A Debugging Engine for a Parallel and Distributed Environment*

José C. Cunha, João Lourenço, Tiago Antão

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática
2825 Monte Caparica
Portugal
{jcc,jml,tra}@di.fct.unl.pt

Abstract. This paper describes a debugging interface that has been developed for a parallel software engineering environment and that was developed on top of the PVM environment in the scope of the SEPP and HPCTI projects of the COPERNICUS Programme. The main goal of this interface is to provide the basic debugging functionalities that are required by some components of that environment. We give special attention to the requirements posed by high-level tools of the environment, and to the need of providing a flexible debugging support layer that can be suitably adapted and extended. We present the system logical architecture and the interface specification of the debugging engine. We discuss its interfacing with other components of the environment, namely a graphical editor for the GRAPNEL visual parallel programming language, and a testing tool. We finally describe current work on the improvement of the debugging engine.

Keywords: Debugging, monitoring, parallel processing, software tools.

1 Introduction

The task of developing parallel and distributed programs faces many difficulties making it very hard to understand and predict the logical behavior of a program as well as its performance. This justifies the need to develop suitable tools and to integrate such tools in a flexible and user-friendly environment.

The work we describe in this paper is part of an ongoing project which aims at the development of a software engineering environment for parallel applications [15] [26]. Overall, this project encompasses issues in the design and implementation of visual programming components for graph-based design of parallel programs, mapping components for specific hardware target distributed-memory architectures, simulation support tools, real parallel execution with monitoring and load-balancing support, and testing and debugging tools. Here we discuss design and implementation issues for one component of such parallel programming environment: the debugging tool.

The organization of the paper is as follows. In the next section we discuss the main design issues for a distributed and parallel debugging tool, and briefly review the current state of the art. Then we present the distributed debugging engine DDBG we have developed on top of the PVM environment. A section follows with the description of how our tool interfaces with two components of the SEPP environment, namely the GRAPNEL graphical editor [16] [24] and the STEPS testing tool [17]. In the conclusions we discuss our ongoing work and future research on this topic.

2 Debugging Parallel and Distributed Systems

Traditional sequential debugging techniques offer the following typical functionalities: cyclic interactive debugging, memory dumps, tracing, and breakpoints [1]. However, these techniques cannot be directly applied in a parallel and distributed environment. This is due to the following facts: parallel and distributed programs exhibit non-deterministic and nonreproducible behavhior; lack of global state makes it very difficult to manage global predicates on the system state; and there is an intrusion effect of the debugging system upon the observed program.

The most immediate approach to support debugging functionalities in a parallel and distributed environment is through the collection of multiple sequential debuggers, each attached to an application process. This may provide similar commands as available in conventional debuggers, possibly extended to deal with parallelism and communication. However, this does not solve any of the above difficulties.

In the past 10 years, several proposals have been made to address these problems [12] [5] [2] [8] [11] [13] [7] [14]. We are particularly interested in an approach that models the debugger as an event-based system. This provides

* In "*Proceedings of DAPSYS'96, 1st Autro-Hungarian Workshop on Distributed and Parallel Systems*", Misckolc, Hungary, October 1996.

several interesting characteristics: it uses a previously recorded event trace, in order to analyze the execution history, and to guide program replay with reproducible behavior, and so it can make use of (suitably adapted) conventional debugging techniques; it may rely upon monitoring techniques, for event generation and recording; it can benefit from optimizations that allow to reduce the amount of collected information, namely based on the instant replay technique [12]) and so it can greatly reduce the intrusion effect; it eases the management involved in the global coordination of parallel processes and inspection of global system states.

There are already several systems offering some of the above characteristics. However, most of these systems have several limitations in the help they offer towards a better user understanding of the application behavior, and in the type of user interaction towards a selective user-driven examination of the causes of incorrect program behavior, as well an inadequate integration with other tools of the parallel programming environment (such as simulator and visualization tools). Additionally some of those systems are very much dependent upon a specific hardware or operating system plataform.

2.1 Integrating Testing and Debugging Tools

One important issue concerns reaching a close integration of static analysis and dynamic analysis methods in order to guarantee the final quality of the parallel and distributed software. Besides formal methods to assure the quality of parallel programs, systematic testing approaches play a very important role in this process. The development of a methodology and tool to aid the user in the process of identifying the paths which should be generated and tested, is a key component of an advanced testing and debugging environment, and encompasses several issues: test case design, data generation, test generation and execution, test evaluation, and global quality assessment with the help of decision support systems. This aspect is being investigated within the scope of the mentioned Copernicus projects [15] [17], and a detailed discussion is beyond the scope of this paper. However, an important aspect of the approach followed in our project is to allow the testing and evaluation stages to be performed through a close interaction with the dynamic debugging tool. This is achieved by supporting user controlled execution of the paths under test, allowing the user to inspect program behavior at the desired level of abstraction and with the guarantee of the reproducibility of its execution.

2.2 Providing Basic Debugging Support to other Tools

A large diversity of debugging tools have been developed for distinct parallel and distributed programming languages, as well as for distinct parallel computer systems. In particular, the appearance of shared-memory and distributed-memory multiprocessors during the 80s has originated the need to develop specific debugging support, both at the level of the operating system and at the level of the communication libraries [5] [8] [7]. The problem with these debugging tools is that usually they are specific to a particular runtime or hardware environment, and as such they are very difficult to adapt to other parallel platforms. On the other hand, the evolution of parallel and distributed systems towards more user-friendly environments requires a very flexible software development platform for the experimentation with new programming models and the corresponding development tools, e.g. for monitoring, debugging, animation, visualization, and performance analysis. Several years of experimentation with these parallel computing platforms still show a need to provide a more unified framework to support the implementation of high-level debugging functionalities. This framework must address two fundamental issues:

- A well-defined interface must be provided to be used by high-level tools of the parallel development environment, namely graphical editors, graphical interfaces, runtime support systems for distinct parallel and distributed language models, and testing and high-level debugging tools;
- A well-defined interface must be provided to the underlying operating system and hardware platform, assuring portability and adaptability of the debugging support architecture, while still allowing efficient implementation on top of each specific physical environment.

Each of the above issues may be separately addressed, to a certain extent.

Concerning the first issue, we have defined an interface to a library of debugging primitives, as described in the next section. Besides providing the commands that are typically supported by conventional debuggers for sequential computation models, this interface provides the basic primitives to inspect and control distributed processes. The design of the debugging interface assumes an asynchronous mode of interaction with the client tools, and its current implementation relies upon a TCP/IP socket-based communication protocol. However, as this protocol is hidden within well-defined interface functions it may change without the need to modify the client tool.

Concerning the second issue, we are exploiting the fact that most of the basic debugging support functionalities can be integrated into a monitoring layer. The goal of a monitoring activity is to gather runtime information about an observed system, including several aspects such as collecting information, registering the detected events, generating and maintaining suitable event trace formats, and providing suitable interfaces to high-level tools [10]. The results provided by a monitoring tool can be used by very distinct tools, at distinct levels of a computing system,

and for very distinct purposes, namely to provide qualitative descriptions of program behavior, to feed performance-oriented tools, and to support program testing and debugging. Concerning the latter aspect, the debugging activity requires some monitoring support, namely for inspection of processes and communication channels, and for process control.

A discussion of monitoring issues is beyond the scope of this paper although they are also being addressed in the above mentioned project [18]. However, it is interesting to note that it is also being recognized that, in order to handle the distinct and evolving requirements posed by high-level tools, a monitoring layer should provide well-defined interfaces, and support easy definition of new interfaces. A uniform set of mechanisms for event handling, i.e. accessing, filtering, searching, and manipulation, is required. This is the only way to offer a stable monitoring layer that will be used by a large number of tools, and will be able to evolve, as new needs arise. Only a few experiments have been proposed towards such goal, namely involving object-based interfaces for event access. A significant and very broad effort has recently been launched towards meeting the above mentioned goals [9]. We are currently pursuing a similar goal, as far as debugging support is concerned.

In order to experiment with the issues involved in the design of a flexible and general-purpose debugging architecture, we have implemented the above mentioned interface library on top of the PVM [21] system. From the user point of view, any application or tool can be linked with the interface library and access all the distributed debugging functionalities. From the implementation point of view, the current design has a distributed organization consisting of multiple monitor/debugger instances which are scattered on the nodes of a PVM platform.

3 A Distributed Debugging Engine

In this section we present the interface provided by our distributed debugging tool called DDBG and its logical architecture. The debugging functionalities may be summarized as follows:

- Dynamic attachment and deattachment of debugger instances to already running distributed processes; control of remote debugger instances from a central debugging user interface;
- An interface library that gives access to such control of remote debuggers, and which can be used by high-level tools, like a graphical editor, and testing tools;
- An event trace is collected with minimal information to support program replay in PVM programs. This allows reproducible behavior and will make the debugging control commands available 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.

Currently there is a working prototype implementing the first two of the above functionalities. Full support for program replay and checkpointing is still under development. The prototype runs on the PVM environment, and relies upon a well known debugger—the GNU *gdb*—to provide conventional debugging commands within each sequential process.

Any user tool can access the debugging engine as a client process that uses the debugging library to interact with the main debugging daemon. The main daemon manages all the interactions with the client process, by forwarding the debugging commands to the machines where the application processes are placed, and gathering their corresponding answers. This is achieved by having a local daemon on each machine that is responsible for the activation and control of multiple debugger instances located in that machine. Each application process can be dynamically attached (detached) to (from) a distinct debugger instance. The master daemon and the set of local daemons form a distributed architecture that can also be used to perform distributed monitoring functions.

Communication between the master and the local daemons is based on the PVM primitives. Communication between the master daemon and the user client process is based on UNIX sockets. In a first design UNIX domain sockets were used which requires the master daemon to run on the same machine as the user client process. However this limitation is removed by using UNIX Internet sockets for that communication. Communication between the local daemons and the debugger instances on each machine is currently based on UNIX pipes.

This is illustrated in the following figure where the main components of the debugging architecture are shown.

4 Debugging Library functions

PVM uses *task ID's* (integers) to identify the processes, but an user application or tool may use specific *Process ID's* (strings) to do the same. In order to support the mapping between the user symbolic name and the PVM naming scheme, a name mapping function is provided. This allows any of the library primitives, as well as the corresponding user consoles, to refer to string or integer process identifiers, although in the following description we always use string identifiers.

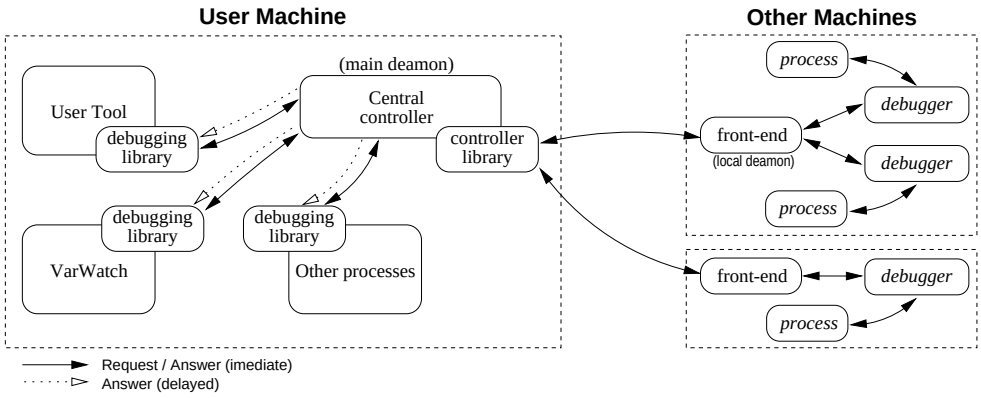

Figure 1. The debugging architecture

Currently the communication between a client and the master debugging daemon requires the client to invoke a *system call* (see **init()** below[2]) to initialize the library [3]. If there is no main debugging daemon running at the time the initialization is requested, it will be started automatically. This initialization also establishes a communication channel that will be used for future interactions between this client and the debugging engine[4].

All the services provided by the debugging engine are classified as belonging to one of two classes:

- **Immediate answer services.** These services will either fail or are immediately executed by returning the relevant data as output parameter(s) to the corresponding *system call* function.
- **Delayed answer services.** These services—e.g. **next()**—will either fail with an immediate return, or they may take an unpredictable amount of time until its execution is finished. In the latter case, the corresponding library function also returns immediately to the calling process but informs the user about the unavailability of the data, which can later be gathered by invoking the function **get_special_info()**, as defined below.

All the library primitives return a status code concerning the immediate or delayed type of answer, although this is not shown in the function description that is presented in the following.

In principle the interaction with the debugging system can be completely transparent. The client just invokes library functions, and gets the corresponding returns in an immediate way or in a delayed way. In the latter case, the following function can be used:

- **int get_special_info(char *procid, struct code_info *info)**
  If there is returning data available (from a *delayed answer command*), the funtion will return a corresponding status indication with the user Process ID in `procid` and the data in `info`. Otherwise it will inform about the unvailability of the data.

This function has a non-blocking semantics so that the asynchronous execution of the user application that controls the debugging interface is allowed. It is up to the user to poll the return status by repeated invocation of this function, or alternatively by selectively waiting on the UNIX file descriptor that was returned by the function **init()**.

4.1 Data types used

The library uses several different structured types which describe information concerning relevant program locations, breakpoints, stack frames, processes, variables and function argumentsi[19]. As an example, the information given by *struct code_info* is presented below and it is used to describe references to the program code.

```
struct code_info {
    int  line_no;
    char function_name[];
    char source_file[];
};
```

The functionalities currently offered by the debugging interface are summarized in the following sections[5].

[2] Actually, all library function names are prefixed with dbg_.
[3] The current prototype assumes that the PVM system is already running.
[4] Currently, a UNIX file descriptor is returned corresponding to an interprocess communication socket.
[5] They are adapted from identical functions provided by the GNU *gdb* debugger, but they operate upon multiple distributed processes.

4.2 Controlling the debugging session

The functions **init()** and **end()** support, respectively, the initialization and the cleanup of the debugging environment.

The dynamic attachment of existing processes to new debugger instances is supported by the function **attach(char *procid, struct code_info *info)**. It starts a new debugger and attaches it to the process specified in `procid`. The process will be stopped and will become under control of the debugger. The relevant data is returned in `info` to determine where the process was stopped. A complementary function (**detach(char *procid)**) kills the debugger associated with a given process and leaves the process running free. The function **kill(char *procid)** kills the specified process and its associated debugger.

Information about new components in the debugging environment may be obtained with the function **get_news(struct news_info *info)**. Currently, only information on new processes on the system is available. It returns a status code indicating if there are news or not, and the information may be obtained referring to user symbolic process identifiers or to PVM task IDs, depending on a selection field in the `info` structure.

4.3 Managing breakpoints

Basic support is provided to control program execution through breakpoints which are currently only associated with individual processes. The function **set_break(char *procid, struct code_info *info)** sets a breakpoint on a given process process in the line/function that is specified in `info`. An unique `breakpoint id` is returned. A similar function sets a temporary breakpoint (one time only).

Breakpoints can be set conditionally, depending on the evaluation of an expression, by invoking **set_cond_break(char *procid, struct code_info *info, char *exp)**. If the expression `exp` evaluates to `TRUE`, a breakpoint is set in the specified process, in the line/function specified in `info` (conditional breakpoint). The expression is evaluated every time the breakpoint is reached. An unique `breakpoint id` is returned.

Watchpoints can also be specified for a given process by invoking the function **set_watch(char *procid, char *exp)**. This function sets a watchpoint on the given process `procid` such that the process will stop when the condition in `exp` will become `TRUE`.

Breakpoints can be temporarily disabled, enabled, cleared (permanently removed), or ignored a certain number of times by invoking corresponding library primitives.

4.4 Managing the program stack

Detailed inspection of stack frames is possible within an specified process by invoking the function **select_frame(char *procid, int count)**. This function selects a new current frame for that process such that the frame is located `count` frames up or down referring to the current frame. If `count` = 0, it does nothing. The *distance* between the previous and the new current frames is returned by the function.

4.5 Controlling the execution of the (debugged) processes

The library supports classical debugging commands to control the execution of each individual process in a detailed way. Using these commands, as well as the other commands that handle breakpoints, and display or update process information, it is possible to implement higher level debugging functionalities. This was used to implement the interfacing of DDBG to other tools.

The function **run(char *procid)** allows to start running a (previously spawned) process from the beginning, until a breakpoint is found or the expression of a watchpoint is true, or until the end, if none above the conditions ocurs. The execution of a stopped process can be continued by invoking **continue(char *procid)**. The execution proceeds until one of the above mentioned conditions occurs. Another function **finish(char *procid)** allows to run a process until the *selected stack frame* returns, as defined by the `select_frame()` function. It is also possible to pop the selected stack frame without executing and return in `info` the relevant data to determine where the process was stopped.

Step by step execution is supported by the functions **next(char *procid)** and **step(char *procid)** which execute the code until the next instruction, by respectively executing subroutine code as one instruction only or as normal code.

Interrupting the execution of a process is supported by the **interrupt(char *procid, struct code_info *info)**. The returned `info` contains information to determine where the process was stopped.

4.6 Information

These primitives allow to inspect or modify local information in each process.

The function **print(char *procid, char *fmt, char *exp, char *value)** returns in `value` the result of the evaluation of expression `exp` in the context of the given process `procid`, according to the given format `fmt`.

By specifying the `var_info` structure, where a variable's name is given, it is possible to get and set the values of individually named process variables, by invoking, respectively, the function **get_var(char *procid, struct var_info *var)** or **set_var(char *procid, struct var_info *var)**.

Information on process breakpoints is also possible using **info_break(int brkptid, struct brkpt_info *info)**. The information about the specified breakpoint is returned in the `info` structure, including e.g. number, type (simple or conditional), line number or expression associated.

The following text could be printed based on a set of return values of this function:

Num	Type	Process	Filename	Where	Condition	State	Hits
1	Breakpoint	0x0a5f8abc	main.c	f:main()		tempign	1
2	TempBrkpt	0xa8762345	main.c	f:event()		disable d	5
3	CondBrkpt	0x00546543	utils.c	l:123	n==5	ignored	0
3	Watchpoint	0x4a7f6bc3	main.c		event==32	enabled	2

Inspection of the process stack is obtained by calling several functions which return a trace of selected stack frames in a given process with information such as the function name, arguments, number of local variables and for each variable, its name and value.

Finally, information about the status of a process can be obtained through the function **info_process(char *procid, struct proc_info *info)** as illustrated below.

If process `0x45677654` is running, and process `0x56784321` is stopped at a breakpoint, then the following information could be reported:

Symbols from "/home/guest/myprogram"

TaskId	Status	File	Line	Function
0x45677654	running			
0x6798ab3c	ready			
0x56784321	stopped	main.c	123	fibb

5 Interfacing the Debugging Engine to other Tools

In order to assess the usefulness of our distributed debugging interface, we discuss our experimentation with its interfacing to two components of the SEPP programming environment. Here we just focus on the specific requirements put upon the distributed debugging system by those tools, which are described elsewhere [15].

5.1 X-based Interface to the Debugging System

In order to provide a more user-friendly interface to the debugging commands, another client of the master debugging daemon actually implements an X-based interface allowing the user to control and inspect the distributed processes. All the interfacing of this client to the master debugging daemon actually relies upon the already described functionalities. So this is a good test to evaluate the flexibility of the debugging interface.

5.2 Interface to the GRAPNEL editor

The SEPP environment is centered around the GRAPNEL model, a graph-based parallel programming language that is described elsewhere [16] [24]. This model supports a structured style for designing parallel applications, and it is implemented through a set of integrated tools, such as a graphical editor, a compiler to an intermediate textual representation, and to the C language extended with PVM primitives, a visualization tool, a simulator, a testing and debugging tool, and monitoring and a load-balancing tools. The development of those tools is a major effort being undertaken within the mentioned projects [26].

Concerning debugging, the GRAPNEL editor offers an user-friendly interface that allows to invoke debugging commands with reference to the graphical entities that are displayed in the user windows. On the other hand, there is a requirement to display in a convenient way the debugging outputs such that only GRAPNEL abstractions should be handled by the user at this level. This offers a very high-level interface to the user, such that the information on specific debugging commands is directly related to the GRAPNEL source program, e.g. by highlighting corresponding entities in the graphical representation, and their corresponding lines of source code in the textual program representation. The editor interface also accesses a X-based window which displays, under user control, the variables defined within the GRAPNEL structures. This X-based interface is automatically started when the user hits the debugger item under one of the menu options provided by the editor. Then a display is presented including the user processes and the user can select the processes to be monitored, and individual variables within these processes. This is part of the functionality that is provided by the X-based interface to the debugging system as mentioned above.

The GRAPNEL editor [25] runs as a process that handles asynchronously generated events, namely user interaction events. When interfacing the editor to the distributed debugger, this asynchronous mode of operation is handled by using the described function **get_special_info()**). A more efficient control is possible by having the editor directly accessing the interprocess communication socket that is created by the function **init()**. Although

this is not so transparent when compared to the exclusive use of the above function **get_special_info()**, it allows the editor to selectively wait [6] on that socket.

5.3 Interface to a Testing tool

Within the scope of the SEPP project, a significant effort has been centered upon the study and development of a testing methodology and associated software tools [17]. Namely, an interactive tool was developed providing the expertise to systematically generate reproducible test patterns for the concurrency and communication-based program structures. During the testing stage, distinct testing scenarios are identified, their corresponding program paths are generated, and then corresponding scripts are produced so that it is possible to submit them to a real parallel execution. Based on that information, generated during the testing phase, it is possible to control the replay of program execution, suitable instrumented with calls to the debugging system breakpoints, stepwise execution, etc. The interfacing of such testing tool (called STEPS [17]) to the distributed debugging system is directly supported by the mentioned script files which consist of sequences of debugging commands such that specific execution paths are followed under real execution.

5.4 Interface to a Parallel Logic Programming Language

Our distributed debugging system is currently being used to develop a distributed debugger for PVM-Prolog [23]. PVM-Prolog is an extension to Prolog [27] that provides full access to the PVM environment. Parallel and distributed applications are built by specifying multiple Prolog processes, which cooperate by message-passing primitives. The interface predicates closely match the corresponding primitives of the PVM system, with suitable interpretations according to Prolog semantics, e.g. messages are interpreted as Prolog terms. In this implementation, the debugging library is made accessible to each Prolog process through a similar set of basic predicates for process control and inspection.

6 Conclusions and Future Work

We have discussed issues in the design and implementation of a flexible distributed debugging engine. Many existing debugging tools are tied to specific hardware and software environments, or are integrated within specific programming environments. We decided to built the DDBG system due to our goal of investigating issues in the design and implementation of an unified layer for monitoring and debugging.

In summary the current design of the distributed debugging engine provides a resonable flexibility as it allows the user tool, the master daemon, and the remote debuggers to run on distinct machines or processors. On the other hand, our experimentation shows that the current definition of the interface has enough flexibility to accomodate very different tools, and it is extensible to support, for instance, the development of the program replay facility. The identification of the sources of non-determinism which arise in a PVM program is the first step in the design of such a replay facility. Once identified such events, the tracing mechanism is supposed to collect them so that their effective execution ordering may be reproduced during further executions. We are developing a basic design for the support of such a functionality under the DDBG architecture for the PVM system.

Ongoing work relates to the implementation of the DDBG architecture, coupled with the support of distributed monitoring functionalities, on top of a heterogeneous distributed architecture consisting of several UNIX nodes, a cluster of ALPHA DEC workstations and two multicomputer machines. The internal architecture of the DDBG system as well as the communication schemes between daemons, debugger instances and application processes will be evaluated and adapted to such environment.

Acknowledgments

This work was partially supported by the CEE COPERNICUS Programme, SEPP Project (Contract CIPA-C193-0251) and HPCTI Project (Contract CP-93-5383), by the Portuguese CIENCIA Programme, and DEC EERP PADIPRO (Contract no. P-005).

References

[1] C.E. McDowell, D.P. Helmbold. *Debugging concurrent programs.* ACM Computing Surveys vol 21 no 4, Dec 1989.

[2] W. Cheung, J. Black, E. Manning. *A framework for distributed debugging.* IEEE Software, Jan 1990.

[3] M. Mackey. *Program replay in PVM.* Hewlett-Packard, Concurrent Computing Department, H.P. Laboratories, May 1993.

[6] Using the *select()* UNIX system call.

[4] A. Fagot, J, Chassin-de-Kergommeaux. *Optimized execution replay mechanism for RPC-based parallel programming models.* IMAG, Jul 1995.

[5] *ACM Workshop on Parallel and Distributed Debugging.* ACM SIGPLAN Notices vol 24 no 1, Jan 1988.

[6] C. Fidge. *Partial orders for parallel debugging.* ACM Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices vol 24 no 1, Jan 1988.

[7] *ACM/ONR Workshop on Parallel and Distributed Debugging.* ACM SIGPLAN Notices vol 28 no 12, Dec 1993.

[8] *ACM/ONR Workshop on Parallel and Distributed Debugging.* ACM SIGPLAN Notices vol 26 no 12, Dec 1991.

[9] T. Ludwig, R. Wismuller, V. Sunderam, A. Bode. *OMIS — on-line monitoring interface specification.* LRR-TUM, Technical Univ. of Munich, Germany, and Emory Univ. USA, Feb 1996.

[10] D.C. Marinescu, J.E. Lumpp, Jr., T.L. Casavant, H.J. Spiegel. *Models for monitoring and debugging tools for paralell and distributed software.* J. of Parallel and Distributed Computing, 9, 171–183, Jun 1990.

[11] P.S. Dodd, C.V. Ravishankar. *Monitoring and debugging distributed real-time programs.* Software—Practice and Experience, vol 22, no 10, Oct 1992.

[12] T.J. LeBlanc, J-M. Mellor-Crummey. *Debugging parallel programs with instant replay.* IEEE Trans. on Computers, vol C-36, no 4, Apr 1987.

[13] Y. Manabe, M. Imase. *Global conditions in debugging distributed programs.* J. of Parallel and Distributed Computing, 15, May 1992.

[14] J.J-P., Tsai, S.J.H. Yang, editors. *Monitoring and debugging of distributed real-time systems.* IEEE Computing Society Press, 1995.

[15] University of Westminster, UK. *Software Engineering for Parallel Processing.* Copernicus Programme, Contract CIPA-C193-0251, Progress Report no. 1, Oct 1994.

[16] G. Dózsa, T. Fadgyas, P. Kacsuk *GRAPNEL: A Graphical Programming Language for Parallel Programs* Proc. of uP'94: The Eight Symposium on Microcomputer and Microprocessor Applications, Budapest, Hungary, 1994.

[17] H. Krwaczyk, B. Wiszniewski *Structural Testing of Parallel Software in STEPS* Proc. of the 1st SEIHPC Workshop, COPERNICUS Programme, Braga, Portugal, 1996.

[18] J.C. Cunha, H. Krwaczyk, B. Wiszniewski, P. Mork, P. Kacsuk, E. Luque, L. Sutovska, L. Hluchy. *Monitoring and Debugging Distributed Memory Systems.* Proc. of uP'94: The Eight Symposium on Microcomputer and Microprocessor Applications, Budapest, Hungary, 1994.

[19] J.C. Cunha, J. Lourenço, T. Antão. *Integrating a debugging engine to the GRAPNEL environment.* HPCTI Project, COPERNICUS Programme, 3rd Progress Report, University of Westminster, 1996.

[20] J.C. Cunha. *Design of Parallel and Distributed Monitoring and Debugging Systems.* SEPP Project, COPERNICUS Programme, 4th Progress Report, University of Westminster, 1996.

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

[22] A, Beguelin, J.J. Dongarra, G.A. Geist, V.S. Sunderam. *Visualization and Debugging in a Heterogeneous Environment.* IEEE Computer, Vol 26, No. 6, 1993.

[23] R. Marques, J.C. Cunha. *PVM-Prolog: Parallel Logic Programming in the PVM System.* Procs. of the 1995 PVM User's Group Meeting, Pittsburgh, May 1995.

[24] P. Kacsuk, G. Dózsa, T. Fadgyas. *GRAPNEL: A Graphical Parallel Programming Language.* Journal of Systems Architecture, Special Issue on Parallel Software Engineering, 1996, No. 1.

[25] P. Kacsuk, G. Dózsa, T. Fadgyas. *Development of Graphical Parallel Programs in PVM Environments.* submitted to DAPSYS'96.

[26] S. Winter,P. Kacsuk. *Software Engineering for Parallel Processing.* Proc. of the 8th Symp. on Microcomputer and Microprocessor Applications, Budapest, 1994, pp. 285-293.

[27] W.F. Clocksin, C.C. Mellish. *Programming in Prolog.* Springer-Verlag, 1981.