ValiPar: A Testing Tool for Message-Passing Parallel Programs

Simone do Rocio Senger de Souza¹,³, Silvia Regina Vergilio², Paulo Sergio Lopes de Souza¹,³, Adenilso da Silva Simão³, Thiago Blicosque Gonçalves¹, Alexandre de Melo Lima¹, Alexandre Ceolin Hausen²

¹State University of Ponta Grossa – UEPG  
Department of Computer Science  
84.030-900 – Ponta Grossa – PR – Brazil  
{srocio, pssouza}@uepg.br  
blicosque@yahoo.com.br  
alexmlima@pop.com.br

²Federal University of Paraná – UFPR  
Department of Computer Science  
CP: 19081 – 81531-970 – Curitiba – PR – Brazil  
{silvia,ceolin}@inf.ufpr.br

³State University of São Paulo –ICMC-SCE , USP  
13560-970 – São Carlos – SP – Brazil  
adenilso@icmc.usp.br

Abstract

The software testing activity is crucial for Software Quality Assessment. To aid at this phase, several testing criteria were proposed. A testing criterion is a predicate to be satisfied by a set of test cases. It is used to guide the selection and evaluation of a test data set and offers coverage metrics that quantify the testing activity. When parallel programs are considered, features as concurrency, communication and synchronization make more complex this activity. In this context, specific criteria and supporting tools are very important. This paper presents a tool, called ValiPar, that implements testing criteria specific for parallel programs in message-passing environments. It provides a baseline to the selection and evaluation of test data. Based on the obtained coverage for a criterion, the tester can evaluate the quality of the parallel program being tested.

1. Introduction

Time prevision, dynamic molecular simulation, bioinformatics and several other problems are usually known as “Grand Challenges”, because they are very complex and have hard solution. These problems motivate the investigation, development and utilization of the high performance computing and, in this context the use of parallel programs is fundamental.

There are three basic forms to build parallel software [1]: 1) automatic environments that generate parallel code from sequential algorithms; 2) concurrent programming languages such as CSP and ADA; and 3) extensions for traditional languages, such as C and Fortran. Message-passing environments implement these extensions. These environments include a library of functions that allow 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: PVM (Parallel Virtual Machine) [7] and MPI (Message Passing Interface) [14].

Parallel programs present some features that turn more complex the testing activity, such as non-determinism, concurrency, synchronization and communication aspects. Moreover, the testing teams are usually not trained and we find a low number of adequate tools. This makes the test of parallel programs very expensive. For sequential programs, many of the testing problems were reduced with the use of testing criteria and the implementation of supporting tools. A testing criterion [11] 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, offering a coverage measure that can be used for stop testing. Structural criteria utilize the code, the implementation, and structural aspects of the program to derive test cases. They are usually based on a control-flow graph and/or definitions and uses of variables in the program [11].

In the literature, there are some works with the goal of extending test criteria for parallel programs [4, 5, 8, 15, 17, 18]. However, the practical application of a testing criterion is only possible if a tool is available. Most of the mentioned work does not address supporting tools. In addition to, in spite of the crescent use and popularization of the message passing environments, only work [16] address specific criteria for message-passing parallel programs. The existent tools [2, 3, 12] for this kind of programs do not support testing criteria; they only aid the simulation and debugging of the message-passing parallel programs. The employment of a criterion and a supporting tool are fundamental to have reliability measures and to ensure the quality of this kind of parallel software.

*This work is supported by CNPq.
To fulfill the demand for tools to support the application of testing criteria in message-passing parallel programming, this paper present ValiPar, a tool oriented to test sessions, that supports the testing criteria family, introduced in [16]. A model that includes the main features of the parallel programs (such as synchronization, communication, parallelism and concurrency) was used to define those testing criteria. ValiPar allows two basic testing procedures: selection and evaluation of test data sets. These procedures are illustrated, in this paper, using a PVM program. However, ValiPar is independent of the environment and can be configured for another message-passing environment.

The paper is organized as follows. In Section 2, the architecture of ValiPar is presented. In Section 3, the procedures for utilization of the ValiPar tool are illustrated and Section 4 contains the conclusions and also refers to future works.

2. ValiPar Tool Architecture

ValiPar tool supports the validation of parallel programs in different message passing environments, by using a set of structural testing criteria previously established [16]. These criteria were defined based on specific characteristics of message-passing parallel programs. Basically, ValiPar supplies functions to create test sessions, save and execute test data and evaluate the testing coverage with respect to a selected testing criterion. To accomplish those and other activities, the tool has four main modules that communicate through files, as showed in Figure 1. The functionality of the modules is detailed in the sequence.

2.1. IDeL

IDeL - Instrumentation Description Language, developed by Simão et al [13], is a meta-language that supports the instrumentation of programs. IDeL accomplishes a syntactic and semantic analysis of the language and extracts the necessary information for instrumentation, also generating the instrumented program. This instrumented program is obtained adding some special statements that do not change the program semantic but do register some information in a trace file. This trace file is generated during the instrumented program execution. Because IDeL is a meta-language, it can be instanced for different programming languages. In the context of this work, the IDeL version for C language was used. This version was extended to treat specific aspects of PVM and MPI, which involve communication and synchronization among parallel processes.

To accomplish the analysis of the parallel program PP, it is considered that the number $n$ of processes of PP is known, such that $PP = \{P_0, P_1, ..., P_{n-1}\}$.

A CFG – Control-Flow Graph is created for each process $P$ and, then, the graph PCFG - Parallel Control Flow Graph for $P$ is generated. In short, a CFG is composed by a set of nodes and a set of edges.

![Figure 1: Architecture of ValiPVM.](image-url)
Each node corresponds to a statement of the code and an edge links a node to another. A node can be associated to a communication function (send or receive). The communication functions are represented by the notations send(i,j,t) (respectively receive(i,j,t)) that means that the process i sends (receives) a message with tag t to (from) the process j.

PCFG contains the synchronization edges among the parallel processes of PP, making possible to extract information about the communication among these processes.

To illustrate the concepts of a PCFG, consider Figures 2 and 3. Figure 2 contains a simple program in PVM and Figure 3 presents its respective PCFG. In this program, the parent process (p^0) creates a child process (p^1) and waits for a message of the child (statement pvm_recv () in node 4). When p^1 is created, it packs the message and sends to p^0 (statement pvm_send () in node 11). Soon after sending the message, p^1 is concluded; the same happens to p^0 after receiving and printing the message.

```c
int main()
{
    int id, tid, bid;
    char msg[20];
    /\ 1/ id = pvm_parent();
    /\ 2/ if (id == PvmNoParent)
    { //
      /\ 3/ pvm_spawn("hello",(char**)0,0,"",1,&tid);
      /\ 4/ bid=pvm_recv(-1,-1);
      /\ 5/ pvm_bufinfo(bid, (int*)0, (int*)0, &tid);
      /\ 6/ pvm_upkstr(msg);
      /\ 7/ printf("from t%x: %s\n", tid, msg);
    }
    else
    {
      /\ 8/ strcpy(msg, "Hello!");
      /\ 9/ pvm_initsend(PvmDataDefault);
      /\10/ pvm_pkstr(msg);
      /\11/ pvm_send(id,1);
    }
    /\12/ pvm_exit();
    /\13/ exit(0);
}
```

**Figure 2: Hello Program in PVM.**

In this example, only a synchronization occurs, represented in PCFG by the dotted edge (11^1,4^0), which represents, respectively, a link between nodes with send and receive commands. During the test activity, this synchronization edge should be exercised to cover the synchronization among the processes. In this way, it is possible to establish testing criteria that require the execution of all existent nodes, edges or synchronization edges in the PCGF. These criteria are based on control and communication flows of the program.

IDE also generates data-flow information that is the information about definitions and uses of variables. A variable *x* is defined when a value is saved in the correspondent 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 also need to consider the communication functions, such as receive, because these functions set one or more variables with the value t received in the message. A use of *x* occurs when the value associated to *x* is referred. A use can be: 1) a computational use (or c-use), which occurs in a computation statement, related to a node in the CFG; 2) a predicate use (or p-use), which occurs in a condition (predicate) associated to control-flow statement, related to edge in the CFG; and 3) a communication use (or s-use), which occurs in a synchronization statement, (that contains passing message functions), related to a synchronization edge in the PCFG.

CFG and the data flow information are stored in a file, for each parallel program. From this file, the module ValiElem generates the associations between definitions and uses of variables, as well as the associations among the communication uses. These associations are the elements required by the data-flow based criteria supported by ValiPar. They should be exercised by the test data [11].

**Figure 3: The PCFG of Hello Program.**
2.2. ValiElem

ValiElem generates the required elements for the coverage testing criteria. These elements are generated from CFGs and data flow information, generated by Idle. For that, two other graphs are used: the heirs reduced graph, proposed by Chusho [6] and the graph(i), used by the testing tool Poketool [10].

In a reduced graph of heirs all the branches are primitive. The algorithm is based on the fact that there are edges inside a CFG that are always executed when another one is. 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. Using both concepts is possible to minimize the number of edges required by ValiPar.

A graph(i) is built for each node that contains a variable definition. A given node k will belong to a graph(i) if exists at least one path from i to k that does not redefine at least one variable x, defined in i.

Therefore, a same node, or edge, of the CFG, can create several nodes or edges in the graph(i), because is just one graph(i) is built for all defined variables in i. In that way, a node k can generate several different images in the graph(i). To avoid unending paths, caused by the existence of loops in the CFG, in a 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 by ValiElem to establish associations of definitions and uses of variables, which are elements required by data flow testing criteria introduced in [16].

ValiElem also produces descriptors for each required element. The descriptor is used in the evaluation (module ValiEval). A descriptor is given in terms of a regular expression that describes a path that exercises (or covers) a required element. For example, the descriptor of the criterion All-nodes is described by the expression:

\[ N^* \text{ni-p N} \]

where N is the set of nodes in the CFG of process p.

A required node ni-p (notation for the node ni of the process p) will be exercised (or covered) by the path p, if p includes ni.

In the same way, this regular expression is defined for each testing criteria.

2.3. ValiExec

ValiExec executes the instrumented program using the test data provided by the user. The user should provide the name of the file that contains the executable for the instrumented program. ValiExec stores the keyboard inputs, inputs parameters, test output and the respective execution trace. The execution trace includes the trace of each parallel process and is utilized during the evaluation of test cases to determinate which elements were covered. After the program execution, the tester can visualize the outputs and also the execution trace to determinate whether the obtained output is the same as the expected. If it is not, an error was identified and must be corrected before the testing activity.

2.4. ValiEval

ValiEval evaluates the coverage obtained by test sets with respect to a selected criterion, supported by ValiPar. The criteria were defined to test control, data and communication flows of the parallel program in message-passing environments. For example, it can be mentioned the All-Nodes-R criterion that requires that all the nodes which contain receive statements are exercised; All-Edges-S criterion that requires that all synchronization edges are exercised; and All-S-Uses criterion that requires that all s-uses associations are exercised.

ValiEval uses the executed paths for each test data to verify which required elements (for one testing criterion) are exercised.

3. Using ValiPar Tool – An Example

To illustrate the use of ValiPar tool, two main procedures are considered: the selection and evaluation of test data. For this purpose, the gcd PVM program [9] is used (Figures 4 and 5). This program calculates the greatest common divisor of three numbers. For this, four parallel processes are created: a master (denoted by m) and three slaves (denoted by 0, 1, 2). In the master process, after the reading of three inputs, the slaves are created and run the gcd program. Each slave waits (in the pvm_recv() statement) two values from the master and calculates the maximum divisor for the received values.

To conclude, the slaves send the calculated values to the master and terminate their executions. After the calculation, which can involve three slaves processes or only two (depending on the input values), the result is presented by the master process, which finalizes all the created processes that are still running.

3.1. Test Data Selection with ValiPar

Suppose that the tester uses ValiPar for supporting the test data selection. For this, the following steps must be carried out:

1. to choose a test criterion to guide the test data selection. Considering the All-Edges-S criterion (this
criterion requires that all synchronization edges must be exercised), the following required elements (edges) are generated by ValiElem:

(7-m,2-0), (11-m,2-1), (20-m,2-2), (12-0,12-m), (12-0,14-m), (12-0,21-m), (12-1,12-m), (12-1,14-m), (12-1,21-m), (12-2,12-m), (12-2,14-m), (12-2,21-m).

Each edge has the format: (node with sending statement – process identifier, node with receiving statement – process identifier).

2. to identify test data that exercise each one of those edges. This way, the tester can provide a test data {x=1, y=2, z=1}, obtaining as output the value 1. Coverage of 33.3% is obtained. The following edges are covered:

(7-m,2-0), (11-m,2-1), (12-0,12-m), (12-1,14-m).

3. to identify new test cases that exercise the edges that were not executed yet. For the example consider that the tester provide the test data {x=3,y=9,z=21}, obtaining as output the value 3. The coverage of All-Edges-S criterion is now 50% and, in addition, the following edges were exercised:

(20-m,2-2), (12-2,21-m).

The tester proceeds with this method until get a 100% coverage, or until obtains the desired coverage. Besides, other testing criteria can be selected to improve the quality of the generated test cases.

In some cases, the existence of infeasible elements does not allow 100% coverage of a criterion. The determination of infeasible elements is an undecidable question; there is no algorithm to determine if a path in the CFG is or not infeasible. Because of this, the tester has to manually determine the infeasibility of the paths and required elements.

3.2. Test Data Evaluation with ValiPar

Suppose that the tester has a test set T and wants to know how good it is, considering a particular testing criterion. The tester can use ValiPar in the following way:

1. to execute the program with all test cases of T to generate the execution traces.

2. to select one testing criterion and evaluate the coverage of T.

3. if the coverage obtained is not the expected, the tester can improve this coverage by performing the steps of Section 3.1 and generating new test data.

Otherwise, suppose that the tester wishes to compare two test sets T1 and T2. The coverage with respect to a testing criterion can be used in both cases. The tester can proceed as before, creating a test session for each test set and then comparing the coverage obtained. The greater the coverage the better the test set can be used by the tester to compare test data sets.
4. Concluding Remarks

This paper introduced ValiPar, a tool for validation of parallel programs. As for as we know, this is the first testing tool that implement testing criteria specific to validation of message-passing parallel programs. The main feature of the ValiPar tool is providing to the tester information on the evolution of the testing activity, through of the coverage measure.

ValiPar can be used to support the test data selection and to evaluate the quality of test sets. It implements a family of testing criteria for validation of control, data and communication flows [16]. The definition of these testing criteria was based on testing criteria for traditional programs, also considering classical errors in parallel programs: communication errors, synchronization errors and errors related with non-determinism. ValiPar helps the tester in the identification of these errors.

ValiPar is independent of the message-passing environment. Module IDEL allows configuration for different languages and environments. There are in the moment two versions of ValiPar: ValiPVM and ValiMPI. These versions are configured for language C and respectively, PVM and MPI programs. We intend to configure other versions of ValiPar for other message-passing environments, such as p4, Express, etc.

Non-determinism is very common in parallel programs and causes problems for validation activity. To minimize these problems, we are implementing in ValiPar mechanisms to permit controlled execution of parallel programs. These mechanisms will allow that synchronization sequences can be re-executed, repeating the conducted test, and, in this way, contributing for the revalidation and regression testing of the parallel programs.

The evolution of our work on this subject is directed to three lines of research: 1) the development of experiments to refine and evaluate the testing criteria; 2) the use of ValiPar for real and complex parallel programs and, 3) the implementation of mechanisms to validate parallel programs that dynamically create processes and other ones to help the tester in the identification of infeasible elements.

References