A Framework to Support Parallel and Distributed Debugging

José C. Cunha, João Lourenço
João Vieira, Bruno Moscão, and Daniel Pereira

{jcc, jml}@di.fct.unl.pt
{jpdv, bmoscao, dlp}@asc.di.fct.unl.pt
Departamento de Informática, Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, Portugal

Abstract. We discuss debugging prototypes that can easily support new functionalities, depending on the requirements of high-level computational models, and allowing a coherent integration with other tools in a software engineering environment. Concerning the first aspect, we propose a framework that identifies two distinct levels of functionalities that should be supported by a parallel and distributed debugger using: a process and thread-level, and a coordination level concerning sets of processes or threads. An incremental approach is used to effectively develop prototypes that support both functionalities. Concerning the second aspect, we discuss how the interfacing with other tools has influenced the design of a process-level debugging interface (PDBG) and a distributed monitoring and control layer called (DAMS).

1 Introduction

Debugging is centered on two main aspects, namely the observation of the computation state, and the need to exercise some modification and control upon the computation, to identify and correct program bugs. While the state of a sequential computation can easily be determined and controlled by a conventional, process-level debugger, the same is not true of the state of a distributed computation. Solutions to this problem must be incorporated into a distributed debugger, requiring a mechanism to evaluate local and global predicates, which involve state variables in multiple processes. Concerning control of the distributed computation, actions such as step-by-step execution and breakpoint marking must be applied to individual processes, as in conventional debuggers, and to sets of distributed processes. Debugging is more difficult for concurrent models because of non-deterministic executions, due to the arbitrary interleaving of multiple concurrent activities, and their dynamic interactions. Although several deterministic re-execution techniques have been proposed supporting cyclic and interactive debugging of concurrent programs [LMC78], they only solve a part of the problem. This is due to the large complexity of real concurrent applications, with many processes and many dynamic interactions, which implies a very large space of alternative computation paths that must be tried during debugging. In

fact, deterministic re-execution has a limitation because it only allows us to try each recorded trace in turn. As it is obviously impossible to exhaustively search all of those paths, we must assure that we are recording and re-executing the "right" traces. The identification of the desired traces that should be inspected under re-execution must be left to the user under the guidance of a testing tool. We argue that a debugger should always be complemented with a "companion" program analysis and testing tool. Although testing is beyond the scope of this paper [LCH+97] it puts an important requirement on the design of a debugger, because it demands specific mechanisms for tool interfacing [CLA96b]. This also applies to the integration of a debugger and a visualization tool, or of a debugger and a graphical editor for a visual language [KCD+98].

Tool Integration. Recently, much research has concentrated on software engineering environments for parallel and distributed applications. It is recognized the need of a coherent integration among all the tools and user interfaces in such environments. In the context of research projects [S+94] on software engineering for parallel processing, our team has developed a debugger for distributed applications (`DDBG` [CLA96a]). One of its basic requirements was the ability to integrate easily with tools from other partners, such as the `GRED` tool for the development of programs in the `GRAPNEL` visual programming language [KCD+98], and the `STEPS` testing tool [LCH+97]. `DDBG` provides the following functionalities: a bi-directional interaction where the client issues inspection/control commands to the debugger, and gets the replies; a mapping of user symbolic process names to system-level names; command-line (text-oriented), and graphical user interfaces; support for multiple simultaneous client processes.

Heterogeneity. Multiple components of a distributed application may be programmed in different languages and may spawn through a variety of different machines, operating systems and communication protocols. Besides this heterogeneity, we should be able to reuse, interconnect, and simultaneous control, distinct pre-existing component level debuggers (*i.e.* process- or thread-level). This is important as such debuggers are typically highly system dependent, and their development is time consuming. The `DDBG` prototype supported the debugging of distributed *C/PVM* programs but its internal architecture and implementation would not allow easy integration of new aspects. In this paper we propose a new debugging support framework with new features for tool integration, and we describe a new architecture and implementation of a process-level debugger.

Organization of the Paper. First we propose a framework for parallel and distributed debugging. Section 3 describes a process-level debugging interface. Section 4 presents a distributed monitoring architecture (`DAMS`) and Section 5 shows its use to implement the `PDBG` debugger.

2 A Framework to Support Parallel and Distributed Debugging

A computational model defines processes, threads, their groupings, and interactions. This suggests two levels of debugging functionalities:

1. *The component-level.* This level defines state inspection and control of each individual component (a process or a thread).
2. *The coordination-level.* This involves state inspection and control of sets of components, and their interactions.

This view allows the incremental design of the debugger, *i.e.* by first developing component-level functionalities, and then coordination level functionalities. While the former are dependent on the component-level debuggers, the latter should be adaptable according to the user needs. Our goal is to define a basic set of mechanisms and low-level functionalities that allow us to build such coordination level services. So in the first step we built the two bottom layers (levels 1 and 2) in our abstract debugging architecture:

1. *A layer supporting monitoring and control functionalities.* (`DAMS`).
2. *A layer supporting component-level functionalities* acting upon processes and threads, and supported as Service Modules in `DAMS`.

The development of coordination-level functionalities concerns the establishing global views and global actions, defined in terms of sets of processes and sets of threads. As these high-level functionalities are very close to the semantics of the distributed application, we provide only the basic mechanisms to support the user-level definition of such global views and actions. In the next incremental step we build the upper levels (layers 3 and 4) of the abstract debugging architecture:

3. *Coordination-level basic mechanisms* support the above high-level services.
4. *Coordination-level services* are defined at the user-level, as specific services with semanticaly-dependent abstractions for each application.

The discussion of the coordination level (layers 3 and 4) is beyound the scope of this paper. Regarding the component-level functionalities for state inspection and control (layer 2), we describe the `PDBG` and the `DAMS`.

3 A Specification of a Process Debugging Interface (`PDBG`)

Client processes can access the `PDBG` interface in order to debug distributed processes.

3.1 Synchronous and Asynchronous Interactions

A synchronous call interface supports the client requests. The invoker waits until a completion status is returned by `PDBG`, reporting success or failure (*e.g.* one cannot "*Continue*" a currently running or a terminated process). A synchronous mode of operation is necessary because there are well-defined sequences of commands that may be issued by a client ("*controller*"), so that the completion

status of each request allows its invoker to make correct assumptions concerning the new state of the target ("*controlled*") process. However, there is also a strong motivation for asynchronous operations. In general, the client must be able to catch well-defined events that occur during the execution of the target processes, *e.g.* when a breakpoint was placed on the target process, then a "*Continue*" command was issued, and the client must be notified that the target process has reached the specified point. Such events may be the (expected logical) result of some requests from the client process, or they may result from (possibly unexpected) exceptional conditions that should also be inspected by the client, *e.g.* the process receives a system-generated signal. Additionally, the environment may encompass several concurrent client and target processes which exhibit asynchronous interactions as a result of each request that is issued by some controller process. For example, besides a controller and the target processes, there may also be a set of "*observers*" *e.g.* when a controller issues debugging commands and a graphical interface and visualization tool generates an animated view of the ongoing computation. In order to support a coherent integration between the debugger and the visualizer an asynchronous event notification mechanism is required so that the visualization tool knows about the evolution of the controller commands and their outcome in the target computation.

Event Handling. By default, events are not delivered to any client. By calling an interface primitive, the invoker declares its interest in being notified of any events of a specified type (event), related to state transitions occurring in the processes that were explicitly identified. On event occurrence, a specified handler is invoked in the context of the invoker.[1] On event notification, the handler receives a structure as an argument which identifies the event type, and provides general information, namely a unique (system-generated) event identifier, plus information on each event subclass, concerning the process state, or the name, input arguments and results of the `PDBG` calls.

There are two kinds of event classes: (i) events associated with target process state transitions; (ii) events associated with client interactions with the `PDBG` system. The first is required to support debugging at the process level. The associated events are called *Exception events.* The second is related to the interaction of client tools with the debugging environment so they are associated with the calls made to the `PDBG`, *i.e.* their requests and the `PDBG` replies. The associated events are called *Request events.* This allows to support a coherent tool integration in a parallel software engineering environment, as well as coordination services involving multiple concurrent tools and the debugger. Some tools can register their interest in getting information concerning actions being performed as a result of commands issued by other tools.

Exception events. The following process states are identified as significant to the definition of the semantics of the `PDBG` primitives. If it's **detached,** the process is unknown. If it's **ready,** it has been created in the `ready_to_run`

[1] The implementation of handler invocation is dependent on the programming model being single or multi-threaded, and is not part of the `PDBG` interface definition.

state, under PDBG, and can start **running:** under PDBG, and then it can become **stopped** by a PDBG command or due to the occurrence of some exception. Finally it can become **terminated** by a PDBG or system command, or because it has reached the end of program.

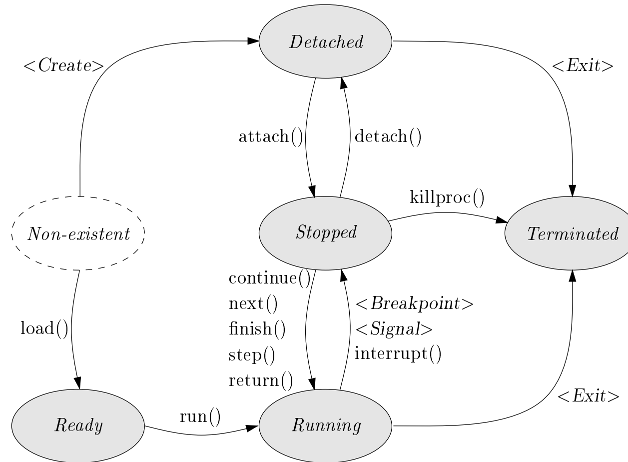

Fig. 1. The process state transition

In the Figure 1 the state transitions are labelled by the names of the associated PDBG or system primitives, or by the associated exceptions.

Request Events. Events of the *Request* class are associated with the invocation and completion of the PDBG primitives.

3.2 The PDBG Interface primitives

The following classes of PDBG interface primitives allow a client tool to control and inspect multiple distributed processes:

Control of the Debugging Session: control of the debugging session; put a process under debugger control (a new process that will execute a given program file is created and put in the ready state, or an existent detached process is put under debugging control in the stopped state; remove a process from debugging control; kill a stopped process.

Process Execution: directly control of the execution path followed by an individual process, once it is known to the debugger; start running a ready process; send a signal to stop a running process; resume the execution of a stopped process.

Process State Inspection and Modification: inspect the state of an individual process in well-defined points, which are reached as a result of breakpoints or due to the occurrence of certain types of events (process stopped or terminated). The information that can be accessed concerns process status, variable and stack frame records, as well as source code information: breakpoints setting and deleting; expression and variable setting and evaluation; stack frame inspection and selection; source code information.

4 The DAMS Architecture

The goal of the DAMS architecture is to provide a basic layer for distributed monitoring and control that clearly separates the low-level mechanisms and the support for application-level services such as debugging, profiling, and resource management.

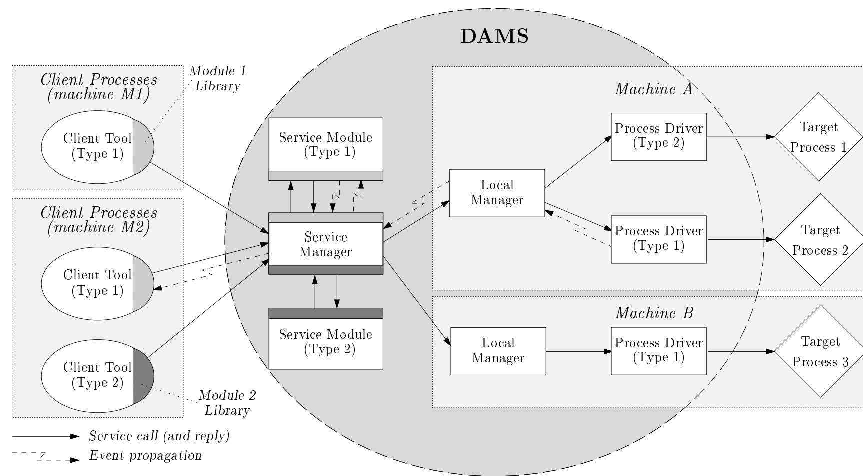

Fig. 2. The DAMS architecture

The DAMS architecture (see Figure 2) is composed as a hierarchy of processes of the following types: the Service Manager (SM) and its Service Modules (Sml); the Local Managers (LM), one per physical node; the Drivers (D), each associated with a Target Process.

The Service Manager . In order to keep DAMS flexible and extensible, the Service Manager does not handle any application-specific commands. It forwards them instead to associated Service Modules which are actually linked to the Service Manager at system generation time. Each Service Module encapsulates all the specific functionalities required by a certain service, such

as for debugging, profiling, or resource management. The Service Manager is implemented as a multi-threaded process that has two distinct programming interfaces, one facing the application and the other facing the Service Modules:

1. The application invokes commands of a library which accesses the SM through the Service Manager Application Programming Interface (SM API) to access a certain service, *e.g.* for debugging.
2. The Service Module Application Programming Interface (Sml API) is used by the SM process in order to communicate with the service module responsible for activating the application-specific commands.

The application side library commands can get the identification of the available service modules, prepare request buffers, and then make calls to the service modules. Such requests are sent as messages, through the SM API, and are then forwarded to the corresponding module by the Service Manager. The application side library commands can be hidden inside an user-level library which offers a more transparent system-call based interface, *e.g.* for the `PDBG` debugging commands.

The Local Managers, the Drivers, and the Target Processes. On each node of the physical architecture there is a Local Manager which is responsible for the communication with the Service Manager and the management of local processes. Each Target Process that runs on a node is controlled and monitored by an associated Driver process. Drivers are dynamically launched and attached to target processes. They are responsible for the direct enforcement of application-level commands to each associated target process, using a well-defined protocol for handling requests and answers. In order to support each individual application-level functionality, one must define a pair (Service Module, Driver), *e.g.* a debugging service module and a debugging driver. The Local Managers are responsible for the management of the driver processes, acting as intermediate between the Service Manager and the Drivers, *i.e.* they forward messages coming from the SM to the Drivers, and handle the passing of replies and event notifications to the Service Manager. They also provide status information about the processes on each node.

Internal `DAMS` Protocols. In order to keep the `DAMS` architecture flexible and easily extensible, there is a well-defined set of protocols ruling the interactions between the `DAMS` components. An architecture independent message-passing library is used in `DAMS` that has been mapped onto the PVM communication primitives in the implemented prototype. We are working on portings to MPI-1 and to Windows NT. The `DAMS` design supports a built-in asynchronous event mechanism from the target processes to the client processes which is used to support `PDBG` events.

Heterogeneity. `DAMS` is neutral regarding the computational model of the target processes. All such dependences are under the responsibility of the Service Modules and their Driver processes. As a result, `DAMS` offers a very expressive environment to monitor and control heterogenous application components.

5 An Implementation of the PDBG Debugging Interface

The PDBG interface supports debugging of sequential *C* programs, as well as distributed programs, *e.g.* *C/PVM* and *C/MPI* programs, or heterogeneous programs with sequential *C* and parallel *C/PVM* or *C/MPI* components. PDBG has been implemented as a service in the DAMS system. The Debugging Module is a DAMS module that provides the debugger interface and allows accessing the target processes. For each target process of the distributed application being debugged, there is a Debugging Driver which is responsible for inspection and control operations. Requests to the Service Manager are forwarded to the Local Manager in the right machine, which in turn forwards the commands to the Debugging Driver.

An Illustration of the Execution of a PDBG Command. When a function from the debugging library, *e.g.* `dbg_breakset()`, is invoked by a client process, the following interactions are generated:

1. The function arguments are converted into a structure and sent as a message to the Service Manager (SM) through the DAMS communication library.
2. The SM forwards the message to the Debugging Module (DM), where it's unpacked to be processed. Here, the requested service is identified and the corresponding set of *debugging driver commands* is generated. For each command, the relevant data is packed into another structure and sent to the Debugging Driver (DD) through the SM and the Local Manager (LM), where it's unpacked.
3. For each request that arrives to the DD, a new set of *process-level debugging commands* is generated and sent to the debugger in sequence. When all commands are processed, the relevant data is packed into another structure and sent back to the DM, again through the LM and the SM, where it's unpacked if needed.
4. The DD processes the data, if needed, and sends/forwards the (new) data to the client debugging library. Here the data is unpacked and returned as *return parameters*.

Current Status. The current status consists of a definition and implementation of the process-based functionalities (the PDBG interface). We are working on the definition of coordination support, concerning global views and actions (distributed replay and breakpoints, and global predicates). A prototype of PDBG was implemented supporting the debugging of distributed processes on a heterogeneous LAN including PC's with Linux, and a cluster of Alpha processors with OSF/1, interconnected by FDDI links. The process-level debugger currently in use is the GNU *gdb* which is highly portable and tested, but there is a considerable drawback: the *gdb* is very heavy. We have measured between 60 and 90% of the elapsed time in a command execution is spent inside *gdb*.

6 Conclusions and Related Work

Our current effort towards the definition of expressive and generic debugging functionalities is focussed on the following main aspects:

1. A precisely defined low-level interface consisting of commands for component-level state inspection and control.
2. A framework to define coordination-level services for distributed state inspection and control, and deterministic re-execution.
3. A specification of the mechanisms to support the interfacing of the debugger with other tools.
4. An underlying software architecture that addresses the heterogeneity issue.

Aspects (1) and (2) concern the fundamental debugging issues of distributed state inspection and control and deterministic re-execution. They depend upon the basic computational model being considered, concerning processes, threads and sets or groups of such basic entities. A most significant effort has recently been launched with the goal of establishing a common definition of debugging functionalities — the HPDF initiative [BFP97]. The work presented in this paper is complementary of efforts such as the HPDF initiative because we expect the HPDF debugging specifications will be easily implemented within our framework. We also expect that the HPDF proposal will strongly influence the evolution of our `PDBG` interface as well as a thread-based interface. The main goal of our work is to promote an effective experimentation with the design and implementation issues by incrementally building prototypes. So, we are following an approach based on defining a hierarchy of debugging support abstractions such that we can address the above issues in an incremental fashion, *i.e.* first at component-level and then at a coordination-level. We have paid special attention to aspect (3) concerning tool interfacing due to our previous experience with the development of parallel software engineering environments [CL97]. In fact, in modern development application-oriented environments one must consider a more general situation where multiple controller processes (client tools) as well as multiple observer processes (also client tools) jointly operate on a set of target processes. A basic asynchronous event notification mechanism is one important aspect to support the required coordination of such processes. The above aspects have influenced the design of the `DAMS` monitoring architecture which is also addressing the heterogeneity issue. Due to the clear separation of functionalities in `DAMS`, one can simultaneous control multiple heterogeneous component-level debuggers, *i.e.* process-based, single-threaded or multi-threaded. We also can provide multiple coexisting client tools such as graphical and visualization interfaces, and other services such as testing tools, that can interact with the debugging tool [LCH+97]. Some of the aspects in `DAMS` can be related to another ongoing project — the OMIS initiative [LWSB97]. However, the main distinction is that `DAMS` does not aim to provide a built-in monitoring interface with a standard definition. Instead, it provides a very flexible organization to define new services, including the definition of the user interface (through Service Modules), and the interfacing with component- or system-level services (through the

Drivers). Besides the support of distributed heterogenous debugging, `DAMS` is also being used to support a resource management service to control the execution of a heterogeneous parallel computational steering environment [MC97].

Acknowledgments. Thanks are due to the reviewers for their comments, to EC COPERNICUS SEPP (CIPA-C193-0251) and HPCTI (CP-93-5383), the Portuguese CIÊNCIA and PRAXIS XXI PROLOPPE (3/3.1/TIT/24/94), SETNA-ParComp (2/2.1/TIT/1557/95), and DEC EERP PADIPRO (P-005).

References

[BFP97] J. Brown, J. Francioni, and C. Pancake. White paper on formation of the high performance debugging forum. Available in "http://www.ptools.org/hpdf/meetings/mar97/whitepaper.html", February 1997.

[CL97] J. Cunha and J. Lourenço. An Experiment in Tool Integration: the DDBG Parallel and Distributed Debugger. *EUROMICRO Journal of Systems Architecture, 2nd Special Issue on Tools and Environments for Parallel Processing*, 1997.

[CLA96a] J. C. Cunha, J. Lourenço, and T. Antão. A Debugging Engine for a Parallel and Distributed Environment. In Hungarian Academy of Sciences-KFKI, editor, *Proceedings of DAPSYS'96, 1st Austrian-Hungarian Workshop on Distributed and Parallel Systems*, Misckolc, Hungary, October 1996.

[CLA96b] J. C. Cunha, J. Lourenço, and T. Antão. A Distributed Debugging Tool for a Parallel Software Engeneering Environment. In *EPTM'96, 1st European Parallel Tools Meeting*, Paris, France, October 1996. ONERA.

[KCD+98] P. Kacsuk, J. Cunha, G. Dózsa, J. Lourenço, T. Fadgyas, and T. Antão. A Graphical Development and Debugging Environment for Parallel Programs. *Parallel Computing*, (22):1747–1770, February 1998.

[LCH+97] J. Lourenço, J. Cunha, H.Krawczyk, P. Kuzora, M. Neyman, and B. Wiszniewski. An Integrated Testing and Debugging Environment for Parallel and Distributed Programs. In *EUROMICRO 97, Proceedings of the 23rd EUROMICRO Conference*, pages 291–298, Budapest, Hungary, September 1997. IEEE Computer Society.

[LMC78] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Transactions on Cumputers*, C–36(4):471–482, April 1978.

[LWSB97] T. Ludwig, R. Wismuller, V. Sunderam, and A. Bode. OMIS — On-Line Monitoring Interface Specification (Version 2.0). Technical report, Lehrstuhl fur Informatik, Technical University of Munich (LRR-TUM), Munich, Germany, July 1997.

[MC97] P. D. Medeiros and J. C. Cunha. Interconnecting Multiple Heterogeneous Parallel Application Components. In *Proceedings of EuroPar'97*, Passau, Germany, August 1997.

[S+94] S.Winter et al. Software Engineering for Parallel Processing, copernicus programme. Progress report 1, University of Westminster, October 1994.

[S+96] S.Winter et al. High Performance Computing Tools for Industry, copernicus programme. Progress report 3, University of Westminster, April 1996.