OMIS 2.0 — A Universal Interface for Monitoring Systems

Thomas Ludwig, Roland Wismüller

Technische Universität München (TUM), Informatik
Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR)
Arcisstr. 21, D-80333 München
email: {ludwig|wismuell}@informatik.tu-muenchen.de

Abstract. The OMIS project aims at defining a standard interface between tools for parallel systems and monitoring systems. Monitoring systems act as mediators between tools and the parallel program running on some target architecture. Their task is to observe and manipulate the program according to the tools' commands. A standardized interface will allow different research groups to develop tools that can be used concurrently with the same program. OCM, an OMIS compliant monitoring system, is the first implementation of such an environment. It is designed for PVM programs running on workstation clusters. This paper will give an outline of the goals of the OMIS project and describe important details of the OCM design.

1 Motivation

In parallel and distributed computing, programmers are still confronted with the situation that there is a lack of support tools for their work. Specifically, when the first prototype is running, it is hard to find efficient debugging facilities to correct e.g. errors due to race conditions. In a production environment finally, only few environments provide tools for e.g. load balancing or checkpointing.

Not only are there just few specialized tools for specific purposes, there is also almost no integrated environment available where several tools can be concurrently applied to program development. Furthermore, there is no uniformity in tools; on every platform you find different implementations that nevertheless share a set of common functionalities.

In this paper we will analyze the preconditions of this situation and present an approach that will help us to overcome them.

Let us first have a closer look on tools for parallel and distributed computing. We are interested in tools that support those phases of the software life cycle where the program is already running. Usually, we find tools for debugging, program flow and data visualization and performance analysis. They help us to produce an error-free, efficient production code of the application. In the production phase, we use tools for resource management, load balancing, checkpointing, and computational steering.

An analysis of tool architectures shows that we can distinguish interactive tools (usually with a graphical user interface) and automatic tools (usually hidden in levels close to the operating system). Considering program interaction, we find tools that merely observe the parallel application and others that in addition manipulate it. Finally, with the question "when is the tool applied", we can separate off-line tools from on-line tools.

Off-line tools get their input information from traces written during the program run. Obviously, an interaction in form of manipulation is not feasible here. On-line tools interact with the running program via a monitoring system that observes the program and also can manipulate it. However, not all monitoring systems are capable of manipulation due to their restricted implementation.

Considering nowadays tools' scenery, we can state the following:

1. There are more trace-driven off-line tools than on-line tools. The reason lies in the implementation complexity of monitoring systems for the second class.
2. On-line tools with underlying monitoring systems cannot be used at the same time during the same program run as their internal mechanisms of information acquisition are usually incompatible to each other.

The approach presented in our paper starts with the observation that the main reason for deficiencies with tools in this field is the complexity of the monitoring system that cannot easily be mastered. A standardized interface between tools and a monitoring system together with a powerful implementation of such a system being freely available to the scientific community could considerably ease the development and integration of various different tools. This is just what the OMIS project aims at. OMIS stands for **O**n-line **M**onitoring **I**nterface **S**pecification.

2 Tools for Parallel and Distributed Programs

We cannot give a detailed overview of all on-line monitoring based tools for parallel and distributed computing. However, we will list some important representatives of different tool classes and show the added value that could result from a standardized monitoring system.

Important representatives of on-line debugging tools are p2d2 by Cheng/Hood [4] and DETOP by Wismüller/Oberhuber [3]. p2d2 is a client-server oriented approach with a specified protocol for communication. The debugger server contains the platform-dependent code while the client consists of the portable user interface code. The server defines objects such as processes and array values. The client can perform various operations on these objects. Cheng and Hood propagate the idea that vendors should provide debugger servers together with their machines. Clients could then be used in a uniform way on different target platforms. In the meantime they implement gdb-based servers. DETOP runs on different target platforms like MPP systems and NOWs. Its monitoring system is always closely connected to the runtime environment of the target architecture. A lack of standard interfaces in the lower levels of these systems forces the monitoring system to be heavily adapted when changing the target architecture.

Regarding performance analysis we like to present Paradyn by Miller/Hollingsworth [9] and PATOP by Hansen, Krammer, Dimke, and Borgeest [13]. Paradyn is a powerful approach with on-line manipulation of the running code. Although from the user's point of view it performs only observation, it inherently modifies the code when adding instrumentation for counters and clocks. It represents the state-of-the-art of performance analysis for distributed programs. However, due to its complex interaction mechanism with the software layers of the target architecture, its portability is critical. The same is true for PATOP as it runs on the same target platforms as DETOP does. Fortunately, porting the monitoring system for DETOP leads to an almost operational version for PATOP. Thus, both tools profit from the fact that they are based on a common monitoring interface.

Several approaches are available for load balancing and resource management, such as the MIST environment by Walpole et al. [1] and the CoCheck checkpointing facility by Stellner [10]. MIST aims at combining load balancing for PVM in an environment together with XPVM. Goals are very ambitious and comprise porting all software to MPI and several target architectures. CoCheck is available for PVM and MPI [11] and allows to checkpoint any number of processes. Thus, process migration and rollback recovery can be based on this mechanism.

What is common to all of these tools (and also valid for almost all other tools not mentioned here)? All of them need a specially adapted runtime environment in order to interact with the system. The implementation of this adaption is difficult and time consuming as no standards exist on which it could be based. In the end, similar software had been developed that unfortunately can only be used for the project of the individual research group. The OMIS project tries to eliminate this additional effort and to define a standard interface to an infrastructure all of these tools could be based upon.

3 Approaches in Designing Monitoring Interfaces

OMIS is not the first approach that tries to define a standard interface for middle-ware software layers. However, it seems to us that it is currently the only one that concentrates on complete tool environments for parallel and distributed systems. Let us have a look on interface specifications that are situated "around" OMIS, i.e. are lying on a higher or lower level of abstraction or are aside OMIS.

ToolTalk [5] is a concept developed by SunSoft. It enables independent applications to communicate with each other without having direct knowledge of each other. Applications create and send ToolTalk messages to communicate with each other. The ToolTalk service receives these messages, determines the recipients, and then delivers the message to the appropriate applications. ToolTalk is on a higher level of abstraction. It could help us to solve interoperability problems that are not covered by OMIS itself. Without OMIS it can only be applied to monolithic tools that work on independent object sets.

An approach concentrating on debugging is defined by the p2d2 project [4]. It exhibits a client-server architecture that permits the user interface client to interact with implementations of debugger services on a variety of target platforms. The particular protocol chosen isolates non-portable debugger code in the server. By means of this, we eventually arrive at uniform debugging tools where identical front-ends exist for many different target architectures.

At an even lower level we find an API for the instrumentation of code during runtime. The concepts were developed by Hollingsworth and are employed in the Paradyn performance analysis tool. Until now, there is no official publication available on this work. However, the approach is very powerful and could form a basis for various different tools which need on-line code instrumentation.

A final candidate to be presented here is the simple network management protocol SNMP [2]. It was developed to manage Internet nodes. Using different management information bases (MIBs) SNMP can also be deployed for purposes similar to OMIS (see e.g. [6]). The interface of SNMP also provides mechanisms for observation and manipulation. Its concept of event detection, however, is not sophisticated enough to support efficient debugging functionalities.

None of the above concepts is by itself an appropriate candidate to cover the list of goals of the OMIS project. However, we try to learn from these approaches how to manage projects that deal with interface specifications and how to solve concrete problems with the syntax and semantics of our interface.

4 The Leading Ideas of the OMIS Project

The leading ideas of OMIS are to define a standardized interface between tools and monitoring systems in order to separate the work of research and development groups being active in one or the other field. By separating it, tool developers could fully concentrate on their tools' functionalities without being forced to think of how to instrument a Unix process or deal with operating system interfaces. Specialists for operating systems related software concepts could concentrate on providing monitoring systems for different target platforms. As for MPP and SMP systems it could be the role of the vendors to fill in this gap.

In the end we will save time: having n tools on m target platforms is now an effort proportional to $n + m$ whereas without OMIS the effort would be $n \times m$.

Two new conceptional possibilities arise from OMIS based environments:

– Eventually we will have **interoperable tools**. They will be based on an identical interface and thus can be made aware of each other. A concurrent application of different tools during the same program run is then feasible.
– Eventually we will have **uniform tool environments**. Porting the monitoring system onto another target platform will immediately make all tools available on this system that were also available at the original system. This is extremely beneficial for the application programmers who can then stick to tools they are accustomed to.

5 OMIS 2.0

5.1 Concepts

This section will give an overview of concepts of OMIS 2.0. More details on issues discussed can be found in [8]. The objective of the interface is that it enables tools to observe and manipulate a running parallel or distributed program. The execution paradigm chosen for the monitoring system in order to reach this objective is the event-action model. It means that we specify event-action relations (called requests) that define actions to be performed when a certain event occurs, i.e. a well-defined change of state takes place in the program under consideration.

The monitoring system must be programmable in the sense that any set of such relations can be sent from a tool to the system. Its task is to check for an event occurrence, trigger corresponding actions, and send results (if any) back to the tool. The interface used to transfer these relations as well as the possible services, i.e events and actions is just what OMIS defines.

What are the general requirements to be met by such an interface? First, its functionality must be sophisticated enough to support many different types of tools: interactive, automatic, centralized, and distributed tools for various purposes listed in earlier sections. Second, it must exhibit a mechanism for extensions. OMIS does not cover the functionality of all current and future tools, either because the functions they offer are very special or because they are unknown today. However, the interface provides a means to extend both the list of available services and the set of accessible objects. Third, the structure of the interface must be appropriate to ensure an efficient implementation. Intrusion on the system to be observed must be kept low. Here, we will discuss the first issue only; for further details please see the specification document [8].

The functionality of the interface is defined by the set of object classes that can be handled and the services that are offered to do so.

Most of the classes are implicitly fixed by the programming paradigm employed. In the first phase of OMIS, we deal with message passing only. Thus, we find the following hierarchy of objects classes: At the top there is the node object class that represents units onto which parts of the parallel program can be distributed. A node might be a single processor workstation, a multiprocessor SMP workstation, or a node of an MPP system (i.e. a node is not just a processor). At the next lower level we find processes as active objects of our program. One process is always located on a single node and cannot be spread over several of them. Inside processes we find thread objects. They always belong to one process and cannot migrate to others. In case of programming models without threads we define that a process contains exactly one thread. For communication between processes we have message queue objects, which finally contain message objects.

All object classes are subject to a hierarchy where some of the concrete relations depend on the chosen programming model. With PVM, there are currently no threads and message queues belong to process objects. There are other models where message queues are realized as mailboxes between processes and thus belong to nodes and not to a single processes.

With the introduction of the monitoring system, we add new objects to this hierarchy: conditional requests and user defined events.

The listed object classes form the core of OMIS and must exist in every OMIS compliant monitoring system. However, with extensions to be added, new object classes might appear like files or shared memory segments.

For each class, a list of services is defined that can be used on the objects. Three categories are distinguished: information, manipulation, and event services. The last category is necessary to notify us that a specified change of state appeared with the object. With processes we can for example inquire their state, stop their execution, or get a notification on process completion. The complete list of services for each class can be found in the OMIS document.

Almost all services work on sets of objects. Object sets are either specified explicitly by listing the tokens (=abstract identifiers) of all objects the service should refer to, or they are defined implicitly. We call a specification implicit, if the token given to the service does not refer to the class of objects on which the service operates, but to a class that is above or below in the hierarchy. If it is above, the specified token is expanded to a set of tokens of the correct class. A request "stop processes" with a node token as input parameter thus means "stop all processes on that node". If the given token is lower in the hierarchy, it is transformed to the appropriate level via localization. A request "give me node information" together with a process token thus means "give me node information of the node where this process is located". In summary, we provide an abstract referencing scheme where globally unique object tokens can be used to refer to objects at different abstraction levels.

Having discussed object classes, services, and object tokens we are now ready to put it all together and to compose a request that is sent from a tool to the monitoring system. Informally, a request is composed of an optional event service definition followed by an action list, where each individual action is an information or manipulation service definition. If no event service is defined, the request is unconditional and is executed immediately. If an event definition is present we call the request conditional. The action list is triggered whenever a matching event takes place.

The main function of the monitoring interface has the following prototype:

```
Omis_reply omis_request(char * request,
             void (* callback) (Omis_reply reply, void * param),
             void * param, Omis_flags flags)
```

A callback function is given for conditional requests in order to send results back to the tool asynchronously. The flags control certain variants of blocking behavior of the call and answering behavior of the monitoring system. They are used to optimize the protocol between tool and monitoring system and to avoid sending unnecessary messages or notifications.

The results returned to the tool have a complex structure according to the number of actions contained in a request and the number of objects each service referred to. What makes it even more complex is the fact that various error conditions may exist that must be handled in an efficient and unambiguous way. For details on this issue please see the specification document [8].

5.2 Example for a Performance Analysis Tool

Let us now consider an example. Assume that a performance analysis tool wants to measure the time spent by task 4178 in the `pvm_send` call. In addition, the tool may want to know the total amount of data sent by this task, and it may want to store a trace of all barrier events. Then it may send the following service requests to the monitoring system:

No.	request string	result token
1	thread_has_started_lib_call([p_4178],"pvm_send") : timer_start([pa_t_1]) counter_add([pa_c_2],$par5)	c_1
2	thread_has_ended_lib_call([p_4178],"pvm_send") : timer_stop([pa_t_1])	c_2
3	thread_has_started_lib_call([],"pvm_barrier") : print(["pvm_barrier entered",$node,$proc,$time])	c_3
4	thread_has_ended_lib_call([],"pvm_barrier") : print(["pvm_barrier left",$node,$proc,$time])	c_4
5	: csr_enable([c_1,c_2,c_3,c_4])	

The tokens c_1...c_4 are identifiers for the conditional service requests. They are delivered by the monitoring system as a direct reply to the request. The fifth request, which is unconditional, does not yield such a token.

The event services used in this example are of special importance as they bridge the gap between the universal monitoring interface and an interface adapted for the PVM programming model. These services allow to monitor library calls of any programming library, in our example of the PVM package. The service `thread_has_started_lib_call` is matched by any event where an appropriate library call is started. Accordingly, `thread_has_ended_lib_call` is matched by any completion of such a call.

With the conditional service request c_1 we wait for pvm_send calls to start. In this case, activate timer 1 for time measurement and increment counter 2 by the number of bytes to be transferred with this call (value of parameter $par5). When the send call completes, we stop timer 1.

Whenever a PVM barrier is entered or left, conditional services 3 and 4 transfer interesting information directly to the tool, which may for example write them to a trace file.

The example also shows the usage of event context parameters: the variables $par5, $node, $thread, and $time will get concrete values before the actions they belong to are started. These values are output values of the triggered event services.

At this point, the measured values are not yet available in the tool itself. To read the counters and timers, further unconditional requests are necessary.

5.3 Goals of the OMIS Project

The goals of the OMIS project are widely spread and cover several issues. First of all we want to improve the specification as much as possible to make it useful to other developer groups. Currently we are already implementing an OMIS compliant monitoring system (OCM) for PVM based programming on workstation clusters. As soon as this monitoring system is operational we will start to adapt already available tools of our group to that interface. All development within the framework of OMIS and OCM will be available under GNU license conditions. We invite everybody to make comments on this project and to contact us for cooperations.

6 Implementation Issues

Since the beginning of 1997 we are working on an implementation of an OMIS compliant monitoring system. The design is rather complex and will be described in a separate paper. The OCM serves two main purposes: a) it provides a working platform for tool developers (especially for us [12]) and b) it establishes a reference for developers of other OMIS compliant monitoring systems for different target architectures.

Obviously, as the system to be monitored is distributed, the monitoring system itself has to be a distributed system, too. It is composed of single monitors, one on each node participating in our environment. The set of node-local monitors composes the monitoring system.

Each local monitor works according to the event-action model: it receives requests referring to its node and its objects and sends back replies to these requests. On the other hand it observes and manipulates objects on its node and starts services if it sees specified changes of state taking place.

In the first implementation phase all requests are parsed and analyzed by a central component called node distribution unit. It checks which nodes the request refers to and sends appropriate partial requests to each node. If replies are issued, the node distribution unit collects all individual results, packs them to a global reply, and sends it back to the tool. By this, local monitors are not and do not have to be aware of each other.

In the implementation, we do not only find monitor processes but also monitor functionality being linked to each application process. This is necessary for a fast detection of events in the application processes' contexts (e.g. a specified function is called) and for the efficient execution of certain actions (e.g. manipulation of counters).

The problem arises, how all these components can communicate. To be flexible, communication is homogeneously handled by an intermediate layer where it depends on its actual implementation which low-level mechanisms are finally used. In our prototype, all node-local communication between the monitor process and the application processes is handled via shared memory based information exchange. All other communication (especially between node distribution unit and monitor processes) uses PVM as communication means.

Finally, how does the monitor software on the nodes observe and manipulate the running program? There are basically two ways: a) by direct instrumentation of all software involved (i.e. of application processes, daemons, etc.) and b) by standardized low level interfaces (such as the `/proc` file system, `ptrace()` functions, etc.). Both methods are inevitable and will be employed in OCM.

All major aspects of our OCM implementation will soon be published in a separate design document.

7 Status of the Work

The OMIS project started in summer 1995 together with Vaidy Sunderam from Emory University, Atlanta. In February 1996 we released OMIS 1.0 [7]. After many discussions with other researchers OMIS 2.0 was issued in July 1997. The design for the implementation of OCM started in 1996, implementation in 1997. A first prototype will be functional in summer 1997.

Starting with OMIS 1.0 we tried to initiate cooperations with other researchers on this topic. Currently, we have closer contacts with research groups in Lyon, Linz, and Budapest to employ OMIS/OCM for their purposes.

8 The Next Steps

After the publication of OMIS 2.0, the most important task is to produce a stable implementation of OCM and to make it publicly available. Also, our already available tools must be adapted and tested.

From the more scientific point of view we will work on concepts for the integration of interactive and automatic tools. Two variants are first candidates: a) interoperable debugging and checkpointing tools in order to restart debugging sessions from any available checkpoint and b) performance analysis and load balancing tools in order to investigate load balancing mechanisms and their influence on program efficiency.

In future we will adapt OMIS to other target architectures, specifically to shared virtual memory systems. New objects and services have to be added to cover the corresponding programming models.

Acknowledgements

We would like to thank our student team members Alexander Fichtner, Manfred Geischeder, Carsten Isert, Andreas Schmidt, Michael Uemminghaus, and Hans Zeller for their contribution within the framework of the OMIS and OCM project. The implementation is managed by Jörg Trinitis who deserves our deep respect for his patience and care. Furthermore we would like to thank everybody who helped us with critical comments and feedback on our project. There were many of them. We did our best to try to integrate everything useful into version 2.0 of the OMIS document.

Request for Cooperations

We would very much like to encourage other researchers interested in this work to get in contact with us. We are especially interested in learning how to improve OMIS and make it more appropriate for all sorts of tools. Furthermore, if you would like to use OMIS/OCM for your own tools, do not hesitate to tell us so.

References

1. J. Casas, D.L. Clark, P.S. Galbiati, R. Konuru, S.W. Otto, R.M. Prouty, and J. Walpole. MIST: PVM with Transparent Migration and Checkpointing. In *3rd Annual PVM Users' Group Meeting, Pittsburgh, PA*, Pittsburgh, May 1995.
2. J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol. RFC 1157, SNMP Research, PSI, MIT, May 1990.
3. P. Fritzson, R. Wismüller, O. Hansen, J. Sala, and P. Skov. A Parallel Debugger with Support for Distributed Arrays, Multiple Executables, and Dynamic Processes. In T. Gyimóthy, editor, *Compiler Construction. Proceedings*, volume 1060 of *Lecture Notes in Computer Science*, pages 341–355, Linköping, Sweden, April 1996.
4. R. Hood. The *p2d2* Project: Building a Portable Distributed Debugger. In *Proc. of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 127–136, Philadelphia, Pennsylvania, USA, May 1996. ACM Press.
5. A.M. Julienne and B. Holtz. *ToolTalk & Open Protocols — Inter-Application Communication*. A Prentice Hall Title. SunSoft Press, Englewood Cliffs, NJ, 1994.
6. S. Kille and N. Freed. Network Services Monitoring MIB. RFC 1565, ISODE Consortium, Innosoft, January 1994.
7. T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. OMIS — On-line Monitoring Interface Specification. Technical Report TUM-I9609, SFB-Bericht Nr. 342/05/96 A, Technische Universität München, Munich, Germany, February 1996.
8. T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. OMIS — On-line Monitoring Interface Specification (Version 2.0). Technical Report TUM-I9733, SFB-Bericht Nr. 342/22/97 A, Technische Universität München, Munich, Germany, July 1997.
9. B. P. Miller, J. M. Cargille, R. B. Irvin, K. Kunchithapadam, M. D. Callaghan, J. K. Hollingsworth, K. L. Karavanic, and T. Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 11(28), November 1995.
10. G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the International Parallel Processing Symposium*, pages 526–531, Honolulu, HI, April 1996. IEEE Computer Society Press, 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos,CA 90720-1264.
11. G. Stellner and J. Trinitis. Load balancing based on process migration for MPI. In *Euro-Par'97*, Lecture Notes in Computer Science; to appear, Berlin, August 1997. Springer.
12. R. Wismüller and T. Ludwig. THE TOOL-SET – An Integrated Tool Environment for PVM. In H. Lidell, A. Colbrook, B. Hertzberger, and P. Sloot, editors, *Proc. High-Performance Computing and Networking*, volume 1067 of *Lecture Notes in Computer Science*, pages 1029–1030, Brussels, Belgium, April 1996. Springer Verlag.
13. R. Wismüller, M. Oberhuber, J. Krammer, and O. Hansen. Interactive debugging and performance analysis of massively parallel applications. *Parallel Computing*, 22(3):415–442, March 1996.