An Integrated Testing and Debugging Environment for Parallel and Distributed Programs*

João Lourenço, José C. Cunha
Departamento de Informática
Universidade Nova de Lisboa
Portugal
{jml, jcc}@di.fct.unl.pt

Henryk Krawczyk, Piotr Kuzora,
Marcin Neyman, Bogdan Wiszniewski
Technical University of Gdańsk
Poland
{hkrawk, kuzor, marcinn, bowisz}@pg.gda.pl

Abstract

To achieve a certain degree of confidence that a given program follows its specification, a testing phase must be included in the program development process, and also a complementary debugging phase, to help locating the program's bugs. This paper presents an environment which results of the composition and integration of two basic tools: STEPS (a testing tool) and DDBG (a debugging tool). The two tools are presented individually as stand-alone tools, and we describe how they were combined through the use of another intermediate tool. We claim that the result achieved is a very effective testing and debugging environment.

1 Introduction

To achieve a certain degree of confidence that a given program follows its specification, a testing phase must be included in the program development process. *Bugs* correspond to program behaviors that don't comply to the given specification, and debugging tools are used to help in its localization.

Parallel and distributed programs pose increased difficulties to the testing and debugging activities, when compared to sequential programs. This is due to several reasons, namely, the presence of multiple control and data flow paths, the interference upon program behavior caused by the monitoring and external control of its execution (the so called "*probe effect*"), and the non-determinism related to the presence of time-dependent events in a parallel and distributed program.

*In *Proceedings of EUROMICRO'97, 23rd Euromicro Conference*, Budapest, Hungary, September 1997.

Testing approaches, for example, based on systematic testing of parallel programs through path examination, play a very important role in the process of assuring final program correctness. The development of methods and tools to help the user in the identification of which such paths should be generated and tested, is a key issue in an advanced parallel software engineering environment [13].

However, such testing tools need to be integrated with debugging tools in order to allow a controlled execution of the paths under test, with the guarantee of reproducibility of the program execution, and to support program inspection and extra control during this controlled execution. We argue that a close integration of testing and debugging tools greatly contributes to a better understanding of parallel and distributed program behavior.

In this paper we discuss the design and implementation of an integrated testing and debugging environment, that is the result of the composition of two independent tools: STEPS (Structural TEsting of Parallel Software) [9], a testing tool, and DDBG (Distributed DeBuGger) [3], a debugging tool.

The paper is organized as follows: In the next section, we briefly discuss the relevant issues in testing and debugging of parallel programs that have motivated our approach. Sections 3 and 4 present the STEPS and DDBG tools, respectively. Section 5 discusses the integration of STEPS and DDBG, and presents the DEIPA (Deterministic (re-)Execution and Interactive Program Analysis) tool. Section 6 describes ongoing and future work, and section 7 presents the conclusions.

2 Testing and debugging of parallel programs

Testing of any program is aimed at a number of experiments with its code to detect unexpected, thus possibly erroneous, behaviors. While the main objective of testing is to reasonably cover the entire set of program behaviors, the purpose of debugging is to run series of relatively simple experiments to isolate and localize specific errors in the code. Program errors are removed and tests run again to check on the corrected program behavior.

The main objective of testing of parallel programs is to find communication errors. Normally, when processes use message passing for communication, an assumption is made that structural sequential parts of the code are then tested first by using appropriate methods for sequential programs. The emphasis is put on detecting and isolating time-dependent errors, which often can only be observed under specific timing conditions.

Three modes of execution for testing of parallel programs should be considered:

1. **Random execution.**
   The program is run with no control on its execution. Input data is specified by the user and the communication sequence only depends on actual conditions in the real system. This kind of testing is very limited—it merely relies on a chance that certain conditions that cause the error may possibly occur. Tests have to be executed several times for the same input data, as errors may be related to some very specific and uncommon system state (eg. network load, unreliable communication, process failure, etc). Adequate monitoring has to be applied, which in turn may result in the probe effect and make recorded data meaningless or void.

2. **Controlled execution.**
   This approach [6] concentrates at one process at the time in order to capture communication errors that can be caused by races. This can be done by analyzing all communication buffers of a single process and determining whether different message receiving orderings could result in different program executions. Unfortunately this technique also has the drawback of the probe effect.

3. **Deterministic execution.**
   The user who is testing a program defines a set of control flow paths through parallel processes that specify a sequence of communication events to occur during a test. The relevant sequence of events determines a specific testing scenario for executing a parallel program in its environment. The program is run according to the scenario and its behavior is logged. Recorded entries in the log sequence are next compared to the expected results given by the specification. Specifically, the order of logged communication events is verified against the specification provided by a script. If they do not agree, this may indicate an error. Using this approach eliminates the probe effect, however, selecting the appropriate set of paths through the program code may require a large amount of work.

Because each of the above methods is essentially different, and is able to find only a certain type of errors, the following new method has been proposed.

Testing of a parallel application can be done at two levels. One is *real execution* and the other is *symbolic execution* of the parallel program under test. Using these two approaches *jointly* has resulted in a new, efficient way of finding time-dependent errors in parallel programs.

Testing activities are organized in two groups: symbolic and dynamic analysis activities.

1. **Symbolic analysis.**
   These activities concentrate on specifying a set of control flow paths determining a sequence of communication events for a specific testing scenario.

   The paper focuses on static analysis of parallel programs written in C, and with interprocess communication implemented with the message passing primitives provided by the PVM system [1]. These programs can be analyzed in two steps:

   (a) *Static analysis of parallelism.* All potentially possible communication events are identified by the means of a reachability analysis. This implies detecting all possible communication actions in the program and all possible connections between processes.

   (b) *Symbolic program execution.* The tool steps through all program statements of respective processes, interacting with the user to consult all conditions it encounters on its way. This analysis results in determining symbolic conditions for each relevant path which has to

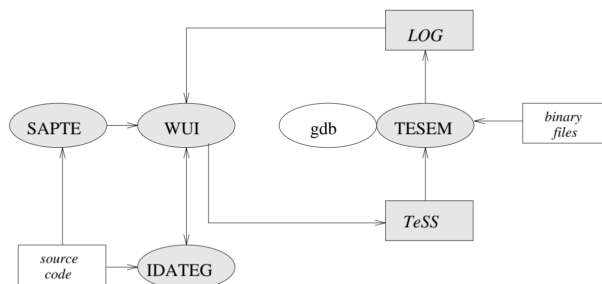

Figure 1: The module structure of STEPS

be satisfied during the real program execution later.

The set of paths selected during symbolic execution and combined with the information on possible communication structure allows to generate the testing scenario script for the deterministic mode execution that follows.

2. **Dynamic analysis.**
   These activities normally assume a random process execution. The program code is run in its real environment and all of its communications events are registered and logged into a special log file. The log file provides an input for the animated program re-execution, which provides the user with the recorded sequence of communication events visualized by special control flow tokens.

   Upon completing (breaking) a dynamic program execution, the user may continue further on with symbolic analysis. The relevant variable values that were set during the real execution are transferred to the symbolic interpreter, so that it may use them as the starting point for further analysis. Each program may be executed several times with different input data and with slightly modified scenarios to better localize the errors.

A testing method should enable stepping back from any point of analysis, either dynamic or static, in order to generate a whole series of alternate scenarios.

In our approach, the STEPS tool (see section 3) is responsible for systematically generating reproducible test patterns for the process spawning and communication parts of a C+PVM program. The information that is collected as a result of the analysis performed during the testing phase, will be used to guide the constrained program execution, under the control of the DDBG debugging tool (see section 4). Such constrained execution is required in order to guarantee the reproducible program execution in a replay mode, as well as to support a more detailed analysis of dynamic program behavior, following the paths that were identified in the testing phase. The debugging phase allows the use of interactive techniques like breakpoints, step-by-step execution, variable inspection, suitably adapted to provide consistent view of the state of the parallel computation, and supported by both text-based and graphical based user interfaces.

3 The STEPS tool

STEPS (Structural TEsting of Parallel Software) [9] is a tool designed for the interactive testing of parallel programs that use PVM message passing primitives for interprocess communication. It supports the modes of execution and testing activities described in section 2. The tool incorporates three basic objects: *a parallel program under test* (source code file and executable file), *test scenario script* (TeSS), and *log file*.

A parallel program is modeled by a set of interconnected nodes, representing either blocks of sequential *processing* statements or *communication events*. Processing nodes encapsulate sequential assignment and decision statements while communication event nodes encapsulate small sets of matching "*send*" and "*receive*" actions. The set of interconnected nodes constitutes a multi-flow graph that specifies the control structure of the program under test. STEPS views parallel program execution as independent control flow tokens progressing through the related multi-flow graph. Each single token progresses from node to node and (possibly) interacts with other control flow tokens.

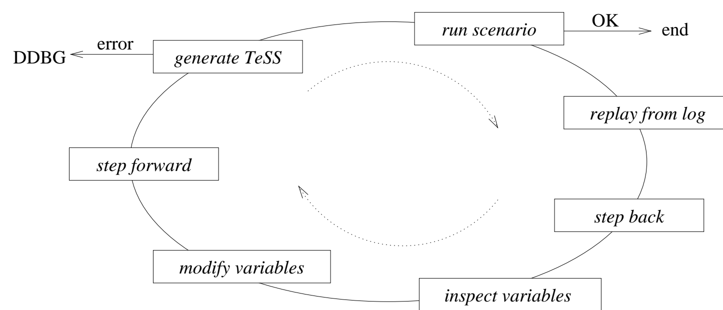

Figure 2: The testing cycle - interactive creation of testing scenarios

STEPS supports four basic groups of activities:

1. Retrieval of information from the program source code in order to construct its respective multi-flow graph representation.

2. Visualization of a multi-flow graph structure for interactive design of testing scenarios.

3. Symbolic interpretation of program paths for designing testing scenarios concerning data values and process timing.

4. Dynamic execution of tests using the logging facility of communication actions to detect the occurrence of communication events.

The above mentioned activities enable three modes of execution for testing of parallel programs. Random, controlled or deterministic execution relies on activities of type (1) when figuring out all possible communication events, and uses activities of type (2) when animating program execution with moving control flow tokens. Deterministic execution, and to some extent controlled execution need (3) to provide appropriate values for program variables. Finally, (4) supports all three execution modes by monitoring states during the dynamic program execution.

STEPS consists of four major functional subsystems that are presented in figure 1:

- **Window user interface [WUI].**
  WUI is a window user interface to manage tool functionality, to interact with static analysis of a program text, and to design testing scenarios.

- **Static analyzer of PVM text [SAPTE].**
  SAPTE is a static analyzer of PVM text to identify communication events using reachability analysis and determining various quality parameters.

- **Interactive data test generator [IDATEG].**
  IDATEG is an interactive data test generator to determine symbolic path conditions and to assist the users in finding suitable data for program execution.

- **Testing scenario execution manager [TESEM].**
  TESEM is a testing scenario execution manager to execute the run-time code of the program under test for various testing scenarios and to record its relevant behavior.

Two kinds of input information are normally required by STEPS: the program input source code and user commands. An initial user command is to start static analysis of parallelism based on input source code. As a result of this analysis three kinds of information are retrieved from the program text: *an interconnection structure* of component processes' *graphs*, the respective sets of matching communication actions identified as *communication events*, and *timing* (ordering) relations between identified events.

Information retrieved by SAPTE is used by WUI to display a multi-flow graph structure needed for preparing testing scenarios. Information about all node connections and communication event nodes is also provided by SAPTE. Based on this the user can design a testing scenario using interaction provided by WUI and then run the tested program for it under the supervision of TESEM. The program can be run also without any testing scenario. In this case such testing scenario can be produced later using the logged information.

Basic tool functionalities provided by SAPTE, WUI, IDATEG and TESEM enable the user to design testing

scenarios interactively. This forms a specific testing cycle, as outlined schematically in figure 2.

There are two ways to enter the cycle in the *run scenario* phase. One is via *static analysis* and enables deterministic execution according to the scenario. Another one is *random execution* without any specific scenario. After the execution, values of program variables can be read. Based on the recorded log entries, the *replay* phase can be performed. The next four phases: *step back*, *inspect*, *modify variables*, and *step forward*, enable the preparation of a new (modified) testing scenario script, that can be used to re-enter the cycle.

Dynamic execution supervised by TESEM relies on the standard GNU 'gdb' debugger. The debugger is used to control installation and removal of breakpoint traps, introduced by the scenario script, and to implement timing of the processes and modifying program variables.

STEPS was implemented mainly using the C/C++ languages, with the major parts of IDATEG implemented in Prolog. The user interface uses the Motif library.

4 The DDBG tool

The DDBG (Distributed DeBuGger) [3] tool is a distributed debugging engine that provides the following functionalities for distributed programs written in C, and with interprocess communication implemented using the message passing primitives provided by the PVM system [1]:

- Central user interface to control the processes being debugged.
- Simultaneous access to multiple (high-level) client tools.
- Dynamic attach and detachment of client tools to the debugging engine.
- Global view of the system being debugged.
- An event trace is collected with minimal information to support program replay [10]. This allows reproducible behavior (concerning external events).
- Support of debugging control commands during a replay session.
- A checkpoint facility under replay mode will support execution replay from an intermediate point, instead of from the beginning of the program only.

DDBG is organized in terms of a distributed collection of daemon processes that cooperate in order to control multiple/remote debugger instances (called *process-level debuggers*), each associated with an individual application (PVM) process. The architecture of this tool is illustrated in figure 3, and includes the following components:

- **Debugging interface library.**
  This library provides full access to all debugging functionalities of the tool, and implements a transparent communication channel between the client tool and the main daemon.
- **Main daemon.**
  The main daemon manages all the interactions between the client processes and the process-level debuggers, controlling the application process components.
- **Local daemon(s).**
  In each node a local daemon works as a gateway between the main daemon and the process-level debuggers located in that node.
- **Process-level debugger(s).**
  This is a sequential debugger[1] used to control and inspect a sequential process (component of a distributed application).
- **Text user interface.**
  A text console, giving the user command line access to the debugging functionalities supported by the DDBG engine [5].
- **Graphical user interface.**
  The X-Windows based graphical user interface gives access to the DDBG functionalities. It supports a browser of the processes under debugging, and a per-process window that display process variables, that are valid in the current execution context of the associated process. The refreshment of the variables values is done on explicit command, avoiding heavy communication between the graphical user interface and the main daemon.

[1] Currently, we are using the GNU "*gdb*" from Free Software Foundation.

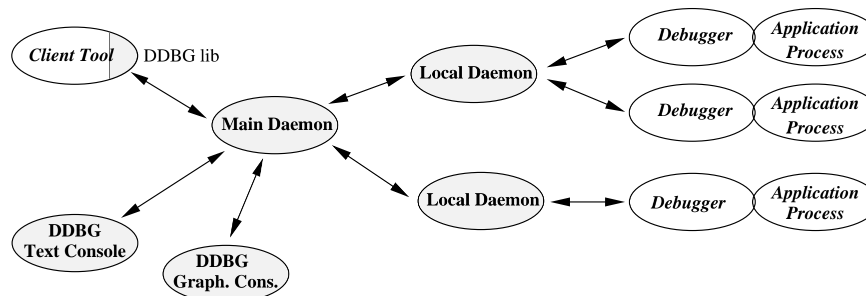

Figure 3: The DDBG architecture

DDBG was developed as a distributed developing infrastructure with flexibility in mind: in the user-level debugging commands, in the interfacing to a diversity of high-level client tools [8, 2, 3, 4, 13], and in the handling of heterogeneous process-level debuggers. It shares the latter goal with related systems, namely [7]. Currently there is a working prototype implementing all of the above functionalities except checkpointing, which is under development.

5 Integrated testing and debugging

The testing tool allows to identify potential critical paths and critical sections in the program. The debugger tool can inspect and control the program behavior, helping in the localization of a program's bug and its causes. When composing both tools, one must ensure that the program will run and behave as expected, *i.e.* the critical conditions found with the testing tool will occur.

The composition of the testing and the debugging tools starts by re-running the program and forcing it to follow some specific path, ensuring that it will reach the critical points previously located by the testing tool and will stop in a consistent state (also called a "*global breakpoint*"). Here, one question should be asked: *What to do whenever a global breakpoint is reached?* Two main options have been considered:

- **Generate a log file.**
  This could give a great help in localizing the program's bugs, but another question arises: *What events should be logged?* The answer to this question is not trivial and depends on the program functionality, current data and the user preferences.
- **Support interactive analysis driven by the user.**
  In this case, when the program stops in a global breakpoint, the user is allowed to interact with it, inspecting and/or changing the program state and variables, making this approach much like the "conventional interactive debugging".

We claim that an integrated testing and debugging environment should provide and use both of the above presented options simultaneously, allowing the user to interact with the running program and providing full access to the execution history recorded in the log file. The work presented in this paper reports our experience on the implementation and use of the last of the above presented options. There are alose plans to integrate the DDBG debugger with a monitoring tool, providing more functionality in the integrated testing and debugging environment.

The DEIPA (Deterministic (re-)Execution and Interactive Program Analysis) tool was developed to support the integration of the STEPS testing tool (see section 3) and the DDBG debugging tool (see section 4). DEIPA acts as an intermediary between those tools, recognizing and processing the output of the STEPS tool—the TeSS file—and converting it into (a set of) commands for the DDBG tool. To support this functionality, the DDBG capability of having multiple simultaneous client tools has been used, by having the DEIPA tool controlling the execution of all the processes of the distributed application and having a textual user interface (DDBG console) and/or a graphical user interface to inspect and change the program state.

The DEIPA tool is mainly composed of 3 modules: the *Console*, the *Vid Database Manager*, and the *Replayer*. The architecture of the DEIPA tool and its relations with the STEPS and DDBG tools are presented

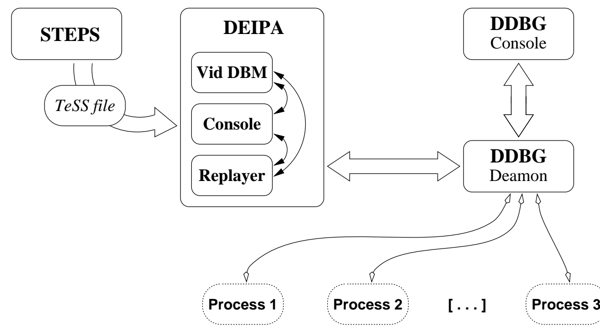

Figure 4: DEIPA integration with STEPS and DDBG

in figure 4, and explained below.

- **The Console module.**
  This module acts as the user interface to the DEIPA tool. Actually, this interface is based on a console (command-line oriented user interface)[2], from which the user can load a TeSS file and control the (re-)execution of the sequecnial processes of the distributed application. The console provides the user with some basic commands that are directly used to control program execution, *e.g.* **`load`**, or that are converted into (a set of) debugging commands and applied to the processes via the DDBG tool, *e.g.* **`step`**.

- **The Vid Database Management module.**
  During the statical analysis, the STEPS tool always refers to processes through symbolic identifiers (this is mandatory, as the processes execution isn't real but simulated). During debugging the processes are really running, and a mapping between symbolic and the real process identifiers is required. This module manages this mapping function, and is composed of 4 sub-modules (the first two are integrated into the DEIPA tool and the last two are used by the client processes):

  1. **SPAWN_TABLE parser.**
     A parser that recognizes the syntactical structure of the SPAWN_TABLE section of the TeSS file and builds a database with the relevant information.
  2. **Vid database manager.**
     This sub-module manages the database created by the above cited parser and updates some extra fields with the dynamic process information,*e.g.* PVM Task Identifiers, when requested.
  3. **Vid DBM client functions.**
     A set of library functions that should be linked to the client code, providing access to the Vid Database Manager. This allows the client process to get data from the database, *e.g.* get the process Task Identifier given its Virtual IDentifier, and store data in the database, *e.g.* register the Task Identifier of a new process. This library provides the interface to the Vid Manager module of the DEIPA tool, supporting the mapping between STEPS Virtual IDentifiers and PVM Task Identifiers during program replay, as well as process status and other informations.
  4. **PVM code instrumentation.**
     An extra module has to be added to the client source code. It's an instrumentation library for PVM code, in order to send registration messages to the Vid Database Manager when spawning new PVM tasks. Every time a new process is spawned, the parent process sends a message to the Vid Manager with the relevant data needed to the future determination of a process Task Identifier based in its Virtual IDentifier.

- **The Replayer module.**
  The replayer module is responsible for the mapping of console commands into DDBG commands. It's also responsible for the processes con-

[2] A graphical user interface to comlement this text-oriented user interface is currently under development.

trol needed to force a process to follow the specific path that is specified in the TeSS file.

In order to allow the STEPS and the DEIPA tools to run in different physical architectures, all the data transfered between these tools should be architecture independent, and an option for ASCII file(s) has been made. In order to provide different types of information in the same ASCII file, a specific file format has been defined [12].

The TeSS file contains all the relevant data (excluding program source files) necessary to support the deterministic replay of the sequential processes of a distributed application in the PVM environment.

Concerning the interface between STEPS and DEIPA tools, three sections from the TeSS file are relevant:

1. **START_FILE**.
   The STEPS tool assumes that the distributed application has a root process, which will start all other application processes. This information is redundant, as the file for the root process is also declared in the SPAWN_TABLE section of the TeSS file (see below) with special values in some of its fields, but it eliminates the need of an extra analysis of the SPAWN_TABLE.

2. **SPAWN_TABLE**.
   The SPAWN_TABLE section of the TeSS file specifies the parental relation between processes, as well as part of the information needed to map the Virtual IDentifiers in real Process Identifiers. Some extra fields are also available for each process, in order to verify the validity of the source files. These files are not included in the TeSS file, as they can be corrupted or may have been changed between the generation of the TeSS file and the program replaying phase.

3. **INITIAL**.
   This section defines a sequence of global breakpoints. A global breakpoint consist of a set of subbreakpoints, one for each running process of the distributed application being tested. They will be used to control the distributed application behavior (*i.e.* the path to be followed by each application process).

6 Further Work

Besides the work already done, some improvements to the individual tools and the integrated system are planned, such as:

- Automatization of selected testing activities like data generation and generation of simple testing scenarios.
- A database of behaviors of programs with typical bugs injected.
- Development of a testing methodology.
- Integration of DDBG with a monitoring tool, such as Tape/PVM [11].
- Development of full functional GUI for DDBG and DEIPA tools.
- Perform further testing and debugging of the software!

7 Conclusions

The use of the STEPS tool complemented with the simultaneous use of both DEIPA and DDBG consoles provides the user with a very flexible and powerful environment to locate potential and real program bugs, through the symbolic analysis of the source files and the controlled execution and inspection of the running processes.

The TeSS file is generated by the STEPS tool and can be loaded into the DEIPA tool environment using its console. Using the same DEIPA console, the program can be started and ran until a global breakpoint is reached. As each global breakpoint is reached, the user can refresh the information available in the graphical interface and/or switch to the DDBG console to analyze in further detail any of the processes under debugging, *e.g.* by inspecting their program stacks and/or by reading/setting local and global variables, or step into the next global breakpoint.

If some unexpected behavior is found and its causes are determined, it is possible to correct the problem and close the testing/debugging development cycle, by restarting the analysis with the STEPS tool.

Considering the relevance of the static and dynamical analysis made by the STEPS tool and reflected in the TeSS file, the DEIPA tool has proven to be adequate to

partially execute the distributed application and to force it to follow the pre-determined path.

The functionality obtained using both consoles together, the DEIPA and DDBG consoles, is a major improvement over the traditional debugging of distributed application based in a collection of sequential independent debuggers, as it is possible to know what the program flow will be *before* its execution.

Acknowledgments

This work was partially supported by the EC within COPERNICUS Programme, Research Projects SEPP (Contract CIPA-C193-0251) and HPCTI (Contract CP-93-5383).

References

[1] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. A User's Guide to PVM Parallel Virtual Machine. Technical Report ORNL/TM-118266, Oak Ridge National Laboratory, USA, 1991.

[2] J. Cunha and J. Lourenço. An Experiment in Tool Integration: the DDBG Parallel and Distributed Debugger. Technical report, Departamento de Informática da Universidade Nova de Lisboa, Portugal, Oct. 1996.

[3] J. C. Cunha, J. Lourenço, and T. Antão. A Debugging Engine for a Parallel and Distributed Environment. In *Proceedings of DAPSYS'96, 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems*, pages 111–118, Misckolc, Hungary, Oct. 1996.

[4] J. C. Cunha, J. Lourenço, and T. Antão. A Distributed Debugging Tool for a Parallel Software Engineering Environment. In ONERA, editor, *EPTM'96, 1st European Parallel Tools Meeting*, Chatillon, France, Oct. 1996.

[5] J. C. Cunha, J. Lourenço, and T. Antão. DDBG: A Distributed Debugger — User's Guide. Technical report, Departamento de Informática da Universidade Nova de Lisboa, Portugal, Aug. 1996.

[6] S. K. Damodaran-Kamal and J. M. Francioni. Testing Races in Parallel Programs with an OtOt Startegy. In *Proceedings 1994 Int. Symp. on Software Testing and Analysis*, pages 216–227, WA, USA, 1994.

[7] R. Hood. The P2D2 Project: Building a Portable Distributed Debugger. In *Proceedings of the SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*. ACM, May 1996.

[8] P. Kacsuk, J. Cunha, G. Dózsa, J. Lourenço, T. Fadgyas, and T. Antão. A Graphical Development and Debugging Environment for Parallel Programs. *(To appear in) Parallel Computing, Elsevier Science*, 1997.

[9] H. Krawczyk and B. Wiszniewski. Interactive Testing Tool for Parallel Programs. In P. C. Chapman & Hal: I. Jelly, I. Gorton, editor, *Software Engineer for Parallel and Distributed Systems*, pages 98–109, London, UK, 1996.

[10] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs wirh Instant Replay. *IEEE Transactions on Computers*, C-36(4), April 1987.

[11] E. Maillet. Issues in Performance Tracing with Tape/PVM. In *Proceedings of the EuroPVM'95*, pages 143–148, Lyon, France, 1995.

[12] M. Neyman and J. Lourenço. Integration of STEPS and DDBG. Technical report, Departamento de Informática da Universidade Nova de Lisboa, Portugal, July 1996.

[13] S. Winter and P. Kacsuk. Software Engineering for Parallel Processing. In *Proceedings of the 8th Symposium on Microcomputer and Microprocessor Applications*, pages 285–293, Budapest, Hungary, 1994.