

A Software Architecture for the Integration of Monitoring, Debugging and Profiling Tools for Parallel Program Development

José C. Cunha Vitor Duarte João Lourenço Pedro Medeiros

{jcc, vad, jml, pm}@di.fct.unl.pt

Departamento de Informática

Faculdade de Ciências e Tecnologia

Universidade Nova de Lisboa

Portugal

Abstract

In this paper we present our contribution towards building tools that support the development of parallel and distributed applications composed by heterogeneous components. This research tries to give a positive answer to the needs arising in many application domains where it is necessary to build systems from separately developed components such as visualization tools, interactive control components, virtual reality interfaces, and simulation components. For such environments, software engineering tools can be considered at two distinct levels: the intracomponent level, which corresponds to tools acting upon individual components, and the metacomponent level, that must address the abstractions for the specification of the application configuration and the components interconnection. The distinct aspects of program analysis, including verification, testing and debugging, monitoring and profiling, visualization and user interaction, also appear at the metacomponent level. We describe our approach to build an integrated development environment where the above functionalities can be incrementally included. In this paper we illustrate the current status of our work by describing a prototype environment with a basic monitoring and control layer, and how this was extended to support profiling and debugging services.

1 Introduction

This research tries to give a positive answer to the needs arising in many application domains where it is necessary to build systems from separately developed components such as visualization tools, interactive control

components, virtual reality interfaces, and simulation components.

Software engineering environments for such heterogeneous software architectures should encompass tools supporting the application development stage as well as the application execution stage.

During application development, one must be able to model and reason about the global structure and properties of a system composed by multiple heterogeneous components. As most of these applications will run on parallel and distributed platforms, one is faced with very complex patterns of interaction among components. So, in order to understand and build such complex systems, it is necessary to provide adequate formal models [S⁺95, SG95, MK96], coordination models [PA97], resource management and interconnection models [BGR97], and monitoring and control models [LWSB97].

Software engineering tools can then be considered at two distinct levels:

- The intracomponent level. To support the development of each individual component, it encompasses the aspects of program analysis, including verification, testing and debugging, monitoring and profiling, all coupled with suitable visualization and user interfacing tools [W⁺94].
- The metacomponent level. These tools must handle the abstractions used for application composition and configuration, and for component interconnection.

1.1 The Intracomponent Level

At the intracomponent level, each component is typically based on a distinct programming and computational model. This includes components which are based on sequential and concurrent models, as well as on parallel and distributed models. Currently, the user

must adapt to many distinct programming environments according to the semantics of the model that is used by each individual component. The development tools are also highly dependent on the semantics of each specific model which makes it very difficult to develop heterogeneous applications.

There have been significant proposals towards the specification of standard models and interfaces that can be used as basis for unifying programming platforms or tools, such as [OMG96, For94, LWSB97, BFP97].

However, much research is still needed to define frameworks that can be used to integrate distinct software engineering tools in a coherent environment.

1.2 The Metacomponent Level

At the metacomponent level, new issues arise and have been recently become the focus of increased research [S⁺95]. In our research we are addressing these issues at distinct conceptual layers of the hierarchy that is shown in figure 1.

Formal mechanisms	<i>Tools</i>
Coordination methods	
Resource managing/ Interconnection services	
Monitoring and control layer	
Heterogeneous hardware/software	

Figure 1: Conceptual Software Layers

The figure illustrate the gap between the high-level specification models for application composition and the low-level system-based layers. The mappings between the successive layers are still being investigated, as far as program development is concerned. This involves understanding the correctness and performance analysis of the component interconnection protocols, the impact of alternative communication topologies and their dynamic reconfiguration, and the incremental refinement and composition of concrete architectures [S⁺95].

At the metacomponent level, there is a need to develop adequate tools that support the process of application building, including the configuration, testing, debugging for correctness and for performance of systems composed by multiple heterogeneous components. At this level the integration of program analysis and dynamic analysis tools becomes particularly important because of the dynamic interaction patterns between the application components. For example it is necessary to gather runtime information about component interactions, and their dynamic activation and reconfiguration, because this is particularly important to assess relevant application properties concerning correctness and performance. However, this requires meta-tools which are able to understand the abstractions related to global system composition and configuration, and which are responsible for the management of the individual tools related to each individual component.

1.3 Contributions and Organization of the Paper

In order to address the above issues there is a very important current concern on tool integration, searching for a software development environment that provides consistent views among its multiple dimensions: multiple user interfaces, tool behavior, tool interaction, and tool composition [CL97]. The goal is to allow the user to work in a unified and coherent environment to access the abstractions provided by each tool.

We aim to contribute to the development of software engineering environments supporting modular and easy integration of new tools, both at the intracomponent and at the metacomponent level. This is a very ambitious goal, because the models illustrated in figure 1 are still being investigated and evaluated in a diversity of heterogeneous applications, so it becomes difficult to develop such meta-tools (dealing with the metacomponent level) from the scratch... without intensive experimentation.

Within the scope of a currently ongoing project, our approach was to develop a framework supporting the incremental extension and integration of new or existing tools into a development environment. This allows us to experiment with the desired tool functionality, and to adapt it according to the specific models being used at each abstraction layer of figure 1. This allows us to build quick prototypes, to experiment with them with selected applications, to extend the prototypes with new or existing tools, and to evaluate them so that we can improve their functionalities.

In this paper, the focus is on the following aspects:

- Section2: the basic functionalities of a monitoring

and control layer that can be customized to work both at the component and at the metacomponent levels. We describe the DAMS software architecture.

- Section 3: the integration of basic distributed debugging services into the DAMS architecture, and how they can be used at higher levels.
- Section 4: the design of a basic profiling service, its integration into the DAMS architecture, and its use at higher levels.
- Section 5: conclusions and current status.

For the debugging and the profiling extended functionalities, we discuss several uses, at distinct levels of abstraction:

- To support component level debugging and profiling of a distributed programming language [CM97].
- To support metacomponent level debugging and profiling of heterogeneous applications using the PHIS interconnection model [MC97].

2 A Monitoring and Control Layer

The basic functionalities provided by DAMS (Distributed Applications Monitoring System) are supported by an architecture that is illustrated in figure 2. It allows the monitoring and control of an application consisting of a set of distributed Target Processes.

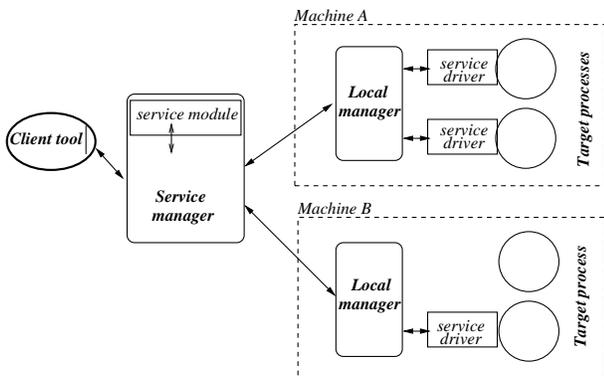


Figure 2: DAMS Architecture

DAMS is organized in terms of a set of Local Manager Processes (LM), one on each node of a physical architecture, that are responsible for the management of local processes and for the communication with the central Service Manager Process (SM).

The SM process does not directly handle the application level commands which are issued by the client tools. These commands are forwarded to specific Service Modules which are responsible for their management and interpretation, according to the semantics of each corresponding service (e.g. for debugging or for profiling). On each node, Driver processes are responsible for the direct enforcement of such commands to each Target Process.

DAMS provides well-defined interfaces between the client tools, the SM, the LM processes, and the Drivers. However, it does not include any built-in service functionalities. Each functionality must be explicitly added to the system by specifying a pair (Service Module, Driver). So, DAMS provides a very flexible and easily extensible environment for the experimentation with both component-level and metacomponent-level tools, as we illustrate in the following sections.

3 Debugging Services

This section illustrates the use of DAMS to support distinct debugging functionalities, and how they can be used at component-level and at metacomponent-level.

3.1 Basic Functionalities

The basic functionalities which are required by a debugging service concern state inspection and control of a computation. This includes abstractions related to 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, but it is possible to identify a set of basic debugging mechanisms (e.g. [BFP97]), and use them in order to implement higher level functionalities. Recently we have been working on the implementation of the DDBG distributed process-level debugger for C/PVM programs [CLA96].

3.2 Debugging Interfacing with Graphical and Testing Tools

Another important aspect that we have exploited concerns the support for the interfacing of the debugging service with other tools in a software engineering environment:

- Interfacing with graphical and visualization tools. The DDBG debugger was used to implement high-level debugging of a visual parallel programming language [KCD⁺97] which is part of the GRADE development environment [KCD⁺97].

- Interfacing with static analysis and testing tools. The STEPS tool [KW96] supports a methodology to aid the user in the process of identifying the paths which should be generated and tested for C/PVM based programs. The DDBG debugger was integrated with such testing tool in order to support user controlled execution of the paths under test, allowing the user to inspect program behavior at the desired level of abstraction and with the guarantee of the reproducibility of its execution [LCH⁺97].

3.3 Heterogenous Debugging Services using DAMS

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.

The DAMS architecture meets such requirement because it allows the integration of any new debugging service by implementing a Debugging Service Module and its associated Debugging Driver. In order to test this idea, we have implemented a process-based debugging interface (PDBG) as a DAMS service [CLJ⁺97]. For each target process under debugger control, a Driver is launched which uses the GNU gdb debugger to inspect and control the application process. The Debugging Service Module supports the user-level debugging interface and is responsible for the generation of commands which are forwarded by the Service Manager to the corresponding driver, and for the interpretation of corresponding replies.

In order to support thread-based debugging it is only necessary to specify the desired debugging interface (to be supported by a new Service Module), and its associated Driver. Any existing thread-based debugger can be integrated into DAMS, by simply writing an associated Driver which is responsible for the interpretation of its commands and replies.

Another example is given by the development of a debugger for the PVM-Prolog distributed programming language [CM97]. PVM-Prolog extends Prolog with predicates supporting a PVM interface, allowing the implementation of distributed Prolog applications where several independent agents can evaluate goals and exchange messages (interpreted as Prolog terms). PVM-Prolog also supports communication between C and Prolog PVM tasks and is being used to implement multi-agent systems [SRWC97]. A heterogeneous debugger

for PVM-Prolog is implemented as a service on the DAMS architecture, by having a Service Module for the language-level debugging interface, and several Drivers. Each Driver controls an individual debugger: this is GNU gdb if the associated Target Process is a C/PVM task, or it is a Prolog debugger if the Target Process is a Prolog task.

3.4 Metacomponent-Level Debugging

In order to support heterogeneous applications, we have developed the PHIS model [MC97]. PHIS is a group-oriented interconnection model that operates at the meta-component level, i.e. it allows the communication between separate application components. Each application component can be based on a distinct programming and computational model, e.g. one PVM-based and other MPI-based. Figure 3 illustrates the abstract schema of a PHIS-based configuration. In [MC97] we illustrate a concrete application of PHIS to support heterogeneous application based on parallel genetic algorithms.

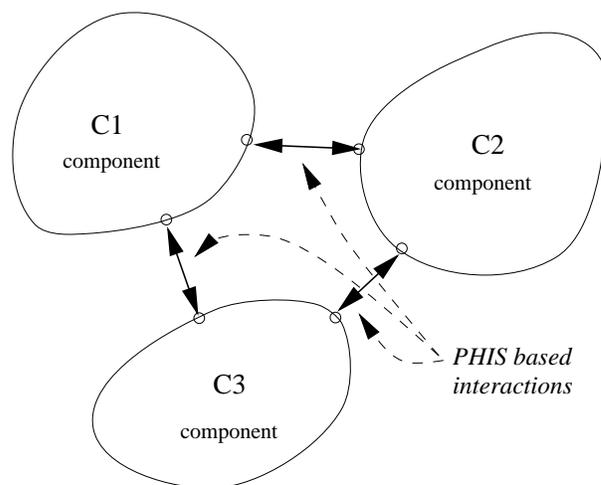


Figure 3: PHIS Architecture

In PHIS, individual processes in each component can communicate with processes in other components, or can join groups where message multicast is supported.

A debugger that operates at the PHIS interconnection level must support facilities for the inspection and control of the interaction points between application components. For example if a process in a PVM-based component broadcasts a message to other PVM and MPI based components, and if one wants to put a breakpoint in each receiver process, this cannot be achieved by typical existing debuggers e.g. for PVM or MPI

systems. However, it is easy to integrate a new Service Module in the DAMS system that acts as a meta-debugger which interprets the debugging interface commands (related to PHIS communication abstractions), and send commands to each specific debugger, e.g. PVM-based or MPI-based, that is associated with component processes.

With the emergent development of formal specification languages for software architectures [S⁺95] we expect to be able to integrate verification, static analysis, and testing tools that operate at the metacomponent level, with our PHIS-based debugger, in a similar way as we have integrated the STEPS [KW96] and DDBG tools.

4 Profiling Service

This section illustrates the implementation of profiling services on top of the DAMS architecture. This service provides the functionalities for event counting and timing of functions calls at several levels of the application: operating system, libraries or user programs.

In order to support such monitoring at several levels, we consider two types of support for information generation:

1. support by the system;
2. support by the application program;

For the first type, the system can provide some support for getting the relevant information, e.g. based upon preinstalled software instrumentation that can be used by the profiling service. For example, PVM can trace its library function calls.

The second type of information generation depends on the programming language model. Typically some kind of processing must be done at the program source code level or by the language runtime support system, so that the relevant information can be retrieved.

Information recording and transfer can be controlled depending on the tool needs. Several modes of operation can be supported: to collect a total trace of the events or just a summary of the number of calls and the time used by each function. The tool can be interested in runtime information (e.g. for online real-time visualization), or it can accept local buffering in order to reduce the perturbation due to communication overhead. The tool will act as a consumer of the collected information. For other tools, a *post-mortem* trace file is more suitable (e.g. for performance analysis of a complete program execution).

4.1 Implementation

For the implementation of this service in the DAMS architecture there will be an Service Module that supports the functionalities needed by the tools. This module interacts with the Drivers on each node for collecting the required information.

Concerning the information that must be provided by the supporting system, there must exist a component in the Driver that interacts with the system being used by the monitored task. Concerning the information related to the programming language concepts, there must exist a component in the Driver that interacts with (ie, it is included in) the application code.

Internally, each Driver instance will manage a collection of counters, timers or buffers, as needed by each operation mode, and will manage their associations with the respective objects being traced.

Using the C/PVM case as an example, this system already supports some kind of trace. Other packages can be used for monitoring PVM applications[Mai95]. The profiling service will act as a front-end and the details regarding the interactions with the underlying PVM system monitor are hidden by the Driver. Using the PVM tracing facility, each Driver can receive the relevant events from the target process and treat them according to the requested operation mode. For a summary-only mode it will accumulate the information in the counters and timers. In trace mode, the Driver will collect the event description.

If the tool needs the report of the new information in real-time, this will be immediately sent using the DAMS communication mechanisms. If a delayed mode was requested, the Driver will try to use buffering mechanisms from the monitoring system, if available, or will implement this mechanism and send the information periodically to the Service Module. Alternatively, information transfer to the central node can occur only after execution.

The tool can select what to record by selecting the individual tasks and functions that should be traced. This information will be propagated by the Service Module, using the DAMS mechanisms, to the Drivers and these will evaluate the trace mask locally and ensure that no more than the requested information is kept, if possible, not even collected to reduce intrusion in the running program.

For the tool to get the information from the Profiling Service, it can request the new information in a synchronous way or register an handler for asynchronous notification.

In order to support programming language level events and taking user defined events into account, the application will need to access the Driver for registering this

kind of information with functions like:

```
event_start(id); event_end(id);
```

and the Driver level that can be used to implement the program instrumentation.

Other possibility, previously presented, is to use dynamic instrumentation during run-time. In this situation there is no need for the programmer to change the source code by instrumenting all the possible points of interest. The Driver can make use of a debugging service to insert instrumentation probes (through break-points) in a target process and then process the triggered events.

4.2 A Profiling Service for PVM-Prolog

At the PVM-Prolog level the monitoring is achieved by instrumenting the PVM related predicates. Such instrumentation was initially implemented using the Tape/PVM[Mai95] monitoring system. The ParaGraph[HF94] visualization tool was then used to display the PVM related communication interactions.

As the recorded information should concern the Prolog semantics one should get descriptions about predicate evaluations. A new Service Module can make use of such information to interact with the programmer at the PVM-Prolog program level, and allow the user to declare points of interest for profiling by using a built-in predicate, e.g. `prof(G)`, where G is a Prolog predicate, for recording information on G evaluation.

4.3 A Metacomponent Profiling Service for the PHIS model

Metacomponent profiling of PHIS-based applications can be achieved by instrumenting the PHIS primitives in order to generate the trace information describing the interactions between application components. A new Service Module is responsible for the interpretation of such information regarding the PHIS abstractions.

The above metacomponent profiling service can co-exist with component level profiling.

5 Conclusions

In this paper we have discussed issues concerning the development of component level and metacomponent level tools and their integration in a software engineering environment. We have presented our approach to support the experimentation with such types of tools. We have illustrated our work by describing a monitoring and control architecture that is easily extensible with new services, according to the desired abstraction

level. Implementations of debugging and profiling services were discussed, and their use at distinct levels was shown.

Current implementation status corresponds to a DAMS prototype running on our local network, the distributed process-level debugging service, the PVM-Prolog profiling service, and a PHIS prototype supporting interconnection of PVM and MPI components.

Acknowledgments

Thanks are due to João Vieira, Bruno Moscão and Daniel Pereira for their work in the DAMS system. Previous work on the DDBG tool and its integration with the STEPS and the GRADE tools was partially supported by the EC Copernicus SEPP (CIPA-C193-0251) and HPCTI (CP-93-5383) Projects. The work reported in this paper was partially supported by the Portuguese CIENCIA and PRAXIS XXI Programmes, PROLOPPE Project (3/3.1/TIT/24/94) and by the DEC EERP PADIPRO Contract 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/>”, 1997.
- [BGR97] M. Brune, J. Gehring, and A. Reinefeldn. A lightweight communication interface between parallel programming environments. In *Proceedings of HPCN'97 High Performance Computing and Networking*, pages 17–31. Springer Verlag, 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.
- [CLA96] J. C. Cunha, J. Lourenço, and T. Antao. 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*, Miskolc, Hungary, October 1996.
- [CLJ+97] José C. Cunha, J. Lourenço, Vieira J, Bruno Mosc ao, and Daniel Pereira. A

- framework to support parallel and distributed debugging. Technical report, Departamento de Informática, Universidade Nova de Lisboa, November 1997. submitted for publication.
- [CM97] Jose' C. Cunha and Rui F. P. Marques. Distributed Algorithm Development with PVM-Prolog. In *Proceedings of the 5th EUROMICRO Workshop on Parallel and Distributed Processing, IEEE Computer Society Press, London, 1997*.
- [For94] Message Passing Interface Forum. MPI: A message-passing interface standard. Computer Science Dept. Technical Report CS-94-230, University of Tennessee, Knoxville, TN, April 1994. (To appear in the International Journal of Supercomputer Applications, Volume 8, Number 3/4, 1994).
- [HF94] Michael T. Heath and Jennifer E. Finger. *ParaGraph: A Tool for Visualizing Performance of Parallel Programs*. University of Illinois and Oak Ridge National Laboratory, 1994.
- [KCD⁺97] P. Kacsuk, J. Cunha, G. Dózsa, J. Lourenço, T. Fadgyas, and T. Ant ao. A Graphical Development and Debugging Environment for Parallel Programs. *Parallel Computing, 1997(22):1747–1770, February 1997*.
- [KW96] H. Krawczyk and B. Wiszniewski. Interactive Testing Tool for Parallel Programs. In P. Croll Chapman & Hal: I. Jelly, I. Gorton, editor, *Software Engineer for Parallel and Distributed Systems*, pages 98–109, London, UK, 1996.
- [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.
- [LWSB97] T. Ludwig, R. Wismuller, V. Sunderam, and A. Bode. OMIS — On-Line Monitoring Interface Specification (Version 2.0). Technical report, Lehrstuhl für Informatik, Technical University of Munich (LRR-TUM), Munich, Germany, July 1997.
- [Mai95] Eric Maillet. *Tape/PVM an efficient performance monitor for PVM applications – User guide*. LMC-IMAG, Grenoble, France, June 1995.
- [MC97] P. D. Medeiros and J. C. Cunha. Interconnecting Multiple Heterogeneous Parallel Application Components. In *Proceedings of EuroPar'97*, Passau, Germany, August 1997.
- [MK96] J Magee and J Kramer. Dynamic structuring in software architectures. In *Proceedings of 4th ACM SIGSOFT Symposium on Foundations of Software Engineering, 1996*.
- [OMG96] OMG. *The Common Object Request Broker: Architecture and Specification*, July 1996.
- [PA97] G.A. Papadopoulos and F. Arbab. Coordination models and languages. provisional version, 1997.
- [S⁺95] M. Shaw et al. Abstractions and tools for software architectures and tools to support them. *IEEE Transactions on Software Engineering*, 1995.
- [SG95] M. Shaw and D. Garlan. *Formulations and Formalisms in Software Architecture*. Number 1000 in Lecture Notes on Computer Science. Springer, 1995.
- [SRWC97] M. Schroeder, R. Marques, G. Wagner, and J. C. Cunha. CAP - Concurrent Action and Planning: Using PVM-Prolog to Implement Vivid Agents. In *Proceedings of PAP'97, 5th International Conference on The Practical Application of PROLOG, The Practical Application Company, London, UK, 1997*.
- [W⁺94] S Winter et al. Software Engineering for Parallel Processing, Copernicus Programme. Progress report 1, University of Westminster, October 1994.