Chapter 13

The DDBG Distributed Debugger

José C. Cunha, João Lourenço and Vítor Duarte

Abstract

This chapter presents the main issues involved in the design of the DDBG distributed debugger. DDBG provides basic support for state based debugging of distributed C/PVM processes. Due to its flexible architecture, DDBG enables the implementation of several debugging methodologies for deterministic re-execution and systematic state exploration. This is achieved through its integration with other tools in a parallel software development environment. The chapter describes how DDBG was integrated with two tools of the SEPP/HPCTI environment: the STEPS testing tool and the GRED graphical editor.

13.1 Introduction

As described in previous chapters, the SEPP/HPCTI projects have promoted the design and implementation of tools supporting visual graph-based parallel program development, including mapping and load-balancing, simulation, monitoring, testing and debugging tools. The main goal of these projects was to achieve an integrated environment that would ensure a suitable degree of consistency among the above mentioned tools.

As discussed in Chapter 5, the debugging activity relies on the observation and control of a computation in order to identify, locate and correct so-called program bugs. In a brief summary, the following issues characterize the main dimensions of the distributed debugging activity:

1. *Observation and control of distributed computations.* The construction of consistent global states of a distributed computation is a fundamental requirement to support the meaningful evaluation of local and global predicates, which involve state variables in multiple processes. Concerning control of the distributed computation, coordinated actions such as step-by-step execution and breakpoint marking must be applied to individual processes, as in sequential debuggers, and to collections of distributed processes.

2. *Program analysis and testing.* Distributed computations exhibit a very large space of alternative computation paths that should be explored during debugging. Deterministic re-execution schemes have a limitation here because they only allow to try the recorded trace that happened during a previous computation and this trace may be the wrong one to inspect. So, the identification

of the desired traces that should be inspected under re-execution should be the result of applying a testing tool. This aspect also relates to the ability to specify suitable global predicates that should be submitted to an evaluation by the underlying system. So a debugger needs to be complemented by program analysis and testing tools.

3. *User level interfacing, program behavior interpretation and visualization.* This includes several aspects:

(i) Due to the complexity of the distributed computations we need to provide high-level views to the user (high level debugging) that are close to the abstraction levels at which the application is specified or programmed. This is opposed to the low-level view of debugging where system or language (sometimes assembly-level) concepts are explicitly visible to the user.

(ii) Such views may be supported through the provision of graphical and visual interfaces. Graphical user interfaces (GUI) support the already conventional windows and menu based access to debugging commands and replies. Visual debugging is another aspect that concerns the suitable interpretation and visualization of the relevant events of a distributed computation, ranging from simple space-time diagrams to more sophisticated views offered by interactive visualization tools.

(iii) Internal consistency of the debugging tool means all the debugging functionalities (for inspection and control of the distributed computation) are accessible through the graphical and visual presentation interfaces as well as through text based command consoles and text based message displays. Also, the user should be able to selectively enable/disable such graphical/visual views for subsets of processes, communication channels, or any other components of a distributed program. Furthermore, it should be possible to consistently observe/control a distributed computation at the desired abstraction level, by changing from high level to low level debugging and vice versa.

The above dimensions cannot be supported by an isolated debugging tool as it is virtually impossible to anticipate all possible requirements posed by each user at each point in time. Several complementary tools should be considered to support each individual aspect and they should be able to cooperate with the debugging tool, as illustrated by the following simplified list:

1. *A debugging console* providing access to the control and inspection commands, in the form of a textual command line oriented interface, and/or in the form of a graphical user interface with a set of associated buttons, plus a text window to display the program source code, as well as several information displays for the computation status.

2. *A graphical display* for the representation of a higher level view of the program structure (e.g., a graph-based view) such that it is possible to perform debugging commands at a higher level of abstraction, and to hide a more detailed view of the internal state transitions in each process.

3. *A visualization tool* that displays the evolution of processes or threads in space-time diagrams.

4. *A testing tool* that allows the generation of specific testing scenarios identifying suspect computation paths that must be subject to a more detailed inspection under debugging control.

13.2 Design Issues for a Distributed Debugger

Two main requirements arise when designing a distributed debugging tool:

(i) Mechanisms for the observation and control of distributed computations. This includes two dimensions, depending on the level of observation that is required at each point during debugging. One level concerns the observation of individual process or thread states. The other level concerns the observation of global states.

(ii) Frameworks to support testing and debugging methodologies. This concerns the support provided by the distributed debugging tool to guide the user along the steps of identifying and precisely locating bugs.

There are two main possibilities to try to meet the above requirements:

(i) A self-contained autonomous debugging tool. It incorporates mechanisms and strategies for all the above aspects into a single debugging architecture. Such aspects are perhaps more easily made transparently accessed by the user which is an advantage of this approach. It may also more easily provide a uniform user interface. However, in general this kind solution is tied to a specific parallel or distributed programming model and/or to a specific debugging methodology (if any). Due to the self-contained design, it becomes difficult to adapt the tool to distinct abstractions and/or testing and debugging methodologies.

(ii) A minimal distributed debugging architecture with a facility for integration of extended services. This approach allows to separate, on one hand, the basic low level debugging mechanisms for observation and control, and the higher level mechanisms that may relate to higher level abstractions. On the other hand it allows to separate the mechanisms from the debugging methodologies or strategies that one wants to enforce in each case.

The DDBG distributed debugger that was developed within the SEPP project is an example of the second approach. The basic mechanisms supported by DDBG are of the following kinds:

(i) Observation and control of multiple distributed sequential processes.

(ii) Tool integration based on an interface library.

13.2.1 Observation and control of distributed processes

Concerning the observation and control mechanisms, DDBG offers the following main functionalities:

(i) Control of the debugging session. This includes commands to start or finish a debugging session, to put a process under debugger control, and to remove a process from the debugging environment.

(ii) Control of the process execution. This includes commands that directly control the execution path followed by a process, once it is under debugger control.

(iii) Process state inspection and modification. This includes commands to inspect the state of a process in well-defined points which are reached due to the occurrence of breakpoints or other types of events (process stopped or terminated). The information that can be accessed includes process status, variable and stack frame records, and source code information.

Such functionalities provide direct support to state based interactive debugging of distributed processes (see Chapter 5). All other aspects such as deterministic re-execution, systematic state exploration, and correctness predicate specification, must be provided as extended services and implemented through tool integration.

13.2.2 Tool integration

Besides text and graphic based user interfaces which give access to the debugging commands, DDBG allows direct access to its services through an interface library that can be linked to each client tool. Each client tool typically provides some high level debugging functionality and/or supports some testing and debugging methodology that in the end must rely upon DDBG basic mechanisms. So the concept underlying the DDBG design is to enable an open architecture on top of a low level built-in distributed debugging framework. This concept was the basis of our experimentation during the SEPP project, and several distinct kinds of tool integration were achieved with reasonable success, as described in the following sections.

In order to support easy integration of the debugger with other tools in a parallel software engineering environment, a well-defined debugging interface must be provided to be used by high-level tools, namely graphical interfaces and graphical program editors, runtime support systems for distinct parallel and distributed language models, and testing and high-level debugging tools.

Concerning such interface to high-level tools, the following aspects must be taken under consideration:

(i) *Concurrent access from multiple separate client tools.* Multiple tools can independently and concurrently issue debugging commands over the same target application. Thus they all share the same information concerning the program state and have the same abilities to issue inspection and control commands.

(ii) *Dynamic attachment and detachment of client tools to the debugging engine.* Client tools can join and exit the debugging process dynamically, having their own life cycle independent of the DDBG debugger life cycle.

(iii) *Support for heterogeneity.* Heterogeneity is supported at multiple levels: hardware, operating system, programming language and model, as the client-server architecture accepts plug-and-play node level debuggers that are used to access each individual target application process.

In order to allow access by distinct client tools, the DDBG interface provides a bidirectional interaction scheme supporting an asynchronous operation mode. In fact, many client tools such as editors and graphical interfaces exhibit an event-driven behavior. The debugging interface primitives that are invoked by each high-level tool must support a non-blocking semantics because some debugging commands don't provide an immediate answer. On the other hand, the communications interface must support the passing of the output information coming from such non-immediate debugging commands back to the user tool (e.g., a graphical editor). A simple solution to this problem is to provide a library function that allows the user tool to poll a communication channel that is associated with this interaction with the debugging system. An alternative solution is to support a facility for the handling of asynchronous events by the client tool such that the invocation of a previously specified handler can be triggered by the arrival of the debugging information.

Multiple simultaneous client connections should be supported by the debugging system, so that multiple cooperating tools can be accessing the debugging environment for inspection and control of the distributed program execution.

A mapping service of high-level process names onto low-level system process identifiers allows to integrate client tools which support distinct high level abstractions such symbolic virtual process identifiers. Conceptually, it is up to any client tool to interpret its abstract entities and convert them to low level DDBG entities. This can also be achieved by some intermediate tool, with the advantage of allowing the clear separation of concerns when designing the high level client tool. An example of this approach is described in Section 13.4.3.

As a consequence of the above aspects, DDBG has no built in fixed user interface. In SEPP, we have implemented a command line user interface giving access to all the debugging functionalities. We have also implemented an unsophisticated graphical user interface that is consistent with the user views being offered by other tools in the environment. As far as SEPP project was concerned, this graphical user interface only allowed selective inspection of the variables of each distributed process. This approach is opposite to the usual approach of having highly sophisticated graphical user interfaces for a parallel and distributed debugger (e.g. TotalView [7]), but lack of integration support mechanisms.

13.2.3 Characteristics of DDBG

How do we classify DDBG according to the dimensions of Chapter 5?

(i) *Debugging methodologies.* DDBG directly supports state based interactive debugging of distributed processes. It can support all other methodologies through tool integration. Within SEPP, we have implemented *systematic state exploration* of C/PVM programs using DDBG, through the integration of DDBG and the STEPS testing tool (see Chapter 16).

(ii) *Debugging at distinct phases of development.* DDBG directly supports on-line dynamic analysis and control of distributed processes. This can be used to observe and enforce specific testing scenarios. It can also be used to "manually" check dynamic program behavior, through the user interfaces. Off-line analysis is not directly supported by DDBG. However, these facilities can be integrated with DDBG.

(iii) *DDBG observation model.* DDBG is a state based distributed debugger. It supports observation and control of individual distributed processes. It can be used to support global observations and global control of distributed computations, through an adequate integration with other tools. Namely, deterministic re-execution and controlled re-execution facilities were achieved for C/PVM programs within SEPP, based on the tool integration of STEPS and DDBG.

(iv) *Debugging at multiple (hierarchical) levels of abstraction.* Due to the support of multiple concurrent client tools and to the minimal design of its architecture, DDBG naturally supports multiple levels of observation and control of a distributed computation. This is illustrated in the following through the integration of GRED (see Chapter 10) and DDBG.

13.3 The DDBG Parallel and Distributed Debugger

In this section we briefly describe the architecture and interface library which were developed for the DDBG distributed debugger within the SEPP project.

13.3.1 Architecture

Figure 13.1 illustrates the DDBG architecture, where three different types of processes are involved in a debugging session: client processes, DDBG processes and target application processes.

DDBG has a basic client-server architecture which follows the lines of the p2d2 design [8]. The reader may find a more detailed presentation of DDBG in [4]. A brief summary is presented here.

The client processes are depicted in Figure 13.1 as user interfaces or other user tools. These processes are linked to the *DDBG Library* that provides access to all DDBG debugging functionalities.

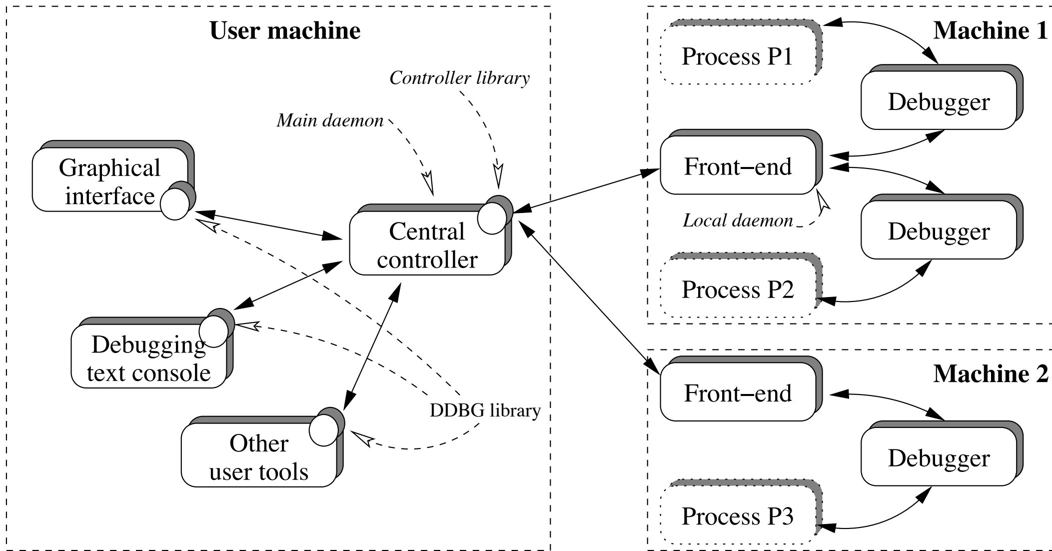

Figure 13.1: The DDBG distributed debugger

The target processes belong to the application being debugged. This application can have multiple processes spread on multiple nodes, with different hardware and operating systems. Heterogeneity concerns are handled by DDBG at the level of its internal communication layer (the SEPP implementation of this layer uses PVM for interprocess communication, and UNIX sockets for the interactions between each client tool and the central controller). Heterogeneity is also handled by allowing multiple possible types of local node debuggers to be integrated into DDBG architecture. SEPP implementation relied upon GNU GDB as the only local node debugger supported.

The DDBG architecture internally consists of several component processes:

(i) *Central Controller.* It coordinates the handling of the client requests, converts them into a set of commands and distributes them to the relevant local node debuggers. It is also responsible for processing the local node debuggers' replies, and sending them back to the client processes as returns to the calls of functions of the debugging library.

(ii) *Local Front-ends.* There is one of these processes in each node. Besides some local interpretation of the debugging commands, it locally distributes them to the local debuggers, and gets the answers back so that they mat be conveyed to the central controller.

(iii) *Local Node Debuggers.* A system-dependent sequential debugger, for a specific programming language and the underlying hardware. There is a local node debugger attached to each process of the target application processes, that applies the inspection and control commands to that process.

13.3.2 Interface Library

Any user tool can access the DDBG system as a client process that uses an interface library to interact with the central debugging controller. The interface library supports functions for the control of the DDBG system and for supporting the interfacing with other tools, and functions supporting

distributed process control and state inspection. The latter type of functions are currently adapted from identical functions provided by the GNU GDB debugger, but they operate upon multiple distributed processes. The functions for distributed process control and state inspection include support for debugging commands that control the execution of each individual process in a detailed way, including step by step execution, handling breakpoints and watchpoints, and displaying or modifying local process information (variables, status, stack frames, current breakpoints). A detailed definition of these functions is presented in [1, 2].

There are functions to support, respectively, the initialization and the cleanup of the debugging environment. The initialization also establishes a connection between each user tool and the central controller, that is used for further interaction with DDBG. It also sets up an interprocess communication channel that is used for the passing of delayed output information between the DDBG and the user tool. This channel can be inspected by invoking another interface function with a non-blocking semantics, corresponding to a design requirement that was discussed in a previous section.

There are also functions supporting the dynamic attachment and detachment of application processes to new debugger instances, as well as to obtain information about new components (e.g., newly spawned application processes) in the debugging environment.

In the implemented prototype for the debugging of C/PVM programs, an user application or tool may use specific *Process ID's* (strings) to identify the processes. In order to support the mapping between the user processes symbolic names and the PVM task names, a name mapping function is provided allowing to associate a *tid*, a PVM task identifier, to a given process identifier. This allows any of the library primitives, as well as the corresponding user consoles, to refer to string process identifiers, besides PVM *task ID's* (integers). This solution currently solves the name mapping problem and it is used by the current interface of the GRAPNEL and DDBG. A similar functionality was implemented for the STEPS-DDBG integration.

13.4 Interfacing the DDBG Debugger with Other Tools

In this section we briefly discuss our experimentation with the interfacing of the DDBG system and two high-level tools of a parallel software engineering environment: the GRED and the STEPS tools. More detailed descriptions of these tools and interfaces is given in Chapters 10 and 16 and in several references [3, 5, 6, 9–12].

13.4.1 User Interfaces

Graphical and text-oriented debugging user interfaces are two examples of client processes, not being part of DDBG by themselves, but which are included in the SEPP distribution. Besides a command line console that gives access to all the interface library functions, it is possible to interface any kind of graphical user interface to the DDBG system. The SEPP prototype provided a X-based window interface for the interactive display of selected process variables.

13.4.2 Graphical Debugging in GRED with the DDBG Debugger

The GRAPNEL model, a graph-based parallel programming language (see Chapter 10), supports a structured style for designing parallel applications. In order to allow the debugging commands and the output information from the debugger to be directly related to the GRAPNEL model, an integration of GRED and DDBG was developed in SEPP. The main goal was to ensure that only GRAPNEL abstractions should be handled by the user at this level.

Two main issues are considered in this integration:

(i) The design of the high-level user interface, with the specification of the adequate set of debugging commands, and its coherent integration within the GRED level abstractions. A detailed description of such interface can be found in [9]. The distinctive aspect is 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 [6, 10].

(ii) The design of the interface between GRED and DDBG. For each debugging action invoked on the GRED editor, corresponding DDBG primitives are invoked, and process names are converted as previously explained. GRED was linked to the DDBG library and so it became a client tool that could be integrated into the DDBG architecture.

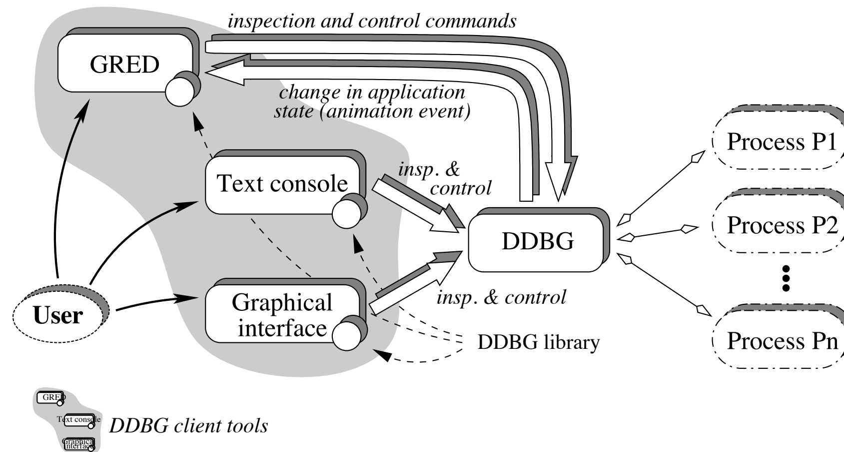

Figure 13.2: The Integration of GRED and DDBG

The GRED-DDBG interface (Figure 13.2) and its SEPP implementation relies upon an UNIX socket-based communication protocol to interact with the DDBG system, but this is hidden in the interface library functions that send commands to the debugger. Output debugging information is asynchronously passed back to GRED handler routines through a socket that is polled by the editor, in an event-driven mode. A DDBG interface library function allows GRED to get that information.

Two-level (or hierarchical) debugging is allowed, so that the user may switch between high level debugging through the GRED interface (see Chapter 10) for a structural level debugging view, to the low level debugging of individual processes, for a component level, textual debugging view of the C/PVM program, directly invoking DDBG commands through the DDBG user interface.

13.4.3 Tool Composition for Testing and Debugging

The SEPP project introduced an interesting approach to support testing and debugging methodologies. Instead of a monolithic approach based on a single testing-and-debugging tool, several specific tools were separately designed and then their developers worked together towards the combination of such tools.

Program analysis and testing of parallel programs were investigated by our partners of the Technical University of Gdansk. They have independently developed a testing tool called STEPS [11] (Chapter 16), that generates selected execution scenarios for a given parallel (C/PVM) program. After the generation of a testing scenario by the STEPS tool, and through a suitable integration with the debugger, it is possible to submit an 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 reproducible execution. The user is allowed to run a complete test scenario until the end or alternatively it is possible to follow a step by step execution controlled by breakpoints. This is achieved by converting the information associated with the specification of each testing scenario onto corresponding information and commands known to the DDBG debugger. Such conversion is supported by an interface component between the STEPS and DDBG tools.

One should note here that the main goal of tool composition or integration is to obtain a new functionality as a result of the integration of two distinct tools, each with its own functionality. Moreover this should be achieved with no change (or a minimal change) to each tool.

The above goal was achieved in the integration of STEPS and DDBG through the development of an interfacing component, the DEIPA tool, as described in Figure 13.3.

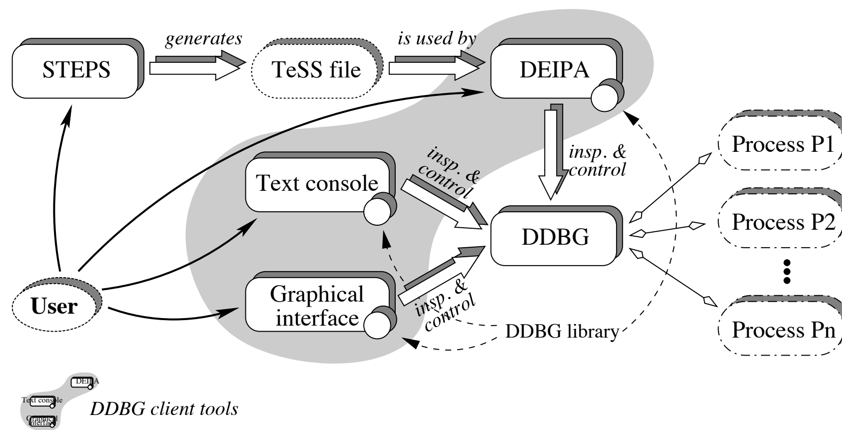

Figure 13.3: The DDBG distributed debugger

DEIPA

The DEIPA tool (Deterministic (Re-)Execution and Interactive Program Analysis) [12] has an interactive console allowing the user to issue the following types of commands:

(i) Opening and closing intermediate TeSS files containing the specification of sequences of global computation states that must be followed under actively controlled execution (see Chapter 16).

(ii) Running the program under the control of DDBG, until the next global breakpoint is reached or until the end. Global step-by-step execution from one global breakpoint to the next one is also allowed.

DEIPA interprets each line of the TeSS file and is responsible for the local interpretation of these commands (e.g., in case of open or close) or for the generation of corresponding sequences of DDBG commands, e.g., in case of the global step command.

The internals of the DEIPA tool, e.g., concerning the handling of virtual process identifiers, or the detailed interpretation of the TeSS file are not described here, but can be found in [12].

Observation of a distributed computation at the level of global states is possible, as the STEPS-DDBG integration does in fact implement global breakpoints. The DEIPA tool controls program execution until a global breakpoint is reached. Then the user can switch to the DDBG console and perform a state based observation of individual processes, using specific DDBG commands. This integration of DEIPA and DDBG is the result of the DDBG architecture allowing multiple concurrent tools.

13.5 Conclusions

We have shown how the DDBG distributed debugger has been designed under the influence of specific interfacing requirements posed by high-level parallel software development tools. We described the SEPP working prototype and its use to support the integration with the GRED and the STEPS tools.

Several significant results were obtained:

(i) High level graphical debugging of GRAPNEL programs was achieved through the implementation of a GRED debugging interface (see Chapter 10) on top of the DDBG debugger.

(ii) Hierarchical debugging of GRAPNEL programs was achieved by allowing simultaneous access to the GRED and the DDBG user interfaces, providing a facility to switch from structural, graphic based, views of the application, to process level, textual based views of individual application components.

(iii) Systematic state exploration of C/PVM programs is supported, based on a testing, active control and debugging approach (see Chapter 5, Chapter 9 and Chapter 16) that was obtained through the integration of STEPS and DDBG. Besides enabling the mentioned testing approach, this integration also ensures deterministic re-execution of the enforced paths.

(iv) Observation of a distributed computation (C/PVM) at the level of global states, as the STEPS-DDBG in fact implements global breakpoints, and followed by a state based observation of individual processes, using specific DDBG commands. The latter aspect is the result of the DDBG functionality of allowing multiple concurrent tools.

The DDBG experience showed the great importance of a flexible distributed debugging architecture that support the requirements for tool integration.

Further work includes improvements on the architecture, its intensive evaluation by real end users, and its integration into other parallel and distributed computing frameworks, namely to support the debugging of component based and metacomputing applications.

Acknowledgements

This work was partially supported by the Centre for Informatics and Information Technologies (CITI) and the Department of Informatics (DI) of FCT/UNL, by the PRAXIS XXI SETNA-ParComp

(Contract 2/2.1/TIT/1557/95), and by the French Ambassy — INRIA/Portuguese ICTTI and the Hungarian/Portuguese Governments cooperation protocols.

References

[1] J. C. Cunha, J. Lourenço, and T. Antão. A debugging engine for a parallel and distributed environment. In KFKI Hungarian Academy of Sciences, editor, *Proceedings of the 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS'96)*, pages 111–118, Misckolc, Hungary, October 1996.

[2] J. C. Cunha, J. Lourenço, and T. Antão. DDBG: A distributed debugger – user's guide. Technical report, Departamento de Informática, FCT-Universidade Nova de Lisboa, Portugal, 1996.

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

[4] J. C. Cunha, J. Lourenço, and T. Antão. An experiment in tool integration: the DDBG parallel and distributed debugger. *Euromicro Journal of Systems Architecture*, (11):897–907, 1999. Elsevier Science Press.

[5] G. Dozsa, P. Kacsuk, and T Fadgyas. Development of graphical parallel programs in PVM environments. In *Proceedings of the 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS'96)*, pages 33–40, Miskolc, Hungary, October 1996.

[6] G. Dózsa, T. Fadgyas, and P. Kacsuk. GRAPNEL: A graphical programming language for parallel programs. In *Eighth Symposium on Microcomputer and Micropocessor Applications*, pages 285–293. IEEE Press, October 1994.

[7] Etnus Inc., Framingham, MA. *TotalView User's Guide (v3.9.0)*, June 1999. http://www.etnus.com/.

[8] R. Hood. The p2d2 project: Building a portable distributed debugger. In *Proc. of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 127–136, Pennsylvania, USA, May 1996. ACM Press.

[9] P. Kacsuk, J. C. Cunha, G. Dózsa, J. Lourenço, T. Fadgyas, and T. Antão. A graphical development and debugging environment for parallel programs. *Parallel Computing*, 22(1997):1747–1770, 1997. Elsevier Science Press.

[10] P. Kacsuk, G. Dózsa, and T. Fadgyas. Designing parallel programs by the graphical language GRAPNEL. *Microprocessing and Microprogramming*, 41:625–643, 1996.

[11] H. Krawczyk and B. Wiszniewski. Object-oriented model of paralel programs. In *Proc. 4th EUROMICRO Workshop on Parallel and Distributed Processing*, pages 80–86, Braga, Portugal, 1996. IEEE Computer Society.

[12] J. Lourenço, J.C. Cunha, H. Krawczyk, P. Kuzora, M. Neyman, and B. Wiszniewski. An integrated testing and debugging environment for parallel and distributed programs. In *Proceedings of the 23rd EUROMICRO Conference (EUROMICRO'97)*, pages 291–298, Budapeste, Hungary, September 1997. IEEE Computer Society Press.