA, Sistla AP. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems* 1986; **8**(2):244–263.
33. Siegel SF, Mironova A, Avrunin GS, Clarke LA. Using model checking with symbolic execution to verify parallel numerical program. *International Symposium on Software Testing and Analysis*, 2006; 157–167.
34. Siegel SF. Model checking nonblocking MPI programs. *Verification*, *Model Checking and Abstract Interpretation* (*Lecture Notes in Computer Science*, vol. 4349). Springer: Berlin, 2007; 44–58.
35. Matlin OS, Lusk E, McCune W. Spinning parallel systems software. *SPIN* (*Lecture Notes in Computer Science*, vol. 2318). Springer: Berlin, 2002; 213–220.
36. Pervez S, Gopalakrisman G, Kirby RM, Thakur R, Gropp W. Formal verification of programs that use mpi one-sided communication. *PVM/MPI* (*Lecture Notes in Computer Science*, vol. 4192). Springer: Berlin, 2006; 30–39.
37. The LAM/MPI Team. *XMPI*. Open Systems Laboratory, Indiana University, Bloomington, IN.
38. Krawczyk H, Kuzora P, Neyman M, Proficz J, Wiszniewski B. STEPS—A tool for testing PVM programs. *Third SEIHPC Workshop*, January 1998. Available at: http://citeseer.ist.psu.edu/357124.html.

DOI: 10.1002/cpe