Fiddle: a Flexible Distributed Debugging Architecture

João Lourenço José C. Cunha

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

Abstract. In the recent past, multiple techniques and tools have been proposed and contributed to improve the distributed debugging functionalities, in several distinct aspects, such as handling the non-determinism, allowing cyclic interactive debugging of parallel programs, and providing more user-friendly interfaces. However, most of these tools are tied to a specific programming language and provide rigid graphical user interfaces. So they cannot easily adapt to support distinct abstraction levels or user interfaces. They also don't provide adequate support for cooperation with other tools in a software engineering environment. In this paper we discuss several dimensions which may contribute to develop more flexible distributed debuggers. We describe Fiddle, a distributed debugging tool which aims at overcoming some of the above limitations.

1 Introduction

The debugging of parallel and distributed applications still largely relies upon ad-hoc techniques such as the ones based on *printf* statements. Such ad-hoc approaches have the main advantage of providing easy and flexible customization of the displayed information according to the user goals, but they are very limited concerning when, how and what information to display. This makes such approaches not acceptable in general.

Intensive research has been conducted in the field of distributed debugging in the recent past, with several techniques and tools being proposed [1–3], and a few debugging tools reaching a commercial status [28, 30]. Overall, such techniques and tools have contributed to improve the distributed debugging functionalities, in several distinct aspects, such as handling the non-determinism, allowing cyclic interactive debugging of parallel programs, and providing more user-friendly interfaces.

Most of the existing tools for the debugging of parallel and distributed applications are tied to a specific programming language and provide a rigid, albeit sometimes sophisticated, graphical user interface. In general, such tools cannot be easily adapted to support other abstraction levels for application development, or to allow access from other user interfaces. They also don't allow other software development tools having access to the distributed debugger interface, so that, for example, interesting forms of tool cooperation could be achieved in order to meet specific user requirements.

We claim there is a need to develop more flexible distributed debuggers, such that multiple abstraction levels, multiple distributed debugging methodologies, and multiple

user and tool interfaces can be supported. In the rest of this paper we first discuss the main dimensions which should be addressed by a flexible distributed debugger. Then we introduce the Fiddle distributed debugger as an approach to meet those requirements, and briefly mention several types of tool interaction concerning the use of Fiddle in a parallel software engineering environment. Finally we discuss ongoing work and summarize the distinctive characteristics of this approach.

2 Towards Flexible Distributed Debugging

In order to better identify the main requirements for the design of a flexible distributed debugging tool, we consider the following main dimensions: i) Multiple abstraction levels; ii) Multiple programming methodologies; iii) Multiple user and tool debugging interfaces.

Multiple distributed debugging abstraction levels

The extra complexity of parallel and distributed applications has led to the development of higher-level programming languages. Some of them are textual languages, while others are based on graphical entities or more abstract concepts. Such higher-level models rely upon e.g., automatic translators to generate the low-level code which is executed by the underlying runtime software/hardware platform. These models have increased the gap to the target languages which are typically supported by a distributed debugger. The same happens concerning the abstractions of lower level communication libraries like PVM [5] or MPI [13], and operating system calls. In order to bridge such semantic gap, a debugger should be flexible and able to relate to the application level abstractions.

Many of the actual distributed debuggers support imperative languages, such as C and Fortran, and object-oriented languages, such as C++ or Java. They also allow to inspect and control the communication abstractions, e.g., message queues at lower-level layers [8]. But this is not enough. The distributed debugger should also be extensible so that it can easily adapt to new programming languages and models. The same reasoning applies to the requirement for supporting debugging at multiple abstraction levels, such as source code, visual (graphical) entities, or user defined program concepts.

Multiple distributed debugging methodologies

Among the diversity of parallel software development tools, one can find tools focusing on performance evaluating and tuning [4, 15, 18, 24, 25], and other focusing on program correction aspects [4, 14, 28, 30]. One can also find tools providing distinct operation modes: i) Tools which may operate on-line with a running distributed application in order to provide an interactive user interface [4,11,14,30]; ii) Tools which can be used to enforce a specific behavior upon the distributed application by controlling its execution, e.g., as described in some kind of script file [22, 26, 27]; iii) Tools which may deal with the information collected by a tracing tool as described in a log file [4, 15, 16, 18, 25].

The on-line modes i) and ii) are mainly used for correctness debugging. The off-line or post-mortem mode iii) is mainly used for performance evaluation and/or visualization

purposes, but is also applied to debugging based on a trace and replay approach [20]. The distributed debugger should be flexible in order to enable this diversity of operation modes, and their possible interactions.

The above mentioned distinct operation modes will be put into use by several alternative/complementary debugging methodologies. The basic support which is typically required of a distributed debugger is to allow interactive debugging of remote processes. This allows to act upon individual processes (or threads), but it doesn't help much concerning the handling of non-determinism. In order to support reproducible behavior, the distributed debugger should allow the implementation of a trace and replay approach for interactive cyclic debugging. Furthermore, the distributed debugger should provide support for a systematic state exploration of the distributed computation space, even if such a search is semi-automatic and user-driven [22].

All of the above aspects have great impact upon the design of a flexible distributed debugger architecture. On one hand, this concerns the basic built-in mechanisms which should be provided. On the other hand, it puts strong requirements upon the support for extensible services and for concurrent tool interaction. For example, a systematic state exploration will usually require the interfacing (and cooperation) between an interactive testing tool and the distributed debugger. In order to address the latter issue, we discuss the required support for multiple distributed debugging interfaces, whether they directly refer to human computer interfaces, or they are activated by other software tools.

Multiple user and tool interfaces

Existing distributed debuggers can be classified according to the degree of flexibility they provide regarding user and tool interfacing. Modern distributed debuggers such as TotalView provide sophisticated graphical user interfaces (GUI), and allow the user to observe and control a computation from a set of pre-defined views. This is an important approach, mainly if such graphical interfaces are supported at the required user abstraction levels and for the specific parallel and distributed programming models used for application development (e.g. PVM or MPI).

The main problem with the above approach is that typically the GUI is the only possible way to access the distributed debugger services. If there is no function-call based interface (API), which can be accessed from any separate software tool, then it is not possible to implement more advanced forms of tool interaction and cooperation.

The proposal for standard distributed debugging interfaces has been the focus of recent research [6]. In the meanwhile, several efforts have tried to provide flexible infrastructures for tool interfacing and integration. Most of these efforts are strongly related to the development of parallel software engineering environments [7, 10, 17]. A related effort, albeit with a focus on monitoring infrastructure, is the OMIS [23] proposal for a standard monitoring interface.

Beyond the provision of a well-defined API, a distributed debugger framework should support mechanisms for coordinating the interactions among multiple concurrent tools. This requirement has been recently recognized as of great importance to enable further development of dynamic environments based upon the cooperation among multiple concurrent tools which have on-line access to a distributed computation.

The architecture of a flexible distributed debugger

All of the above mentioned dimensions should have influence upon the design of the architecture of a distributed debugger:

- *Software architecture.* From this perspective distributed debuggers can be classified in two main types: monolithic and client-server. Several of the existing distributed debuggers and related software development tools belong to the first class. This has severe consequences upon their maintenance, and also upon their extensibility and adaptability. A client-server approach, as followed in [11, 14, 15], clearly separates the debugging user interface from the debugging engine. This allows changes to be confined to the relevant sub-part, which is the client if it deals with the user interface, or the server if it deals with the debugging functionalities. This eases the adaptability of the distributed debugger to new demands.
- *Heterogeneity.* Existing distributed debuggers provide heterogeneity support at distinct levels: hardware, operating system, programming language (e.g., imperative, object-oriented), and distributed programming paradigm (e.g., shared-memory, message passing). The first two are a common characteristic of the major debuggers. The imperative and object-oriented languages are also in general simultaneously supported by distributed debuggers [14, 15, 30]. The support for both distributed-memory and shared-memory models also requires a neutral distributed debugging architecture which can in general be achieved using the client-server approach.
- *User and tool interfacing.* In a monolithic approach, the interfacing defined (graphical or text based) is planned beforehand according to a specific set of debugging functionalities. So it has limited capabilities for extension and/or adaptation. On the other hand, a client-server approach allows fully customizable interfaces to be developed anew and more easily integrated into the distributed debugger architecture.

3 The Fiddle approach

Research following from our experiences in the participation of the EU Copernicus projects SEPP and HPCTI [10] has led to the development of Fiddle.

3.1 Fiddle Characteristics

Fiddle [21] is a distributed debugging engine, independent from the user interface, which has the following characteristics:

- *On-line interactive correctness debugging.* Fiddle can be interactively accessed by a human user through a textual or graphical user interface. Its debugging services can equally be accessed by any other software tool, acting as a Fiddle client;
- *Support of high-level user-definable abstractions.* Access to Fiddle is based on an API which is directly available to C and C++ programs. Access to Fiddle is also possible from visual parallel programming languages, allowing a parallel application to be specified in terms of a set of graphical entities (e.g., for application components and their interconnections). After code generation to a lower level language

such as C, it would not be desirable for the user to perform program debugging at such a low level. Instead, the user should be able to debug in terms of the same visual entities and high-level concepts which were used for application development. The Fiddle architecture enables such kind of adaptation, as it can incorporate a high-level debugging interface which understands the user-level abstractions, and is responsible for their mapping onto Fiddle basic debugging functions;

– *Many clients / many servers.* Multiple clients can connect simultaneously to Fiddle and have access to the same set of target processes. Fiddle uses smart daemons, spread over the multiple nodes where the target application is being executed, to do the heavy part of the debugging work, relying on a central daemon for synchronization and interconnection with the client tools. By distributing the workload by the various nodes, reaction time and global performance are not compromised. This software architecture also scales well as the number of nodes increases;
– *Full heterogeneity support.* Fiddle architecture is independent of the hardware and operating system platforms as it relies on node debuggers to perform the system dependent commands upon the target processes. Fiddle is also neutral regarding the distributed programming paradigm of the target application, i.e., shared-memory or distributed-memory, as the specific semantics of such models must be encapsulated as Fiddle services, which are not part of the Fiddle core architecture;
– *Tool synchronization.* By allowing multiple concurrent client tools to access a common set of target processes, Fiddle needs to provide some basic support for tool synchronization. This can be achieved in two ways: i) As all requests made by client tools are controlled by a central daemon, Fiddle is able to avoid certain interferences among these tools; ii) Tools can also be notified about events originated by other tools, thus allowing them to react and coordinate their actions (see Sec. 4). In general, however, the cooperation among multiple tools will require other forms of tool coordination, and no basic support is provided for such functionalities, as they are dependent on each application software development environment. However, some specific tool coordination services can be integrated into Fiddle so that, for example, multiple tools can get consistent views of a shared target application;
– *Easy integration in Parallel Software Engineering Environments.* The event-based synchronization mechanism provided by Fiddle can be used to support interaction and synchronization between Fiddle debugging client tools and other tools in the environment, e.g., on-line program visualization tools.

3.2 Fiddle Software Architecture

Fiddle is structured as a hierarchy of five functional layers. Each layer provides a set of debugging services which may be accessed through an interface library. Any layer may be used directly by a client tool which finds its set of services adequate for its needs. Layers are also used indirectly, as the hierarchical structure of Fiddle's software architecture implies that each layer $\mathcal{L}_{i\ (i>0)}$ is a direct client of layer $\mathcal{L}_{i-1}$ (see Fig. 1). In this figure, Layer 3_m has two clients (tools $CT_2^{3_m}$ and $CT_1^{3_m}$), Layer 2_m has also two clients (tool $CT_1^{2_m}$ and Layer 3_m), and each of the remaining layers is shown with only one client, in the layer immediately above.

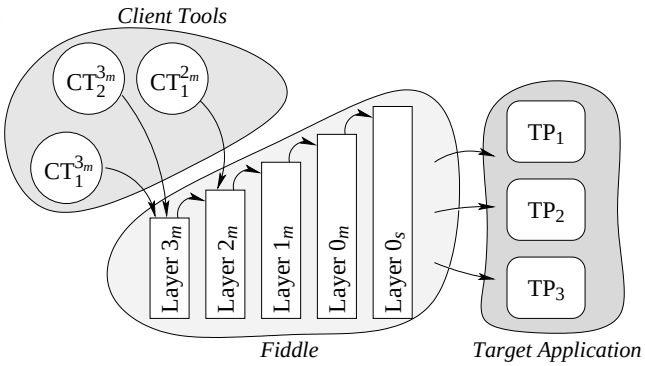

Fig. 1. Fiddle's Layered Architecture

A service requested by a Layer 3_m client tool will be processed and transferred to the successively underlying layers, until Layer 0_s is reached. At this point, the request is applied to the target process. The result of such service request is also successively processed and transferred to the upper layers until the client tool gets the reply.

There is a minimum set of functionalities common to all Fiddle layers, namely:

- *Inspect/control multi-threaded target processes.* Fiddle provides a set of debugging services to act upon threads within a multi-threaded process. These services will be active only if the node debugger being used on a specific target process is able to deal with multi-threaded processes;
- *Inspect/control multiple target processes simultaneously.* Any client tool may use Fiddle services to inspect and control multiple processes simultaneously. Except for Layer 0_s and Layer 0_m, the target process may also reside in remote nodes;
- *Support for client tool(s).* Some layers accept only one client tool while some others accept multiple client tools operating simultaneously over the same target application.

These common functionalities and the layer-specific ones described below are supported by the software architecture shown in Fig. 2.

Layer 0_s This layer implements a set of local debugging services. A client tool for this layer must be the only one acting upon the set of local single- or multi-threaded target processes.

The following components are known to this layer: i) *Target processes*: a subset of processes that compose the target application and are being executed in the local node; ii) *Node debuggers*: associated to each target process there is a node debugger which is responsible for acting upon it; iii) *Layer 0_s library*: provides single-threaded access to the debugging functionalities; iv) *Client tool*: this must be a single-threaded tool.

Layer 0_m This layer extends Layer 0_s to provide support for multi-threaded client tools. Client tools may use threads to control multiple target process and interact simultaneously with the user.

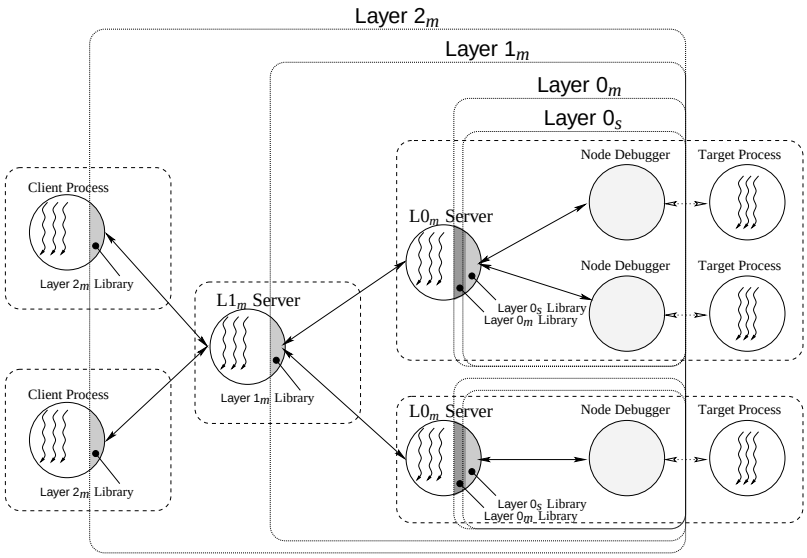

Fig. 2. The Fiddle software architecture

Layer 1$_m$ This layer extends Layer 0$_m$ to provide support for remote target processes. The target application can spread over multiple nodes and Fiddle can be used to debug local or remote target processes.

The components known to this layer are: 1. *Target processes*: running in the local or remote nodes; 2. *Node debuggers*: associated to each target process there is a node debugger; 3. *L0$_m$ Server.* the daemon is a Layer 0$_m$ client tool and acts as a gateway between the Layer 1$_m$ client tool and the target application. 4. *Layer 1$_m$ Library*: provides thread-safe access to local and remote debugging functionalities; 5. *Client tool*: Layer 1$_m$ client tools may transparently access local and remote target processes, relying on *L0$_m$ Server* daemons to act as tool representatives in each node.

Layer 2$_m$ This layer extends Layer 1$_m$ to provide support for multiple simultaneously client tools. These tools may concurrently request Fiddle's debugging services. The software architecture of Layer 2$_m$ includes an instance of Layer 1$_m$, plus a daemon, the *L1$_m$ Server*. This daemon is a Layer 1$_m$ client tool, which multiplexes the service requests from the multiple client tools to be submitted to Layer 1$_m$. It also demultiplexes the corresponding replies back to the clients..

Layer 3$_m$ This layer extends Layer 2$_m$ to provide an event-based interaction between Fiddle and the client tools. In contrast to the lower layers, a thread that invokes a method in the *Layer 3$_m$ Library* doesn't block waiting for its reply. Instead, it receives a *Request Identifier* which will be used later to process the reply. When the service is executed, its success status and data (the *reply*) is sent to the client as an event. These events may be processed by the client tool in two different ways: i) *Asynchronous mode.* The general

methodology to process Fiddle events is to define a handler. When a service is executed by Fiddle, a new thread will be created to execute the handler. A structure describing the results of the requested service is passed as an argument to the handling function, together with the *Request Identifier*. ii) *Synchronous mode.* In this mode, Fiddle keeps the notification of the event pending until the client tool explicitly requests it by invoking a Fiddle primitive.

4 Tool Integration and Composition

The flexibility provided by Fiddle, concerning multiple abstraction levels, multiple debugging methodologies and multiple user and tool interfaces, is being assessed through experimental work in the scope of several ongoing projects on distributed debugging:

- *FGI (Fiddle Graphical Interface).* Fiddle basic software distribution includes a command-line oriented debugging interface. Recently, a new project started for the development of a Graphical User Interface to access Fiddle. FGI aims to make full use of Fiddle capabilities, by providing a source-level view of the target application, and possibly cooperating with other tools which present higher-level views of the same application;
- *Deipa (Deterministic Execution and Interactive Program Analysis).* The use of an interactive *testing tool* which partially automates the identification and localization of suspect program regions can improve the process of developing correct programs. Deipa is an interactive steering tool that uses Fiddle to drive the program execution path according to a specification produced by STEPS [19] (an interactive testing tool developed at TUG, Gdansk, Poland). Deipa cooperates with other Fiddle client tools, such as FGI, to allow localized program state inspection and control. With Deipa the user may direct the program execution to a specific testing point, and then another tool, such as FGI, can be used to inspect and fine-control the application processes. When satisfied, the user may reuse Deipa to proceed execution until the next testing point;
- *PADI (Parallel Debugger Interface).* PADI [29] (developed at UFRGS, Rio Grande do Sul, Brasil) is a distributed debugging GUI, written in Java, which uses Fiddle as the distributed debugging engine, in order to implement debugging operations on groups or individual processes;
- *On-line Visualization and Debugging.* Research is under way to support the coordination of the on-line observation of a distributed application, as provided by a parallel program visualizer Pajé [15], and the debugging actions of Fiddle. Both tools were independently developed so they must now be adapted to achieve close cooperation. For example, whenever a process under debugging reaches a breakpoint and stops, then such a state transition must be accordingly updated by Pajé and reflected on its on-line visualization. On the other hand, if Pajé shows that a given process is blocked waiting for a message, we may be interested in selecting the process and having Fiddle automatically stopping the process, and selecting the source line containing the message reception code, and refreshing (or possibly opening) a source-level debugging GUI.

5 Conclusions and Future Work

The Fiddle distributed debugger is a result of our research on distributed debugging for the past 6 years [11,12]. It currently provides support for debugging multi-threaded/multi-process distributed applications. Its distinctive characteristics are the incremental design as a succession of layers, and its strong focus on the tool cooperation aspects. Fiddle current prototype fully implements Layer 0_s to Layer 2_m, while Layer 3_m is under development. It runs under Linux, uses GDB as the node debugger, and has a C/C++ API.

In order to evaluate and improve the support provided by Fiddle concerning flexible tool composition and integration in a Parallel Software Engineering Environment, there is experimental work under way. This is helping us to assess the design options, through the implementation of several case studies. Ongoing work also includes the development of a Java-based API.

As part of future work we plan to integrate Fiddle into the DAMS distributed monitoring architecture [9] as a *distributed debugging service*, in order to allow more generic and extensible tool cooperation and integration. We also plan to address large scale issues on distributed debugging for cluster computing environments.

Acknowledgments. The work reported in this paper was partially supported by the PRAXIS XXI Programme (SETNA Project), by the CITI (Centre for Informatics and Information Technology of FCT/UNL), and by the cooperation protocol ICCTI/French Embassy in Portugal.

References

1. *Proc. of the ACM Workshop on Parallel and Distributed Debugging*, volume 24 of *ACM SIGPLAN Notices*. ACM Press, January 1988.
2. *Proc. of the ACM Workshop on Parallel and Distributed Debugging*, volume 26 of *ACM SIGPLAN Notices*. ACM Press, 1991.
3. *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, volume 28 of *ACM SIGPLAN Notices*. ACM Press, 1993.
4. Don Allen et al. The Prism programming environment. In *Proc. of Supercomputer Debugging Workshop*, pages 1–7, Albuquerque, New Mexico, November 1991.
5. A. Beguelin et al. A user's guide to PVM parallel virtual machine. Technical Report ORNL/TM-118266, Oak Ridge National Laboratory, Tennessee, 1991.
6. 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.
7. Christian Clémençon et al. Annai scalable run-time support for interactive debugging and performance analysis of large-scale parallel programs. In *Proc. of EuroPar'96*, volume 1123 of *LNCS*, pages 64–69. Springer, August 1996.
8. J. Cownie and W. Gropp. A standard interface for debugger access to message queue information in MPI. In *Proc. of the* 6^{th} *EuroPVM/MPI*, volume 1697 of *LNCS*, pages 51–58. Springer, 1999.
9. J. C. Cunha and V. Duarte. Monitoring PVM programs using the DAMS approach. In *Proc.* 5^{th} *Euro PVM/MPI*, volume 1497 of *LNCS*, pages 273–280, 1998.

10. J. C. Cunha, P. Kacsuk, and S. Winter, editors. *Parallel Program Development for Cluster Computing: Methodology, Tools and Integrated Environment*. Nova Science Publishers, 2000.
11. J. C. Cunha, J. Lourenço, and T. Antão. An experiment in tool integration: the DDBG parallel and distributed debugger. *Euromicro Journal of Systems Architecture*, 45(11):897–907, 1999. Elsevier Science Press.
12. J. C. Cunha, J. Lourenço, J. Vieira, B. Moscão, and D. Pereira. A framework to support parallel and distributed debugging. In *Proc. of HPCN'98*, volume 1401 of *LNCS*, pages 708–717. Springer, April 1998.
13. MPI Forum. *MPI-2: Extensions to the message-passing interface*. Univ. of Tennessee, 1997. http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.
14. R. Hood. The p2d2 project: Building a portable distributed debugger. In *Proc. of the* 2^{nd} *Symposium on Parallel and Distributed Tools*, Philadelphia PA, USA, 1996. ACM.
15. J. C. Kergommeaux and B. O. Stein. Pajé: An extensible environment for visualizing multi-threaded programs executions. In *Proc. Euro-Par 2000*, volume 1900 of *LNCS*, pages 133–140. Springer, 2000.
16. J. A. Kohl and G. A. Geist. The PVM 3.4 tracing facility and XPVM 1.1. In *Proc. of the* 29^{th} *HICSS*, pages 290–299. IEEE Computer Society Press, 1996.
17. D. Kranzlmüller, Ch. Schaubschläger, and J. Volkert. A brief overview of the MAD debugging activities. In *Proc. of AADEBUG 2000*, Munich, Germany, August 2000.
18. D. Kranzlmuller, S. Grabner, and J. Volkert. Debugging massively parallel programs with ATTEMPT. In *High-Performance Computing and Networking (HPCN'96 Europe)*, volume 1067 of *LNCS*, pages 798–804. Springer, 1996.
19. H. Krawczyk and B. Wiszniewski. Interactive testing tool for parallel programs. In *Software Engineering for Parallel and Distributed Systems*, pages 98–109, London, UK, 1996. Chapman & Hal.
20. 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.
21. J. Lourenço and J. C. Cunha. *Flexible Interface for Distributed Debugging (Library and Engine): Reference Manual (V 0.3.1)*. Departamento de Informática da Universidade Nova de Lisboa, Portugal, December 2000.
22. J. Lourenço, J. C. Cunha, H. Krawczyk, P. Kuzora, M. Neyman, and B. Wiszniewsk. An integrated testing and debugging environment for parallel and distributed programs. In *Proc. of the* 23^{rd} *EUROMICRO Conference*, pages 291–298, Budapeste, Hungary, September 1997. IEEE Computer Society Press.
23. T. Ludwing, R. Wismüller, V. Sunderam, and A. Bode. OMIS – On-line monitoring interface specification. Technical report, LRR-Technish Universiät München and MCS-Emory University, 1997.
24. B. P. Miller, J. K. Hollingsworth, and M. D. Callaghan. The Paradyn parallel performance measurement tools. *IEEE Computer*, 28(11):37–46, November 1995.
25. W. E. Nagel et al. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69–80, January 1996.
26. M. Oberhuber. Managing nondeterminism in PVM programs. *LNCS*, 1156:347–350, 1996.
27. Michael Oberhuber. Elimination of nondetirminacy for testing and debugging parallel programs. In *Proc. of AADEBUG'95*, 1995.
28. Michael Oberhuber and Roland Wismüller. DETOP - an interactive debugger for PowerPC based multicomputers. pages 170–183. IOS Press, May 1995.
29. D. Stringhini, P. Navaux, and J. C. Kergommeaux. A selection mechanism to group processes in a parallel debugger. In *Proc. of PDPTA'2000*, Las Vegas, Nevada, USA, June 2000.
30. Dolphin ToolWorks. *TotalView*. Dolphin Interconnect Solutions, Inc., Framingham, Massachusetts, USA. http://www.etnus.com/Products/TotalView/.