Computers and Artificial Intelligence, Vol. 13, 1995, 1–10

USING DDBG TO SUPPORT TESTING AND HIGH-LEVEL DEBUGGING INTERFACES

José C. Cunha
João Lourenço
Vítor Duarte

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

Abstract. This paper describes our experience with the design and implementation of a distributed debugger for C/PVM programs within the scope of the SEPP and HPCTI Copernicus projects. These projects aimed at the development of an integrated parallel software engineering environment based on a high-level graphical parallel programming model (GRAPNEL) and a set of associated tools supporting graphical edition, compilation, simulated and real parallel execution, testing, debugging, performance monitoring, mapping, and load balancing. We discuss how the development of the debugging tool was strongly influenced by the requirements posed by other tools in the environment, namely support for high-level graphical debugging of GRAPNEL programs, and support for the integration of static and dynamic analysis tools. We describe the functionalities of the DDBG debugger and its internal architecture, and discuss its integration with two separate tools in the SEPP/HPCTI environment: the GRED graphical editor for GRAPNEL programs, and the STEPS testing tool for C/PVM programs.

Keywords. Tool integration, parallel and distributed debugging

1 INTRODUCTION

The development of adequate parallel software engineering environments became a very important issue in recent years. In the SEPP and HPCTI Copernicus projects [19], although each partner was responsible for the development of individual tools, a

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

major concern was their coherent integration into a graphical development environment. The whole development cycle is supported, including graphical editing, compilation, simulation and real parallel execution on top of PVM [1]. Associated tools for testing, debugging, performance monitoring, mapping, and load balancing are also supported.

In this paper we discuss the main issues involved in the design and implementation of the DDBG distributed debugger, and its integration into the GRADE environment. GRADE [13, 12] consists of a set of development tools built around the GRAPNEL model for graphical parallel programming. GRAPNEL [12] is a graph-based visual programming model supporting the structured design of parallel applications. In order to provide an adequate view to the user, all development tools should refer to the abstractions provided by GRAPNEL. For example, as far as debugging is concerned, the inspection and control of the computation state should refer to the GRAPNEL program components and structures, and should be integrated with the graphical user interface supported by GRADE. However, at the same time, debugging at a lower level should also be supported, allowing the user to inspect and control the C/PVM-based components that are part of the GRAPNEL program. This requires the debugging tool to provide an interface to the GRED [13] graphical editor, while it should also give direct access to the C/PVM programming level.

Another important aspect of a parallel software engineering environment is to achieve a close integration between static analysis and dynamic analysis tools. In fact, due to the great complexity of parallel computations, a tool is required to help the user in the generation of testing scenarios that depend on the parallel program structure and the dynamic process interactions. One can obtain further information on program behavior and inspect specific computation paths in greater detail if a debugger can be coupled to a testing tool.

The above interfacing requirements were satisfied by the distributed process-level DDBG debugger. The prototype of the DDBG system allows the inspection and control of distributed C/PVM processes.

In section 2 we describe the DDBG debugger. In section 3 we discuss its integration with the GRED [13] graphical editor and the STEPS [15] testing tool. Then, we discuss related work and identify ongoing research.

2 THE DDBG DEBUGGER

2.1 Design Issues

The basic functionalities of a distributed debugger concern state inspection and control of individual processes or threads, and coordination-level abstractions such as deterministic re-execution, global distributed breakpoints, and evaluation of global

predicates. Such functionalities strongly depend upon each programming and computational model, so it is desirable to identify a set of basic debugging mechanisms (e.g. [2]), and use them in order to implement higher level functionalities.

The DDBG debugger [4] allows an user or another tool to control and inspect multiple distributed processes. There are the following classes of debugging primitives that are supported by an interface library:

1. Control of the debugging session. This includes commands to start or finish a debugging session, to put/remove a process under/from debugger control.
2. Control of the process execution. This includes commands that directly control the execution path followed by a process, once it is under debugger control.
3. Process state inspection and modification. This includes commands to inspect the state of a process in well-defined points that 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.

In order to support easy experimentation with debugging services for distinct computational models, a flexible software architecture is required. This architecture should be able to integrate and manage distinct types of process-level or thread-level debuggers, which depend on each hardware and operating system platform, and on each programming model. In the following, we describe the DDBG architecture.

2.2 The DDBG Architecture

The DDBG (**D**istributed **D**e**B**u**G**ger) [4] tool provides a set of debugging functionalities for distributed programs written in *C* on the *PVM* system [1]:

1. *Simultaneous access from multiple (high-level) client tools.* Multiple tools can (independently) issue debugging commands over the same target application.
2. *To dynamically attach and detach client tools to the debugging engine.* Client tools can "enter" and "leave" the debugging activity dynamically, having their own life cycle independent of the DDBG debugger life cycle.
3. *Global view of the system being debugged.* All the client tools share the same information concerning the program state and have the same abilities to issue inspection and control commands.
4. *Support for heterogeneity.* Heterogeneity is supported at hardware, operating system and programming language levels.
5. *Easy integration with client tools.* Tool integration functionalities are included in the debugger specification.

Three different types of processes can take part in a debugging session with DDBG (Figure 1):

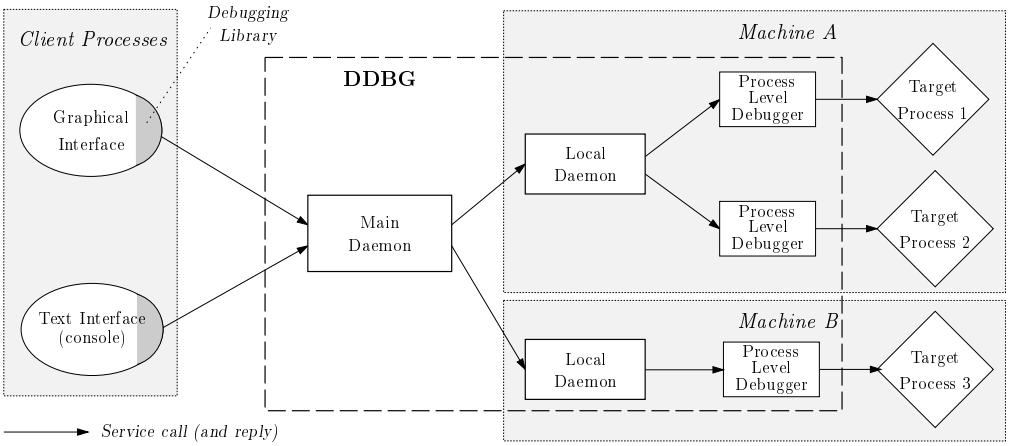

Fig. 1. The DDBG distributed debugger

1. *Client Processes (CP).* These processes use a *Debugging Library (DL)* that provides access to DDBG.
2. *DDBG processes.* DDBG includes the following components:
   (a) *Main Daemon (MD).* It acts as a coordinator, and is responsible for receiving the CP requests, convert them into a set of commands and sending them to the relevant *Process-level Debuggers (PLDs)* (see below). The MD also receives and processes the PLD replies, and sends them back to the CP through return parameters of the DL.
   (b) *Local Daemons (LD).* There is a LD in each machine, doing some local interpretation of the debugging commands and working as a multiplexer, forwarding these commands and controlling all the PLDs running on that machine.
   (c) *Process-level Debugger (PLD).* A system-dependent sequential debugger, that supports the programming language and the underlying hardware. There is a PLD attached to each target process of the *Target Application.*
   (d) *Debugging Library (DL).* This is included into any client process in order to provide a set of functions that give access to the DDBG primitives.
   (e) *Graphical-* and *Text-oriented debugging user interfaces (UI).* These are two examples of CP that were developed and integrated into DDBG, providing two different debugging user interfaces.
3. *Target Application Processes.* The application being debugged can have multiple processes spread on multiple machines.

There are alternative designs to DDBG, depending on how the responsibilities are distributed among its processes. In a pure hierarchical design the MD is responsible for the interpretation of debugging commands received from the CP, i.e. it

performs all the necessary conversions, it forwards the actual PLD-level commands to the corresponding LD and sends the replies back to the CP. In this solution, the LD processes are just responsible for contacting the right PLD and send its answer back to the MD. There are several disadvantages in such kind of design:

1. *High MD complexity.* The MD process must also support multiple concurrent client connections and so it needs to manage a lot of information concerning pending requests.
2. *Hard to support heterogeneity.* In heterogeneous distributed computing an application can be decomposed into multiple parts, each running on distinct sequential or parallel machines, with distinct PLD processes. This design requires the MD process to process all command and data interpretations.
3. *Reduced flexibility.* As the MD becomes very complex, it is more difficult to integrate new services into the architecture.

A more flexible design would distribute the responsibility for actual command and data interpretation to each LD, and let the MD do only the interfacing to the client tools. Each LD can then independently perform its tasks, according to the specific characteristics of each local PD. This is a better solution to support heterogeneous debugging, as well as to support extended services, because the required modifications are associated with specific LD processes. The functions left to the MD are the interfacing with client tools, the management of multiple connections to the debugger, and the presentation of global views to the user concerning the global state of the distributed computation. As a result of our experience, a new architecture that reflects the above design options is under development where the MD is a multi-threaded process with associated services [3].

2.3 Support for Tool Integration

DDBG provides a well-defined interface that can be accessed by high-level tools of a parallel development environment. The debugging library that is linked to each client tool uses a well-defined protocol to communicate with the DDBG main daemon. This was built using TCP/IP sockets, and supports two-way interaction between the DDBG and the client tools. It also provides an asynchronous operation mode to support event-driven interactions with the client tools.

3 USING DDBG IN SEPP/HPCTI

3.1 Experiences With Tool Integration

It is very difficult to provide full integration among a large set of development tools such as the ones found in the SEPP/HPCTI projects. This is due to the need to

offer consistent views at several levels: multiple user interfaces, tool behavior, tool interaction, and tool composition. Even in our project, where many of the tools were jointly developed from the beginning, a full integration was a difficult goal to achieve because it required a tight collaborative effort between the involved partners, concerning their design options, and the associated working environments (e.g. with distinct graphical user interfaces, and operating system platforms). However, we have obtained a reasonable degree of integration between several tools, and have opened the way to possible further integrations [5, 6, 7, 18]. One of the distinctive goals of our approach when designing and implementing DDBG was to provide a platform supporting easy experimentation with tool integration. Two main experiments were performed concerning the interfacing of DDBG with two tools with very distinct functionalities.

3.1.1 Integrating DDBG into the GRADE Environment

GRADE (**GRA**pnel **D**evelopment **E**nvironment) is an integrated environment for the development of parallel programs in the GRAPNEL programming language. The GRAPNEL language is a graph-based visual parallel programming language, that supports a structured style for designing parallel applications, and is supported by the GRED graphical editor [12]. In this section we focus on the close integration of DDBG and GRED (**GR**apnel **ED**itor) in order to support debugging of GRAPNEL programs.

In such integrated environment the user should work mainly with the same abstractions that were used during program development[1].This requires highlighting the entities in the graphical representation and their corresponding lines of source code in the textual program representation.

For such high-level debugging of GRAPNEL programs, each debugging action of the GRED graphical editor is mapped into a set of DDBG debugging actions. Such commands are then sent to DDBG which in turn replies with DDBG-level answers that must be converted into the corresponding action in the GRED visual editor. In order to support possibly long-execution commands, such as "*proceed until next breakpoint is reached*", an asynchronous (event) notification feature has been integrated into DDBG and is used by GRED to detect the completion of such kind of commands. The integration of DDBG into GRADE is detailed in [11].

3.1.2 Integrating DDBG with STEPS

The STEPS testing tool [14, 15] allows to identify potential critical paths in a C/PVM program. The DDBG debugger can inspect and control the program behavior, helping in the localization of programs bugs and their causes.

[1] This is a general concept, as it makes no sense to develop a program using the *C* programming language and then debug this same program only at *assembler* level.

When integrating these tools, one must ensure that the program will run and behave as expected, and so the composition of the testing and the debugging tools starts by re-executing the target applications and forcing each process to follow some specific path until a pre-determined point. It is necessary to ensure that the application will reach the critical points previously identified by the testing tool and will stop in a consistent state (also called a "*Global Breakpoint*"). At this potential critical point, the user can enter an interactive debugging session, using both the graphical and the command-line debugging interfaces, and issuing inspection and control commands directed at any of the target processes.

The DEIPA (**D**eterministic (re-)**E**xecution and **I**nteractive **P**rogram **A**nalysis) tool supports the integration of STEPS and DDBG. DEIPA recognizes and processes the output of the STEPS tool that is kept in the TeSS file. This file includes all the information that is required by the controlled execution of the parallel computation paths that were generated by STEPS. Namely, it defines a sequence of *global breakpoints*. A global breakpoint is a set of breakpoints, one for each individual application process. This information is analyzed and converted into (a set of) commands for the DDBG tool. 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 illustrated in Figure 2.

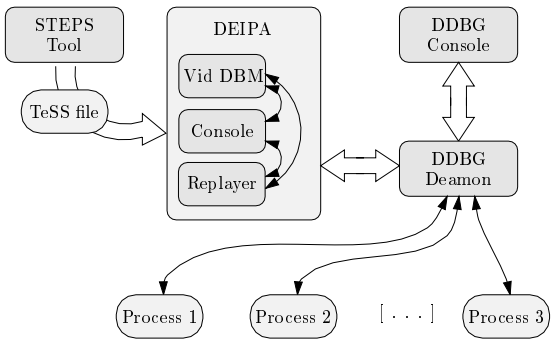

Fig. 2. The integration of STEPS and DDBG

The Console module. It supports the user interface to the DEIPA tool. This interface is based on a command-line console allowing to load a TeSS file and control the (re-)execution of the target application. Some commands allow to control the DEIPA tool, e.g. `load` (to load a new TeSS file), while others are converted into sets of DDBG commands and applied to the target application, e.g. `step` (to proceed into the next *Global Breakpoint*).

The Vid Database Management module. During static analysis, STEPS uses symbolic names for processes. This module implements the mapping from symbolic to the real (PVM) process identifiers that are used by DDBG during real execution.

The Replayer module. This is responsible for the mapping of DEIPA commands into DDBG commands, e.g. converting a DEIPA step command into a set of DDBG set_breakpoint and continue_execution commands. It also supports the required process control, e.g. setting variables in an **if-then-else** statement, so that each process is forced to follow the path that is specified by the TeSS file.

In [16] there is a complete discussion of DDBG and STEPS integration. The DDBG support for multiple simultaneous client tools has been used here. The DEIPA tool is used to control the execution of all the application processes as explained, and the DDBG console and/or GUI can be used to provide a more detailed inspection and modification of each individual process.

4 RELATED WORK

There are many current efforts on the field of parallel and distributed debugging [2, 3, 8, 9, 10, 17]. Concerning the debugging primitives, the High-Performance Debugging Forum (HPDF) [2] is a collaborative effort aiming to define a standard for parallel debuggers. Concerning tool interfacing, the On-line Monitoring Interface Specification (OMIS) [17] aims to define an open interface to software development tools, such as e.g. debuggers and performance evaluation tools. It precisely defines the communication protocols and formats of the exchanged information. Concerning the debugger architecture, the p2d2 system [8] is a distributed debugger that also uses a client-server approach, with a well defined interface, promoting portability by isolating the system dependent code into a debugger server.

Our work with DDBG will provide a flexible architecture allowing us to experiment with the proposed HPDF specifications. It will also allow us to experiment with improved forms of tool integration‘.

5 CONCLUSIONS AND FUTURE WORK

We have discussed the DDBG debugger, and how it was used to offer debugging functionalities to other tools in a parallel software engineering environment. The DDBG functionalities were found adequate to support the requirements posed by other tools in the GRADE environment. This experience allowed us to identify several directions to improve current DDBG functionalities. This includes the support of process-level and thread-level debugging, as well as the support of coordination-level services, such as distributed global breakpoints, and evaluation of global predicates. Our approach can be used to support a testing and debugging methodology that allow the user to systematically inspect specific computation paths. The DEIPA tool uses DDBG to support such controlled execution of a parallel program. Additionally, recently we have implemented a deterministic re-execution mechanism in DDBG that relies on a previously recorded execution trace. We are working on the integration of both approaches (controlled execution and trace–driven) so that we

can better assist the user in the localization of the bugs occurring in a distributed program.

Concerning tool interaction and integration, this includes more flexible support for interfacing the debugger with distinct concurrent tools and support their coordination. Concerning heterogeneity, this includes supporting other parallel and distributed platforms besides PVM, based on MPI and WindowsNT. There is a need for systems that can work in several distributed platforms, can be used by several tools, and can be extended and adapted to new environments and functionalities. In ongoing work, the DDBG architecture is evolving toward a layered architecture that tries to meet such requirements.

REFERENCES

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

[2] Brown, J.—Francioni, J.—Pancake, C.: White Paper on Formation of the High Performance Debugging Forum. Available in "http://www.ptools.org/hpdf/meetings/-mar97/whitepaper.html", 1997.

[3] Cunha, J. C.—Lourenço, J.—Vieira, J.—Moscão, B.—Pereira, D.: A Framework to Support Parallel and Distributed Debugging. Proceedings of the International Conference on High-Performance Computing and Networking (HPCN'98). Amsterdam, The Netherlands, 1998.

[4] Cunha, J. C.—Lourenço, J.—Antão, T.: A Debugging Engine for a Parallel and Distributed Environment. Proceedings of the 1^{st} Austrian-Hungarian Workshop on Distributed and Parallel Systems (DAPSYS'96). Miskolc, Hungary, 1996.

[5] Delaitre, T. et al.: EDPEPPS: An Environment for the Design and Performance Evaluation of Portable Parallel Software. Proc. of the 3rd SEIHPC Workshop. Madrid, Spain, 1998.

[6] Delaitre, T.—Justo, G.—Spies, F.—Winter, S.: A Graphical Toolset for Simulation Modeling of Parallel Systems. Parallel Computing Journal, 22, 1997, pp. 1823–1836.

[7] Hluchý, L.—Dobrucký, M.—Astaloš, J.: Hybrid approach to task allocation in distributed systems. Lecture Notes in Computer Science 1277. Springer. 1997, pp. 210–216.

[8] Hood, R.: The p2d2 Project: Building a Portable Distributed Debugger. Proceedings of the 2^{nd} Symposium on Parallel and Distributed Tools (SPDT'96). Philadelphia PA, USA, 1996.

[9] Kacsuk, P.: Macrostep-by-macrostep Debugging of Message Passing Parallel Programs. Accepted in IASTED PDCN'98, Las Vegas, USA, 1998.

[10] Kacsuk, P.: Systematic Testing and Debugging of Parallel Programs by a Macrostep Debugger. Submitted to DAPSYS'98. Budapest, Hungary, 1998.

[11] Kacsuk, P.—Cunha, J. C.—Dózsa, G.— Lourenço, J.—Fadgyas, T.—Antão, T.: A Graphical Development and Debugging Environment for Parallel Programs. Parallel Computing Journal, 22, 1997, pp. 1747–1770.

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

[13] Kacsuk, P.—Dózsa, G.—Lovas, R.—Fadgyas, T.: Enhancing GRADE Towards a Professional Parallel Programming Environment. Proc. of the 3rd SEIHPC Workshop. Madrid, Spain, 1998.

[14] Krawczyk, H.—Wiszniewski, B.: Interactive Testing Tool for Parallel Programs. Software Enginneering for Parallel and Distributed Systems. I. Jelly, I. Gorton, and P. Croll, Eds. Chapman & Hall, London, UK, 1996, pp. 98–109.

[15] Krwaczyk, H.—Wiszniewski, B.: Structural Testing of Parallel Software in STEPS. Proceedings of the 1st SEIHPC Workshop. Braga, Portugal, 1996.

[16] Lourenço J.—Cunha, J. C.—Krawczyk, H.—Kuzora, P.—Neyman, M.—Wiszniewski, B.: An Integrated Testing and Debugging Environment for Parallel and Distributed Programs. Proceedings of the 23^{rd} Euromicro Conference. Budapest, Hungary, 1997.

[17] Ludwig, T.—Wismuller, R.— Sunderam, V.— Bode, A.: OMIS — On-Line Monitoring Interface Specification (Version 2.0). Tech. rep., LRR-TUM, Munich, Germany, July 1997.

[18] Luque, E.—Ripoll, A.—Cortés, A.—Margalef, T.: A Distributed Diffusion Method for Dynamic Load Balancing on Parallel Computers. Proc. of the EUROMICRO Workshop on Parallel and Distributed Processing. San Remo, Italy, 1995.

[19] Winter, S.—Kacsuk, P.: Software Engineering for Parallel Processing. Proc. of the 8th Symp. on Microcomputers and Microprocessor Applications, Budapest, Hungary, 1994.

José C. Cunha was born in Viseu, Portugal. He got the Diploma in Electrical Engineering from Instituto Superior Tecnico in 1975, the Diploma in Informatics Enginering in 1979, and a PhD in Computer Science from Universidade Nova de Lisboa (FCT/UNL) in 1989. He is Associate Professor in the Computer Science Department of FCT/UNL. He is the head of the Computational Systems and Architecture Group and of the Parallel and Distributed Processing Group. He coordinates research in distributed languages, tools, and heterogeneous environments, and has been publishing regularly in these topics.

João Lourenço was born in Alcobaça, Portugal, in 1966. He graduated in Informatics Engineering in 1991, and got the M.Sc. degree in Informatics Engineering in 1995 from FCT/UNL. He is currently a Research Assistant at the Computer Science Department of FCT/UNL and is preparing his Ph.D. on parallel and distributed debugging.

Vítor Duarte was born in 1966, graduated in Informatics Engineering in 1990 and got the M.Sc. degree in Informatics Engineering in 1995 from FCT/UNL. He is a Research Assistant at the Computer Science Department of FCT/UNL and is preparing his Ph.D. on monitoring and visualization of distributed systems.