# OMIS — On-line Monitoring Interface Specification [1]

## Version 1.0

http://wwwbode.informatik.tu-muenchen.de/~omis/

email: omis@informatik.tu-muenchen.de

Thomas Ludwig, Roland Wismüller,
Vaidy Sunderam[†], Arndt Bode

Lehrstuhl für Rechnertechnik und Rechnerorganisation
Institut für Informatik (LRR-TUM)
Technische Universität München
D-80290 München, Germany
tel.: +49-89-2105-2042 or -8164 or -8240
fax: +49-89-2105-8232
email: {ludwig,wismuell,bode}@informatik.tu-muenchen.de

[†]Emory University
Mathematics & Computer Science
Atlanta, Georgia 30322
tel.: +1-404-727-5926, fax: +1-404-727-5611
email: vss@mathcs.emory.edu

February 1, 1996

[1]This work is partly funded by the German Science Foundation, Contract: SFB 342, TP A1

## Abstract

The On-line Monitoring Interface Specification (OMIS) aims at defining an open interface for connecting on-line software development tools to parallel programs running in a distributed environment. Interactive tools like debuggers and performance analyzers and automatic tools like load balancers are typical representatives of the considered class of tools.

The current situation is characterized by the fact that tools either follow the off-line paradigm by only having access to trace data and not to the running program or else they are on-line oriented but suffer from the following deficiencies: they do not support interoperability in the sense that different tools can be used simultaneously — not even tools from the same developer. Furthermore, no unified environment exists where the same tools can be used for parallel programs running on different target architectures.

A reason for this situation can be found in a lack of systematic development of monitoring systems, i.e. systems which provide a tool with necessary runtime information about the application programs and make it possible to even manipulate the program run.

The goal of the OMIS project is to specify an interface which is appropriate for a large set of different tools. Having an agreed on on-line monitoring interface facilitates the development of tools in the way that tool implementation and monitoring system implementation are now decoupled. Bringing $n$ tools to $m$ systems (consisting of hardware, operating system, programming libraries etc.) will be reduced in complexity from $n \times m$ to $n + m$. In addition, it will eventually be possible to simultaneously use tools of different developers and to compose unified tool environments.

The research group at LRR-TUM will implement an OMIS compliant monitoring system for the PVM programming model running on a network of workstations. Several interactive and automatic tools will be connected to this concrete system.

The present document defines the goals of the OMIS project and list necessary requirements for such a monitoring system. We will describe the system model OMIS is primarily intended for and give an outline of available services of the interface. A special section will give details on how to extend OMIS, as this is an indispensable feature for future tool development.

We would appreciate to get feedback on the design of OMIS. If you would like to see special issues incorporated into this specification document you are invited to contact the authors.

# Contents

# Preface to this Document Version

This document version is an extensive reorganisation and improvement of previous releases. Part I describes motivation, project goals, and requirements of the project. Part II comprises a description of the structure of the monitoring interface. It describes the underlying system model, what services are, and how to extend OMIS. Part III gives a formal description of services provided by the monitoring interface. Part IV describes concepts for the implementation of an OMIS compliant monitoring system.

*We would like to encourage people to send us feedback on the "Requests for Comments" and also maybe on "Known Problems". Any further ideas, comments, criticism etc. are also welcome.*

Please see also chapter History for the document history.

# Part I

# The OMIS Project

# Chapter 1

# Motivation

Parallel processing is a key technology for the 21th century both for commercial/industrial applications and for research. Current needs of computational power can only be satisfied by using parallel and distributed architectures like multiprocessor and multicomputer systems. For already several years also networks and clusters of workstations (NOWs, COWs) play an important role, as there aggregate power can often meet user requirements. We can distinguish systems by their architectural concepts like e.g. coupling of processing elements (busses, switches etc.) or memory organization (distributed memory, shared memory). Various programming paradigms can be used for implementing software for these systems, message passing and usage of shared memory segments being the most popular amongst them.

The approach presented in this document will in its first phase concentrate on systems with distributed memory architecture and programs being implemented with message passing libraries. Developing these programs on such systems is usually considered to be the machine language level of parallel programming; it might exploit a high potential of the system's theoretical computational power but exhibits also a great complexity with respect to coding, debugging and performance optimization. The goal of the OMIS[1] project is to undertake steps in reducing this complexity by providing better support for tools for these environments. Future work will try to transfer the concepts to shared memory architectures.

The available support for parallel programming in environments with distributed memory varies considerably in quality and quantity. Results of the Second Pasadena Workshop on System Software and Tools for High Performance Computing Environments [SMP95] show that main problems are a lack of sophisticated tools and the almost complete absence of any interoperability of tools of different origin. Also there are no uniform environments at all, where the developer could use the same tools for different types of hardware or parallel programming libraries.

On most systems we can find simple tools for debugging and runtime management like e.g. processor allocation. More powerful tools or special tools like for example load balancing facilities or support for fault tolerance mechanisms do rarely exist. In addition, only very few tool environments support design, implementation, and maintenance of parallel software in a consistent manner. Finally, tools of different vendors do not interoperate because they are based on e.g. different monitoring techniques or trace data formats or just use proprietary programming libraries or special adaptions of publicly available programming libraries. Usually, the programmer can use only one tool at a time or even must specially adapt his program before applying another tool.

Tools which support an investigation of the running parallel program can be divided into on-line and off-line tools. Off-line tools exclusively support post-mortem inspection of the program behavior. The drawback of this approach is its lack of interactive program manipulation facili-

[1] The acronym OMIS stand for **O**n-line **M**onitoring **I**nterface **S**pecification.

ties and its delay between problem recognition and problem correction. On-line tools, however, support interactive program manipulation with an immediate feedback to the user thus shortening the time spent for debugging and performance analysis. In addition, special automatic tools like e.g. load balancer systems, must necessarily interact with the running program. An off-line concept is not feasible for them.

Both tool approaches are based on different classes of information which are collected by different mechanisms. Off-line tools typically offer information which was collected during a program run (trace data) or after program completion (core dump). Trace data is gathered with a monitor component being introduced into the program, the runtime libraries, the operating system, or even the hardware. The monitor's task is restricted to only collecting information about the system behavior and forward it to a file for storage. In addition, as we do not know in advance which information the user would like to evaluate, all possibly interesting data have to be transferred to this file. We prefer to call this mechanism "recording" instead of "monitoring". For long program runs or fine grained information this is not a feasible approach as it requires to collect an enormous amount of data.

With on-line tools the situation is more complex. In addition to above mentioned data (trace data, core dump) we need information about the current program state. Moreover, as the user evaluates this data immediately, additional tasks have to be added to the monitor's responsibility: First, it must be adaptable in the sense that new evaluations can be activated while others will be stopped. (This, however, decreases dramatically the amount of data which has to be transferred to the tools.) Second, the monitor must be able to manipulate the program, e.g. to stop a task or to force it to single step mode. A software component which supplies this class of functionality will be called on-line monitor. The goal of this paper is to lay the basis for the design (and also the implementation) of on-line monitors for parallel environments with distributed memory.

Currently, several on-line tools are already available for the above mentioned environments, but most of them are proprietary solutions of parallel computer manufacturers. Public domain or copy-left tools are for the most part only available for programming libraries like e.g. PVM or implementations of MPI. Any activity in this field is hindered by the fact that for every new tool, for every new hardware, and for every new implementation of a programming library a new monitoring system has to be developed.

The monitoring system is the software layer which connects the running program system (consisting of hardware, operating system, and program processes) to the tool environment and guarantees on-line capabilities. Caused by its intermediate position it has two interfaces: one to the tools and one to the running program. Up to now, there are no approaches undertaken to standardize at least the monitor/tool interface. Without a reasonable standard, however, new tools must also have new monitoring systems even if their functionality is not completely disjunct to already existing tools. Finally, the missing standard makes the integral use of a set of tools impossible as they are currently always based on differently implemented monitoring concepts which are incompatible to each other. For most of the on-line tools the monitoring system itself is the software layer which poses the greatest implementation difficulties as it controls a spatially distributed environment by means of a centralized interface. The monitoring system is by itself a distributed software system and would already require sophisticated tools for its implementation.

If we compare different tools we will find that they use a considerable part of the monitor interface which is identical for all of the tools. For some tools like e.g. an interactive performance analyzer and an automatic load balancer system the functionality might even coincide. For that reason it is desirable to design a monitor system which can be adapted to different underlying hardware/software environments and can be expanded for connecting it to new tools. The goal of this paper is to develop a monitoring concept with a high degree of flexibility and extendibility.

Having an agreed on on-line monitoring interface, different tool developer groups could design and implement new interactive and automatic on-line tools whereas other groups could do implementations of the monitors for various hardware/software environments. The amount of effort for having $n$ tools on $m$ systems (being composed of hardware, operating system, and runtime library of the programming environment) would be reduced from $n \times m$ to $n + m$ thus bringing more tools onto more parallel systems which finally would ease software development on these architectures. A further implication of having OMIS compliant monitoring systems is that finally we will reach the goal of having uniform environments, i.e. tools which are identical for a variety of target architectures.

# Chapter 2

# Project Goals

## 2.1 Background

The OMIS project being described in this paper has not been started as an isolated project although there might be a real necessity for such an approach. It is embedded into research and development activities at the Lehrstuhl für Rechnertechnik und Rechnerorganisation at Technische Universität München (LRR-TUM). Before going into details with OMIS let us give some information on the background.

During the last eight years the parallel processing group at LRR-TUM has been working in the field of interactive and automatic on-line tools for parallel programming. Starting point was the TOPSYS project which was funded by a special research grant of the German Science Foundation. TOPSYS is an acronym which stands for TOols for Parallel SYStems; for detailed information about it please refer to [BBB+90, BB91, Lud93b].

Within the framework of TOPSYS we developed a set of tools for Intel iPSC hypercube computers. A debugger [BW95], a performance analyzer [BHL90], and a program flow visualizer [BB92] were the main interactive components. In addition, we investigated tools for automatic load balancing [Lud93a]. The environment was based on our proprietary programming model MMK (multiprocessor multitasking kernel) [BL90]. Tools were using on-line monitoring systems which were realized with identical functionality in both, hardware and software [BLT90].

Already in parallel to TOPSYS we designed and implemented tools within frameworks of other projects and direct industry cooperations. Examples are an adaption of TOPSYS to workstation clusters (cooperation with SUN Microsystems), adaptions of the performance analysis tool in the Esprit project PREPARE (an on-line version) and the Esprit project HAMLET (an off-line version) as well as smaller cooperations with e.g. INTEL, Siemems, Genias, and others. (For more details please refer to the annual report of LRR-TUM.)

For already several years a very important cooperation links LRR-TUM and PARSYTEC, a German vendor of parallel supercomputers and embedded systems. Within that project we designed and implemented versions of our debugging and performance analysis tools especially adapted for PARSYTEC parallel systems [Han94, OW95, HKOW96]. Both tools are successors of former TOPSYS tools and became an integral part of the PARIX operating system for PowerXplorer systems in 1994. Currently we adapt both tools to the new CC systems running EPX, a special version of PARIX for embedded systems. Again, the main effort concerns an appropriate port of the underlying monitoring system to the now used AIX operating system.

In the last years a change in paradigm took place: multiprocessors systems are no longer the only vehicles of parallel processing. In addition, workstation clusters enjoy an increasing popularity. The style of programming (with respect to message-passing) did not change significantly. However, the environment structure increases complexity: time sharing replaces or adds to space sharing thus making it necessary to have appropriate development tools. The main difference is the step from single user/single program environments to multiuser/multiprogram environments.

As a reaction to that the parallel processing group at LRR-TUM started two new projects in 1995 namely the OMIS project and THE TOOL-SET project (see [LWB+95]). Before going into details with the project goals of OMIS let us give a quick overview on the latter project. Its global goal is to design and implement an integrated environment of various tools to make cluster computing easier. All implementations will initially be based on PVM which represents the current de facto standard for parallel programming. The PVM library and runtime environment is available for all major workstation brands as well as for all important multiprocessor and multicomputer systems. PVM supports the main aspects of distributed computing: work distribution (by process management mechanisms) and cooperation (by message passing mechanisms). In the first project phase, THE TOOL-SET will be made available for workstation clusters only. Adaption to genuine parallel architectures might follow in the future. THE TOOL-SET will comprise a set of interactive and automatic on-line tools such as a debugger, a performance analyzer, a visualizer, a tool for deterministic program execution, a dynamic load balancer, a consistent check-pointing facility, and a trace data comparison tool.

## 2.2 Goals

From the above paragraphs we see that the monitoring system is a main issue for every tool environment with on-line tools like e.g. THE TOOL-SET and the CC series tools. As already mentioned earlier, it must offer the following functionality:

- It must be able to extract data describing the current state of the HW/SW-system (hardware, operating system, application programs) on request and on a regular basis.
- It must be adaptable in the sense that the user can define which data should be monitored (e.g. the occurrence of user-specified conditions).
- It must be able to modify and influence the HW/SW-system (e.g. assign values to variables, stop and restart process execution)

The central goal of the on-line monitoring interface specification OMIS is to provide means to efficiently design and implement monitoring systems which fulfill above mentioned requirements and to define a standardized tool/monitor-interface. With an OMIS compliant monitoring system being connected to a running HW/SW system, several tools from possibly different developers can concurrently manipulate application programs. Tool interoperability and integration of future tools into existing environments are the greatest features OMIS will be able to support. In addition, we will reach the goal of having uniform environments where identical tools exist for a variety of target architectures.

The detailed list of goals comprises research oriented issues, design and implementation issues, and standardization issues. The project will be driven by people who were already involved in the design of TOPSYS thus making it possible to take profit of existing long year experiences in that field.

The central research topic is to investigate on-line monitoring methodologies for parallel systems and to achieve a deeper understanding of the issues involved in tool interfaces for parallel and distributed computing. Especially the interaction of the monitor with all other components of the system (hardware, operating system, application programs) and its possible and necessary interconnections with them will be carefully studied. Furthermore, adaptability is a big concern. Although the project will lead to a realization for a concrete set of tools, programming libraries, and operating systems, we will concentrate on the question of how to keep the interface specification abstract enough to guarantee its applicability to various other environments.

The major objective is to define a tool/monitor-interface which meets two main requirements: First, it should be extensive and complete in a sense that the functionality of all common type of tools (including of course THE TOOL-SET) will be guaranteed. Second, as there will be new tool functionalities in the future or even completely new tools, the interface must be extendible in a well defined manner. Also other research groups must be able to use the approach and adapt it to their needs.

A first version of the specification will be published in newsgroups and at workshops. We expect to get feedback from other tool designer groups giving us details what type of interface is necessary to meet their special requirements. From that we will produce a refinement of OMIS.

Starting from a sophisticated specification document two further goals have to be achieved. First, we will implement an OMIS compliant monitoring system which serves as a basis for THE TOOL-SET. In more detail this means to eventually have an on-line monitor for PVM running on workstation clusters where tools developed at LRR-TUM and at other sites can be used concurrently with the same application programs.

Second, if the approach proves to be powerful enough and proves to be a viable basis for making tool design and implementation easier and less time consuming, we will support OMIS to become a new standard in the world of tools for parallel systems. The parallel processing group at LRR-TUM will coordinate extensions to OMIS being brought in by other research groups. Thus a reliable standard will exist, for which other groups can do both, develop tools and implement compliant monitoring systems for specific parallel architectures.

The project policy will be to release all software products being developed in the framework of OMIS and THE TOOL-SET under GNU license conditions to provide a maximum profit to the user community.

We would like to strongly encourage other researchers working in the field of parallel programming environments and tools to participate in this project by discussing with us their special needs or wishes and making critical comments to this proposal.

# Chapter 3

# Requirements

The design of the monitoring interface imposes several requirements which the specification will have to meet. These requirements can be divided into the following categories:

- functional requirements
- conceptional requirements
- efficiency requirements.

Their scope is limited to the tool/monitor-interface as this is what we would like to specify. Further requirements will arise for the implementation of an OMIS compliant monitoring system dedicated to a given system architecture. However, they will not influence the tool/monitor-interface but the monitor's internal structure and the monitor/program-interface.

A general requirement standing above these three categories is derived from the goals of the project itself. The monitoring interface must be powerful enough to give on-line tools an efficient access to the programming system. The sum of its functions and its conception will have to allow the integrated application of different tools at the same time. Also it must guarantee future adaptability to more sophisticated tools (e.g. problem domain oriented tools).

### Functional Requirements

Functional requirements can be summarized as follows: the monitoring interface should be versatile enough to allow all possible tools to observe and manipulate all objects of the running program (e.g. processes, messages, variables etc.). Obviously, we can not state requirements for tools which might be of interest in the future. Therefore, the requirement list primarily addresses well known tool types like debuggers, performance analyzers, program flow visualizers, checkpointing facilities, and load balancing components.

In order to achieve the desired degree of versatility it is not sufficient that the monitor interface just offers a set of services which can be requested. It should rather support any service request complexity. This is attained by service request composition. Service request composition will be used for more complex operations, e.g. measurement activation in dependence of the system activity. Also this feature supports new and more abstract functions to be realized in a tool. Depending on the application type a tool might want to measure performance values related to semantical constructs like e.g. iterations of a numerical algorithm or transactions in a database system. Service request composition will guarantee the usability of the monitoring interface for future tools.

Concerning objects types which we might like to monitor we can state the following requirements: As implementations of monitoring systems will be realized for distributed memory environments we will definitely be interested in processes and messages. Functions of the interface should give access to these object types on several levels of abstraction: e.g. with processes we might want to look at procedures or even individual statements of its code, with messages we would like to know, from which primitive data types they are composed. Furthermore, interactive tools require these objects not only to be observable but also need manipulation functions, e.g. for stopping of process execution.

In addition to this we also need access to hardware objects of the system. Inquiry functions might want to have information on the amount of available or used main memory or on some architecture characteristics like e.g. technology of installed network devices. These are observed objects which can not be manipulated by the monitor. Finally, the existence of the monitor creates also new objects, i.e. monitor objects, which a tool must be able to interact with. Especially routines for monitor-monitor interaction or filter mechanisms must be accessible. New objects will appear with extensions of the tool/monitor-interface. These user defined objects have to be specified separately and will not be integrated into the main part of OMIS[1]. Later chapters will however show, how to introduce these extensions.

Any of the tools will finally interact with all these objects in one or the other way. The interface's task is just to provide an appropriate means for this interaction. Let us give some examples which types of functionality is required by individual tools, using THE TOOL-SET as an example:

- THE DEBUGGER[2]

  Show process status information

  Set breakpoints on reception of a message

- THE PERFORMANCE ANALYZER

  Measure node idle time and process CPU utilization

  Include process into measurement if certain criteria are met

- THE VISUALIZER

  Show dynamically created objects

- THE LOAD BALANCER

  Evaluate system's load distribution and trigger process migration

These examples should make it evident that the design of a versatile functionality of the tool/monitor-interface is of crucial importance.

### Conceptional Requirements

As we neither know the complete functionality of the tools in advance nor the types of tools themselves we have to require that the monitoring interface offers enough extendibility for future developments. This conceptional requirement will be fulfilled with our specification by designing means of how to enhance the specification.

A second conceptional requirement is imposed by the variety of tools which will use this interface. We can distinguish tools with and without a graphical user interface. The first

[1] Obviously, an object oriented approach for implementing a monitoring system seems to be favorable. Software objects like 'processes', 'PVM tasks', 'threads' can be handled by classes derived from some abstract class 'computingObject'. Methods can be similar for all derived classes.

[2] For a brief description of what the functionality of the following four TOOL-SET-components will be, please refer to [LWB+95]

group will usually have a single point of control, e.g. a tool environment on a workstation which communicates with all monitors of the system via a selected dedicated master monitor. On the other hand we will have tools without user interface which reside in the system in a centralized or decentralized version. Decentralized tools are for example load balancer systems. The monitoring system must be flexible enough to serve all these different types together with their different spatial distribution of control.

### Efficiency Requirements

Finally, we have efficiency requirements. Although it might seem to be an improper approach to discuss interface specifications in terms of the efficiency of their potential implementations we will see that there are good reasons to discuss this issue. Communication between a monitor and a tool must be handled by a kind of message passing mechanism. In order to keep the overhead minimal it is necessary to have powerful basic services and a possibility of composing service requests into a single request.

In addition to being a functional requirement composite service requests are necessary for efficiency reasons. They replace a fixed sequence of already available basic services. E.g. to get an overview over the system utilization the tool could ask each monitor for the idle time percentage. Instead, it will ask one monitor which sends requests to all others and returns a collective response to the calling tool. These types of composite services are of special interest for performance analysis where a measurement value is logically a computation of several primitive values. In this case the monitor itself can do some pre-calculation in order to reduce message passing overhead.

In order to avoid delays due to communication, the monitoring system should also be able to handle certain kinds of events occuring in the application program without interacting with the tool. If a performance analysis system wants to measure the mean time between e.g. sending a message and receiving the reply, it would be prohibitively expensive to inform the tool on each event occurrence. Instead, the monitoring system should be able to start and stop a timer autonomously. Service request composition can also support this situation.

# Part II

# Structure of the Monitoring Interface

# Chapter 4

# The System Model

This chapter describes the model of the target systems for which OMIS is primarily designed for and also the embedding of an OMIS compliant monitoring system into such a system. In our example, the environment is composed of a parallel programming library and an additional specialized runtime library (e.g. for handling parallel I/O operations). Programs consist of a collection of tasks which usually spawns over a set of nodes.

Thus, this paper does not only present an interface specification, but also specifies the kind of environment that can be handled by the interface. Although the specification exhibits a high degree of flexibility, we can not expect to be able to integrate it in every architectural environment, especially if they differ considerably from those found with the message passing paradigm.

Let us now look at the components participating in such an environment. Figure 4.1 shows an abstract view where individual nodes of the parallel system are not yet visible. The application programs consist of a certain number of tasks which communicate by mechanisms provided by the parallel programming library (e.g. message passing between nodes, shared memory on one node). Task management and other organizational work is performed by special modules of the programming environment (e.g. daemons). It might for example be based on the parallel programming library PVM and the specialized runtime library PFSLib[1]. Technically, this means that there might be libraries and daemons with which the program works together. The complete complex runs on top of the operating system and the hardware.

As soon as we add tools to this ensemble (either interactive or automatic tools) we need additional layers. The part which joins the tools to the running program is just the monitoring system. Its role is to establish the tool/program-interaction. Consequently, this layer is located between the application program and the tool.

Interactive tools usually reside on a host machine which is connected with the target system[2]. With automatic tools like e.g. load balancers, the situation is different. They exist in a distributed manner on the target nodes only. We will therefore call them a distributed tool. Two further modules are of interest[3]. The distributed tool extension (DTE) is a set of user supplied functions to perform certain manipulations outside of the centralized tool (e.g. calculate certain performance metrics, write traces to local disks, etc.). Finally, we might have monitor extensions (ME). These are extensions of the monitoring system and its specification which are dedicated to a new software component like e.g. a parallel file system (PFSLib). It highly depends on the concrete tool environment which of these three additional components are available in a given system. However, if there is an interactive tool then there will also be in almost any case a

[1] PFSLib is a parallel file system for workstation cluster environments. See [] for details.

[2] Note, that with many modern parallel computers and with workstation clusters the distinction between host and target machine is obsolete.

[3] Please refer also to chapter 6 for more details on these additional components.

distributed tool extension because many activities can more efficiently be handled directly on the nodes and not in the central tool (e.g. preprocessing of information data).

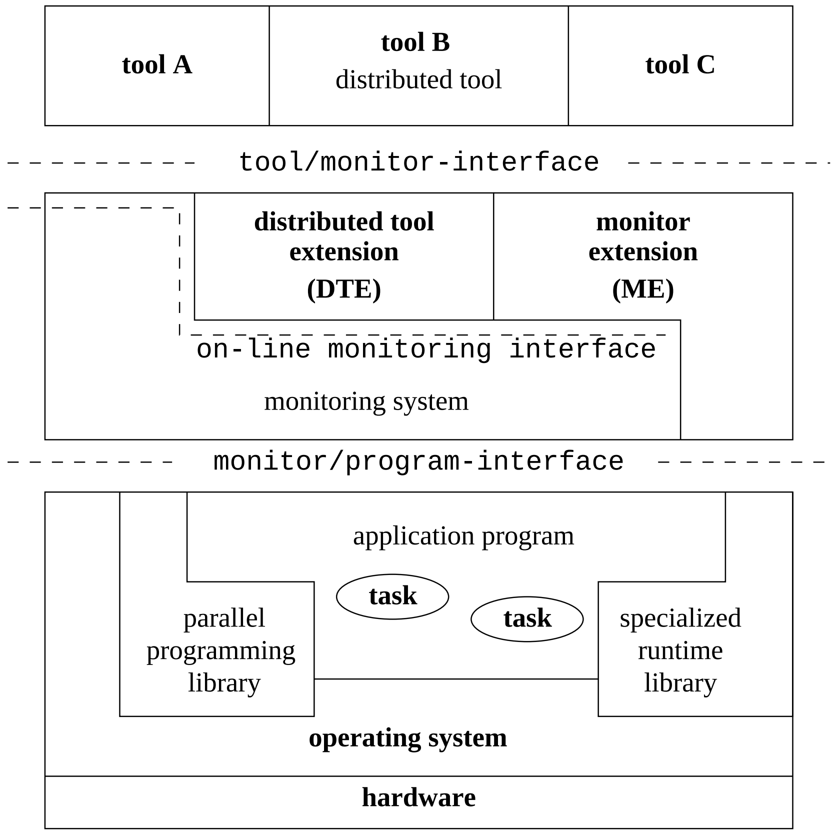


Figure 4.1: System model: embedding an OMIS compliant monitoring system into an environment with tools and a parallel programming library as basis

How do the individual layers of figure 4.1 inter-operate? The monitoring system has two interfaces: one for interaction with the different tools (tool/monitor-interface), a second one for interaction with the program and all underlying layers which keep the program running. For simplicity reasons we will call the latter the monitor/program-interface although it comprises interface parts to different modules of the system (program code, libraries, operating system). Activities at this level are restricted to low-level inspection and manipulation requests. Obviously we will have to handle different information types depending on the concrete low-level module with which we inter-act. Details will be discussed in later chapters.

The tool/monitor-interface is what the developer of a tool finally will use. However, OMIS makes a specification of semantics only for the on-line monitoring interface. In addition, means are designed to extend this interface. We end up with a definition of the tool/monitor-interface where the native services provided by the on-line monitoring interface are defined with their syntax and semantics whereas all extensions are only defined with respect to their syntactical structure.

The interaction between tools and the monitoring system is handled via asynchronous remote procedure calls. The tool invokes a service request and waits for results coming back from the monitoring system.

The interaction between the monitoring system and the additional components monitor extension and distributed tool extension happens via function calls and activation of call-back functions.

# Chapter 5

# A Basic Outline of Available Services

Since a central idea of OMIS is to provide an interface that allows complex requests to be built by combining primitive ones, the tool/monitor-interface is based on strings in order to achieve the needed flexibility. Thus, the tool/monitor-interface conceptually consists of only a single procedure that can be invoked by both centralized tools and distributed tool components. This procedure receives a string as an input parameter, interprets this string, and returns a result which again is a string[1]. The individual monitoring functions available by invoking this procedure are called *services*; the string that is passed to the procedure (requesting the activation of a service) is called *service request*, the result is called *service reply*.

In order to meet the efficiency requirements stated in Chapter 3, the structure of a service request follows the event-action-paradigm, allowing the monitoring system to quickly react on state changes in the monitored system without having to communicate with the tool. A service request can consist of an event definition X associated with an action list Y, meaning "whenever an event of type X occurs invoke the actions in Y, passing some relevant information on the event occurrence to these actions. If no event definition is present, the service request is unconditional and all actions are invoked immediately and exactly once.

The interface procedure accepts requests for all monitors in the monitoring system; therefore, a distributed tool component can request services not only for its local node, but also for any other node. Likewise, the actions associated with an event can be requests for services on nodes different from the one where the event occurs. The monitoring system automatically takes care of forwarding the requests to the proper nodes. In fact, requesting a service for a remote node is exactly the way how monitor/monitor communication is used. In addition, we allow services that are global, i.e. that involve more than one monitor.

The next section introduces the different classes of services available at the tool/monitor-interface; Section 5.2 provides a basic outline of the mechanisms and the syntax used in this interface. Section 5.3 presents two examples giving an impression of the interface's expressiveness. A detailed description of the interface procedures is contained in Section 5.4. Section 5.5 finally presents some additional remarks on the tool/monitor-interface.

## 5.1 Classification of Monitoring Services

The services that are offered by the tool/monitor-interface can be classified according to three different properties:

- First, we have to distinguish between *basic services* and *composite services*. Basic services are those monitoring functions that are built into the monitor and form the building blocks

[1] To support efficient usage there should be a utility library containing functions to assemble and parse these strings. However, the specification of such a library is not part of this document.

for the (composite) service requests that can be sent to the tool/monitor-interface. As stated above, a service request is composed of an event definition and associated actions. Here, the event definition and each of the actions are basic services, while the whole service requested is composite. However, when the meaning is clear from the context, we will simply speak of a service in this document.

- Further, we can classify both basic and composite services according to their input/output behavior:

  1. **Manipulation services**. These services receive some input parameters from the caller, but will not have a result, except for an acknowledge or an error code. Thus, their only effect is to change the state of the monitored system by manipulating objects in the application or in the monitor itself. Examples are "stop a process", "set a variable of a process to a given value" or "raise a user defined event".
  2. **Synchronous services**. These services receive some input parameters and immediately return a result exactly once. The result will contain information about the current state of the monitored system (including the monitoring system itself). Examples are "return the list of valid task identifiers on a node" or "return the CPU time of a process".
  3. **Asynchronous services**. In contrast to the two classes above, asynchronous services can have an arbitrary number of replies. Usually, neither the number of replies nor the time when replies will be returned is known when the service request is issued. These services are of type: "whenever X happens, do something".

  The description of basic services in Section 8 is structured according to these classes.

  The interpretation of an asynchronous service is somewhat dependent on whether we have a basic service or a composite one. A basic asynchronous service only defines an event, i.e. when something has to be done, but it does not define what has to be done once the event occurs in the system. Thus, in contrast to basic manipulation/synchronous services, a basic asynchronous service cannot be used in isolation, but only in combination with an action definition, forming a composite asynchronous service.

  Therefore, composite asynchronous service requests always consist of a basic asynchronous service (*event definition*) and a list of basic manipulation/synchronous services (*action definition*). They are mainly used to monitor the occurrence of the defined event in the application, since they trigger the execution of services (the actions) without tool interaction. An example is a request to stop a task whenever it tries to send a message.

  Throughout this document, we use the terms "event definition" and "basic asynchronous service" interchangeably. In addition, we will simply speak of an event when the context makes clear whether we mean an event definition or an event occurrence.

- Basic services can further be classified according to the type of objects they refer to. On the first level, we distinguish between application objects and monitor objects. Currently, three different kinds of application objects are supported:

  1. **processes**, e.g. PVM tasks,
  2. **messages**, including synchronization objects, and
  3. **hardware**, i.e. the parallel or distributed computing system.

  More special application objects, such as groups and barriers in PVM, are not yet included in this specification. We will add them in a later stage of the OMIS project in form of a monitor extension. In Part III, we will use parallel I/O as an example for this kind of extension of the monitoring interface that aims at observing additional objects.

Monitor objects are those objects that are introduced by the monitoring system itself. For example, each asynchronous service is a monitor object, since it has to be stored in the monitoring system and can be manipulated using other services. Other monitor objects are user-defined events or timers and counters, although the latter are not included in the basic specification.

Since this classification of the monitored system into a hierarchy of objects is a natural way of structuring the monitoring services, we are thinking towards the future use of an object oriented paradigm for the tool/monitor-interface instead of the current procedural one. By exploiting inheritance, object oriented techniques could also provide a way to define the interface at an abstract level independent of the supported programming library. Services specific to a programming library could then be realized by an extension to the generic monitor that implements object classes (e.g. PVM task) derived from the interface's base classes (e.g. abstract process). See item 1 in our requests for comments (Chapter 12).

## 5.2 Syntactical Structure of Monitoring Services

This section is intended to give an impression how the syntax of service requests and service replies looks like and how basic service requests can be combined to more complex ones. For a complete formal description of the request and reply syntax, please refer to Section 7.

### 5.2.1 Service Requests

#### Basic Service Requests

A basic service request is a string that complies with the following syntax:

$$\textit{basic\_request} ::= \textit{request\_id} \text{ ' ' } \textit{receiver\_nodes} \text{ ' ' } \textit{service\_name} \text{ '(' } \textit{input\_parameters} \text{ ')'}$$

The *request_id* is an arbitrary integer number defined by the tool. It will be sent back to the tool with every reply to this service request, so that the tool can uniquely associate the replies with the corresponding requests when several basic service requests are combined to a composite request.

An arbitrary list of nodes in the monitored system can be specified in *receiver_nodes*. The request will be sent to every node in the specified list. If the list is empty, the request will be broadcast to all nodes. The sending of requests will make use of an atomic multicast protocol to ensure that the monitoring system always remains in a globally consistent state. If two requests are sent from different sources to a common set of destination nodes, this protocol ensures that each node in the set will receive them in the same order. Thus, if a request for a global **stop** (i.e. stop all processes), and another one for a global **continue** (i.e. resume all processes) are issued nearly simultaneously, then as a result either all processes are stopped or all processes are running.

In order to simplify the handling of nodes within tools and within the monitoring system, nodes are identified by numbers instead of host names. There is a service **list_hosts** that provides the correspondence between the host names and their node numbers.

The *service_name* is a string identifying the requested basic service. The monitoring system uses an internal table to map the service names to the functions that actually implement the services. This mapping table is the key to the interface's extendibility (see Chapter 6). Thus, *service_name* may identify both a basic service defined in this document and a basic service coming from an extension.

The *input_parameters* are defined by a list of values where each value is either an integer, a floating point number, a quoted string, a special $-variable (see below) or a lists of these items. The number, the types and the meaning of these parameters depend on the basic service being requested. Chapter 8 contains a complete description.

### Composite Service Requests

The basic service requests defined above can be combined to more powerful, composite requests using the event-action paradigm. The general syntax of such a combined request is:

| | | |
|---|---|---|
| *request* | ::= | [ *event* ':' ] *action_list* |
| *action_list* | ::= | *action* ',' *action* ',' ... \| *action* ';' *action* ';' ... |
| *event* | ::= | *basic_request* |
| *action* | ::= | *basic_request* |

Each composite service request consists of an optional event definition and a list of actions. Actions can only be basic manipulation services or basic synchronous services, while the event must always be a basic asynchronous service. The separator used in the action list defines whether these actions may be executed in parallel (comma) or have to be executed sequentially (semicolon).

If no event definition is present, the action list is executed immediately and the results are sent back to the requester. Otherwise, we have an asynchronous service request that is stored as an explicit object within the monitoring system. Special services then allow to manipulate (i.e. enable, disable or delete) these requests. Initially, asynchronous service requests are in a disabled state, i.e. they are not yet considered by the monitoring system. Once the request has been enabled – either directly by the tool or as a result of executing the actions of another asynchronous service – the monitoring system is looking for event occurrences matching the event definition contained in that request. Whenever a matching event occurs, the request's action list is executed and and their replies are sent to the requester. The monitoring system ensures that the local state of the object (e.g. the process) that triggered the event will not change until all actions are executed, i.e. each action has access to the state at the time the event occured. In addition, each event definition provides a set of output parameters containing information on the actual event occurrence. These parameters can be passed to the actions in the action list using a $-notation. The string $1 refers to the event's first output parameter, $2 to the second one, and so on. The symbol $0 always refers to the number of the node where the event occured.

### 5.2.2 Service Replies

The replies of services are strings having a syntax very similar to that of service requests:

| | | |
|---|---|---|
| *basic_reply* | ::= | *request_id* ' ' *sender_nodes* ' ' *service_name* '(' *result_parameters* ')' |
| *reply* | ::= | *basic_reply* ';' *basic_reply* ';' ... |

Thus, a reply is a list of results, each being assorted with the *request_id* of the proper basic service request, the sender's node number(s), the basic service's name, and its result parameters. For each action in a composite service request there can be one *basic_reply* or more than one, depending on the number of node the action is executed on. Identical results on different nodes may be unified; in this case *sender_nodes* contains the list of node that produced the given result.

The data types that can occur in the result parameters are the same as with input parameters: integers, floating point numbers, strings, and lists.

## 5.3 Examples

In the following two subsections we will present short examples that will show how the monitoring interface supports different tools, namely a performance analysis system and a debugger. Although the basic services will be defined not till later in this document, their semantics should be intuitively clear in the examples, whose primary goal is to give an impression of the interface's structure and expressiveness, rather than its concrete services.

### 5.3.1 Performance Analysis

Assume that a performance analysis tool wants to measure the time spent by task 4178 (located on node 1) 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:

```
10 [1] start_lib_call([4178],"pvm_send"): 11 [1] start_integrator(1), \
                                          12 [1] add_counter(2,$4)
13 [1] end_lib_call([4178],"pvm_send"):   14 [1] stop_integrator(1)
15 [] start_lib_call([],"pvm_barrier"):   16 [$0] trace("barrier0",$0,$1,$2,$3)
17 [] end_lib_call([],"pvm_barrier"):   18 [$0] trace("barrier1",$0,$1,$2)
19 [1] enable(10), 20 [1] enable(13), 21 [] enable(15), 22 [] enable(17)
```

The numbers 10 ... 22 are the request identifiers that will be included in the monitoring system's replies. However, in this example there will be no replies, except on error conditions, because only manipulation services are used as actions. While the events used in the example are basic services defined by this document, all the actions (with the exception of **enable**) come from a distributed tool extension. Thus the performance analyzer can use its own semantics for integrators and counters, and it can also use its own trace format. The trace action simply has to write its parameters to the trace file in the proper format. The string $0 that is used as receiver node of the **trace** action results in each node writing a local trace file, since it will be replaced by the number of the node where the event took place. If you want a single trace file for all nodes, all you have to do is replacing the $0 by a fixed node number, say 0. The monitoring system then takes care of sending the data to that node.

### 5.3.2 Debugging

The following example shows how the basic service requests defined by this document can be used during debugging. The debugger wants to know the task id, the procedure stack, and the contents of all general purpose registers, each time task 1123 (assumed to be located on node 1) starts sending a message, or task 1234 (on node 2) reaches instruction address 0xfe08. This can be achieved by using a user defined event:

```
10 [1,2] define_user_event(5)
11 [1] start_lib_call(1123,"pvm_send"): 12 [$0] raise_event(5,[$1])
13 [2] breakpoint(1234,0xfe08):         14 [$0] raise_event(5,[$1])
15 [1,2] user_event(5):                 16 [$0] print([$1]); \
                                        17 [$0] stack_backtrace($1); \
                                        18 [$0] read_int_regs($1,0,32)
20 [1] enable(11), 21 [2] enable(13), 22 [1,2] enable(15)
```

First, the user defined event #5 is declared on nodes 1 and 2. This event is then raised on the local node whenever task 1123 sends a message or task 1234 reaches the breakpoint. The parameter passed to **raise_event** is the task id. Whenever user defined event #5 occurs, the task id passed as parameter is sent to the debugger (`print([$1])`), followed by the task's procedure stack backtrace (`stack_backtrace($1)`) and its register contents (`read_int_regs($1,0,32)`). After the actions have been executed, the task continues execution. To stop the task with the occurrence of the event, simple add

```
19 [$0] stop($1)
```

to the action list.

When the breakpoint is hit, a possible reply would look like

```
16 [2] print(1234); 17 [2] stack_backtrace(...); 18 [2] read_int_regs(...)
```

where, of course, the proper stack backtrace and register contents are returned instead of the dots.

## 5.4 Interface Procedures

Although the concept of the tool/monitor-interface requires only a single interface procedure, for reasons of convenience there are two of them. The first one, called **OMIS_request_block** blocks its caller until the service reply is available. Although simple to use, this procedure is not suited for asynchronous service requests, since they will result in an arbitrary number of replies. Therefore there is a second flavour of the interface procedure, called **OMIS_request**. This procedure takes an additional call-back function as a parameter. This call-back will then be invoked once for each service reply, allowing to process the replies in a completely asynchronous fashion. The ANSI-C prototypes of these two interface procedures are:

```
char * OMIS_request_block(char *request);

void   OMIS_request(char * request,
                    void (* callback)(char * reply, void * param),
                    void * callback_param);
```

The first procedure accepts a string containing a service request that must be conform with the syntax defined above and returns the request's reply as a dynamically allocated string that should be deallocated by the caller using the **free** library call. Since this function blocks until the reply is available, it is normally not useful for asynchronous services. However, for manipulation services or synchronous services it is convenient to use, since the reply is already accessible when the function returns.

Requesting a service without blocking until its results are available can be done with the second procedure. In addition to the request, you have to pass a pointer to a function and an arbitrary argument pointer. When the service returns a reply, the call-back function will be invoked. It will get the the reply string in the **reply** parameter, and the argument pointer as its **param** parameter. Again, the reply string is allocated dynamically and must be deallocated in the call-back, using **free**. Note that this procedure can be used for all kinds of services (manipulation, synchronous and asynchronous ones) where the caller should not be blocked while waiting for a reply.

## 5.5 Remarks

There are some general remarks about the monitoring interface that should be mentioned here:

- In principle, the interface is asynchronous. This means that with the **OMIS_request** procedure there is no guarantee that you will receive replies in the same order in which you have sent the requests. However, there is one exception. If you send requests for manipulation services or synchronous services to a single node where they can be handled locally, you will receive the replies in the expected order, since the monitors process incoming requests in a FIFO fashion.

- Parameters of services must be either constants, or (in case of an action) output parameters of the corresponding event (i.e. $0, $1, etc.). They cannot themselves be actions or events. Thus no recursion is possible with service requests.

- Events are either predefined (e.g. task starts sending a message) or user-defined. User defined events can be raised by a special service.

- Only one event is allowed in a service request. Combinations of events can be implemented by means of user defined events, the **disable** and **enable** services described in Section 8.2.1, and distributed tool extensions.

  For example, the **or** operator can be implemented by a user defined event. Assume that action list **A(x)** shall be executed on node 1, when event **E1** orrurs on node 1 or event **E2** occurs on node 2. The following sequence of commands will achieve this:

  ```
  10 [1] define_user_event(5)
  11 [1] E1:  12 [1] raise_event(5, [$1])
  13 [2] E2:  14 [1] raise_event(5, [$1])
  15 [1] user_event(5): A($1)
  ```

  A simple kind of distributed event detection can be realized by using the **enable** service as an action. If you want to execute an action list **A** when event **E2** (on node 2) occurs after event **E1** (node 1), the following requests will achieve this:

  ```
  10 [2] E2:  11 [2] A
  12 [1] E1:  13 [2] enable(10)
  14 [1] enable(12)
  ```

  Other combinations (e.g. an **and** operator) can be realized in a distributed tool extension as a service that triggeres a user-defined event when the proper conditions are fulfilled.

- The interface offers no direct way of passing the result parameters of one action to the input parameters of another action. If you need this, you have to code a new service in a distributed tool extension library that calls the actions and passes the parameters between them.

# Chapter 6

# Extending OMIS

As we have already stated before, it is not possible to define a monitoring interface that offers all services that may be needed by any existing or future tool. Therefore, it is of utmost importance to provide a means of extending the interface. OMIS thus includes provisions that allow new services to be added to the basic monitoring system by any research or development group using the monitor, not just by the one that implemented it. The additional services can be implemented as a library of C or C++ functions that is linked with the monitor libraries.

## 6.1 Types of Extensions

There are three situations where linking additional code to the basic monitoring system is profitable or even necessary:

1. A tool may want to observe new objects within an application, which are not covered by this document. These objects may come from specialized runtime libraries. For instance, if an application uses a parallel I/O library, a tool may want to observe I/O objects.

   The code linked to the monitoring system, that provides these new services is called a monitor extension (ME).

2. Usually, different tools use very different methods to process events in the monitored application. For instance, one performance analyzer is based on event traces, thus it wants to write events to a file in its own proprietary trace format. Another performance analyzer is based on distributed on-line analysis. It will therefore need services that perform this analysis, e.g. counting some values, starting or stopping interval timers, and so on.

   Therefore, it is desirable to define new services for this kind of tool-specific data processing. We will call these distributed tool extensions (DTE). They can also include specialized, more complex services that are based on the basic services defined by this document.

3. Finally, there are also fully distributed tools without a central user interface component, e.g. a load balancer. For performance reasons it is profitable that the components of such a tool can access the monitoring interface on their local node via simple procedure calls without the need for interprocess communication.

   We call this kind of "extension" a distributed tool component (DTC).

Of course, a single library can contain code of all these categories.

## 6.2 The Method of Extending OMIS

Once we have decided to allow new services to be linked to the monitoring system, the question is how to make these services accessible at the monitoring interface. The approach we are using is as follows: Each service is addressed indirectly by a unique service name, i.e. a unique identifier string passed to the tool/monitor-interface, which is mapped to the implementing function using a mapping table. Now we have a couple of requirements that must be fulfilled:

- The service names used by different extensions must be disjunct, otherwise we will not be able to use both extensions at the same time. Being able of having multiple extensions is extremely important, since a single version of the monitoring system should be able to support all tools available in a certain environment.
- The name of a service must not change with different versions of the monitoring system. When a group of researchers extends the monitor by providing new services, they must have the same name in every version of the monitoring system, independent of whether or not other extensions are present.
- Besides having unique service names, also the function names and other externally visible names in the code of the extension libraries must be disjunct, since different extension libraries may be linked to the monitor.
- It must be possible to generate and use the monitoring system both with and without any extension.
- The coordination necessary to meet above requirements should be as light weight as possible. Developers should be able to provide extensions without being forced to know about all other extensions and without being forced to apply for each new service at a central location.

We have decided to use the following strategy for extensions:

- Each service name and each externally visible name in the code of an extension library has a prefix that separates it from the names in all other extensions and from the ones in the basic monitor.
- A research group that wants to make extensions to the monitoring system is assigned a new prefix on request. Then this prefix is reserved exclusively for that group; thus, no further coordination is required. Developers can assign arbitrary service names, provided that they start with the assigned prefix.
- In order to realize above strategy, the table mapping service names to the functions that implement the services cannot be built statically, but must be expanded dynamically by each extension linked to the base monitor. Therefore, each extension library must contain an initialization function whose name is "register", prepended by the extension's prefix. This routine will then register all services provided by the corresponding extension. When a new prefix is requested, the basic monitoring system will be modified in such a way that it invokes this initialization routine in the startup phase.
- In addition, an empty routine with the same name will be created in a dummy library. This library is linked to the monitor as the *last* library. Thus no undefined symbols occur when some extensions are not linked to the monitor.

The procedure of requesting a new prefix has to be done only once by each group that plans to extend the monitoring system. We will try to make this procedure fully automatic, e.g. by using a WWW form or a mail server.

Services from extensions may possibly be included into the documentation of OMIS, if they are of public interest. But in order to keep the implementation modular, they will nevertheless be implemented in a separate library.

# Part III

# Services of the Monitoring Interface

# Chapter 7

# Formal Syntax of Service Requests and Replies

## 7.1 Service Requests

A service request is a string that complies with the following syntax:

```
request          ::=  [ event ':' ] action_list
action_list      ::=  seq_action_list  |  par_action_list
seq_action_list  ::=  action  |  action ';' seq_action_list
par_action_list  ::=  action  |  action ',' par_action_list
event            ::=  basic_request
action           ::=  basic_request

basic_request    ::=  request_id ' ' receiver_nodes ' '
                          service_name '(' parameters ')'
request_id       ::=  integer
receiver_nodes   ::=  nodes
nodes            ::=  '[' node_list ']'  |  '[ ]'
node_list        ::=  node  |  node ',' node_list
node             ::=  integer  |  '$' integer
service_name     ::=  identifier
parameters       ::=  parameter_list  |  empty
parameter_list   ::=  parameter  |  parameter ',' parameter_list
parameter        ::=  integer  |  floating  |  string  |  list  |  '$' integer
list             ::=  '[' parameters ']'
```

The symbols *integer*, *floating*, *string*, and *identifier* represent (32-bit) integers, (double precision) floating point numbers, quoted strings, and identifiers with a syntax comaptible to the language C. *list* denotes an (untyped) list of entities; the $-notation can be used in actions to refer to the output parameters of an event. $\$i$ refers to the $i$-th output parameter. The symbol $0 always refers to the node where an event occured.

The semantics of the input *parameter_list* depends on the concrete service and is specified in Chapters 8.1 and 8.2.

## 7.2 Service Replies

A service reply has a structure similar to that of a service request. The general syntax is:

| | | |
|---|---|---|
| *reply* | ::= | *basic_reply* \| *reply_list* |
| *basic_reply* | ::= | *request_id* ' ' *sender_nodes* ' ' *service_name* '(' *result_parameters* ')' |
| *reply_list* | ::= | *basic_reply* ';' *reply_list* |
| *sender_nodes* | ::= | *nodes* |

Again, the exact semantics of the result *parameter_list* depends on the service and is specified in Chapters 8.1 and 8.2.

# Chapter 8

# Specification of Available Basic Services

In the following subsections we will present a list of all basic services currently defined by OMIS. As you can see from the service descriptions, the goal of OMIS is to define a basis for building higher-level monitoring systems. For instance, OMIS does not include the generation of event traces, but it provides a very easy and powerful mechanism for the monitoring of events. So if you need some kind of event trace, you only have to provide the functions writing the events to disk, but you don't have to implement the event detection. Similar, OMIS services operate on the machine level, i.e. they use addresses or pointers rather than symbolic names for referring to programming objects. Thus, an OMIS compliant monitor is not forced to work with a symbol table generated during compilation of the monitored application. But, of course, it is possible to add extensions that make use of these symbol tables.

Throughout the service descriptions in Sections 8.1 and 8.2, we will use ANSI-C-like prototypes to define the input and result parameters, since this type of description is much more clear than presenting a grammar or BNF for the syntax of the request and reply strings. In addition to the types *integer*, *floating*, and *string*, we will use the type-identifier *any* which stands for any of these three types. To define more complex parameters, we will use **typedef**'s and/or C-**struct**'s[1]. Lists of values are specified by the type name followed by either a '+', denoting a list consisting of multiple (at least one) repetitions of that structure, or the name followed by a '*', denoting zero or more repetitions. The resulting string is then simply the linear layout of a value having the specified type. I.e. a **struct** corresponds to several values separated by commas, while a list (indicated by '+' or '*') corresponds to several values of its component type, that are again separated by commas, but are enclosed in brackets ('[' and ']').

A simple example will clarify this: Assume the following definition of a service:

```
typedef struct {
    string name;
    integer state;
    integer priority;
    floating cpu_time;
} Process_info;
struct {
    integer status;
    Process_info* info;
```

[1]These constructs are only used in this document to define the structure of request and reply strings, they are *not* part of the tool/monitor-interface itself.

```
}
process_info (integer+ tid_list, integer flags);
```

This says that **process_info** has two input parameters, the first one is a non-empty list of integers, the second one is a single integer. It returns an integer and a list of $4n$ elements, where elements 0,4,... are strings, elements 1,5,... and 2,6,... are integers and elements 3,7,... are floating point numbers. Therefore, a correct request for this service could be:

```
123 [1] process_info([231,345,654], 9)
```

A valid reply could look like[2]:

```
123 [1] process_info(0, ["foo",12,0,12.7, "bar",1,10,0.65,\
                     "tst",9,-10,1.1e3])
```

A few synchronous basic services return a result where some components are optional. In this case, the elements are placed in square brackets in the type definition. An input parameter of the service will then determine which components are actually present. The real **process_info** service is of this type, i.e. the definition of this service really looks more like:

```
typedef struct {
    [string name;]       // present if bit 0 is set in flags
    [integer state;]     // present if bit 1 is set in flags
    [integer priority;]  // present if bit 2 is set in flags
    [floating cpu_time;] // present if bit 3 is set in flags
} Process_info;
struct {
    integer status;
    Process_info* info;
}
process_info (integer+ tid_list, integer flags);
```

This means that all components of **Process_info** are optional. The **flags** parameter is a bit-vector that determines which of them will be present in the result. For example, the reply for the request

```
123 [1] process_info([231,345,654], 9)
```

could be (since bits 0 and 3 are set in **flags**):

```
123 [1] process_info(0, ["foo",12.7, "bar",0.65, "tst",1.1e3])
```

Asynchronous basic services have two different types of results, namely the status value that is returned as an immediate response to the service request, and the parameters that can be accessed by the actions when the defined event occurs. Therefore, we need a further notation to define the types of these results. It looks like this:

```
integer some_event(integer param)
--> struct {
   integer first_result_parameter;
   string  second_result_parameter;
   integer third_result_parameter;
}
```

[2] Note that the reply always will be a single line, without any line break. The line break in the example (indicated by the '\' at the end of the first line) is only for the sake of printing

The type of data returned immediately as a response to the service request is given in the C-like prototype (it is always **integer**, since it is only a return status). The event's output parameters, that are accessible to its actions via the $-notation are defined as a **struct** after the **-->** symbol. In the example, this means that $1 as a parameter of an action associated with **some_event** will be replaced by the value of **first_result_parameter**, $2 by the value of **second_result_parameter** and so on. Note that basic asynchronous services never have optional components in their result parameters.

## 8.1 Application Objects

Application objects are those objects in the monitored system that do not belong to the monitor itself. Currently, we only make a very coarse grain classification of these objects. We distinguish between

- processes (Section 8.1.1),
- messages (Section 8.1.2), and
- hardware (Section 8.1.3).

In Section 8.1.4 we introduce I/O objects as an example for a monitor extension.

We are currently working towards a more refined classification, that among others will also include light weight processes (threads). This will most probably also lead to an object oriented monitoring interface (see item 1 in the request for comments in Chapter 12).

### 8.1.1 Processes

#### Manipulation Services

The following services are provided to manipulate the behavior of application processes. The result of these services is a return value indicating whether or not the service could be executed correctly.

1. **start** ........................................................ start a process.

   ```
   integer start(string exec, string* argv)
   ```

   The **start** service starts a new process, e.g. a PVM task. **exec** is the executable's path name, **argv** is the vector of command line arguments.

2. **stop** ........................................................ stop a process.

   ```
   integer stop(integer* tid_list)
   ```

   This service stops all processes specified in the **tid_list** by putting them into a special state that ensures that the process no longer gets any CPU cycle. From this state, the process can only be released with the **continue** service. If **tid_list** is empty, all processes of the monitored application on the node(s) where the service request is sent to are stopped.

3. **continue** ........................................................ continue a process.

   ```
   integer continue(integer* tid_list)
   ```

This service continues the processes specified in the **tid_list** that are in a stopped state. If **tid_list** is empty, all processes of the monitored application on the node(s) where the service request is sent to are continued.

4. **kill** .................................................. send a signal to a process.

```
integer kill(integer* tid_list, integer sig)
```

Sends the signal **sig** to all processes in **tid_list**. If **tid_list** is empty, the signal will be sent to all processes of the monitored application on the node(s) the request is sent to.

5. **nice** .................................................. change priority of a process.

```
integer nice(integer* tid_list, integer val)
```

Changes the scheduling priority of the processes in **tid_list** to **val**. Again, an empty **tid_list** denotes all application processes on the node(s) the request is sent to.

6. **write_memory** ............................... write into the memory of a process.

```
integer write_memory(integer tid, integer addr, integer+ val)
```

Writes the 8-bit values in the integer list **val** into the memory cells at **addr**, **addr+1**, ... of process **tid**.

7. **write_int_registers** ......................... write into a process' integer registers.

```
integer write_int_registers(integer tid, integer reg, integer+ val)
```

Writes the values in **val** into registers **reg**, **reg+1**, ... of process **tid**. The register numbers and their bit-length depend on the node architecture; they are defined in the node processor's ABI (Application Binary Interface).

8. **write_fp_registers** .................... write into a process' floating point registers.

```
integer write_fp_registers(integer tid, integer reg, floating+ val)
```

Writes the values in **val** into registers **reg**, **reg+1**, ... of process **tid**. The register numbers and their bit-length depend on the node architecture; they are defined in the node processor's ABI (Application Binary Interface).

9. **goto** .................................................... set the PC of a process.

```
integer goto(integer tid, integer addr)
```

Sets the program counter (PC) of process **tid** to the value **addr**. This has the effect of executing a jump instruction to that address in the current context of the specified process.

10. **prepare_checkpoint** ................................ prepare a global checkpoint.

```
integer prepare_checkpoint()
```

Creates a globally consistent state for a checkpoint. This includes saving all receivable messages in the processes' address space.

11. **migrate_process** ............................... migrate a process to another host.

```
integer migrate_process(integer tid, integer node)
```

This service migrates the process **tid** to a new host defined by its node number **node**, after a checkpoint has been prepared using **prepare_checkpoint()**.

**Synchronous Services**

The following services can be used to obtain information on an application's processes:

1. **process_info** ......................................... get information on processes.

```
typedef struct {
   integer tid;                   // the global id of this process,
                                  // e.g. the PVM task id
   integer pid;                   // the local id of this process,
                                  // e.g. UNIX pid
                                  // present if bit 0 is set in flags
   [string* argv;]                // the process' argument vector,
                                  // i.e. pathname and parameters
                                  // present if bit 1 is set in flags
   [integer scheduling_state;]    // (running, sleeping, stopped, etc.)
                                  // present if bit 2 is set in flags
   [integer memory_size;]         // the process' current memory size
                                  // present if bit 3 is set in flags
   [integer priority;]            // the scheduling priority
                                  // present if bit 4 is set in flags
   [floating user_time;]          // the process' current user time
                                  // present if bit 5 is set in flags
   [floating system_time;]        // the process' current system time
                                  // present if bit 6 is set in flags

                                  // this list is not yet complete, more
                                  // information may be added, e.g. other
                                  // relevant data from the rusage structure
} Process_info;
struct {
   integer status;
   integer number_of_processes; // number of application processes on node
   Process_info* processes;
}
process_info(integer* tid_list, integer flags)
```

Detailed information about a set of processes can be obtained by the **process_info** service. **tid_list** defines the processes that have to be inspected; an empty list stands for all processes of the monitored application on the node(s) where the service request is sent to. The second parameter **flags** is a bit set that allows to mask each kind of information individually. E.g. if **flags** is equal to 10, the processes' argv vector (i.e. name and command line parameters) and their memory size will be returned. So it is possible to get all relevant process information with a single service request, but still the monitor only needs to retrieve the information that is really needed. For instance, the list of the **tids** of all application processes can be requested with **process_info([],0)**. In this case, the monitor does not need to acquire any further information.

We will also provide a mechanism that allows to get state information not only for the monitored application's processes, but also combined information for all the other processes. Most probably there will be a special **tid** representing 'all other processes'.

2. **stack_backtrace** .................. determine a process' procedure stack backtrace.

```
typedef struct {
   integer pc;
   integer fp;
} Stack_element;
struct {
   integer status;
   Stack_element* stack;
}
stack_backtrace(integer tid)
```

For debugging or performance analysis based on sampling, there is a service **stack_backtrace** that returns the process' procedure stack backtrace. It consists of a list of pairs for each active procedure invocation. Each pair contains the procedure's frame pointer and the current execution address in that procedure. The record for the most recent procedure invocation is returned as the first element of the list.

3. **read_memory** ...................................... read the memory of a process.

```
struct {
   integer status;
   integer* values;
}
read_memory(integer tid, integer addr, integer num)
```

Reads **num** bytes from process **tid**'s memory area, starting at address **addr** and returns their contents as a list of bytes.

4. **read_int_registers** .............................. read integer registers of a process.

```
struct {
   integer status;
   integer* values;
}
read_int_registers(integer tid, integer reg, integer num)
```

Reads **num** integer registers of process **tid** starting at register **reg** and returns their contents. The register numbers and their bit-length depend on the node architecture; they are defined in the node processor's ABI.

5. **read_fp_registers** ......................... read floating point registers of a process.

```
struct {
   integer status;
   floating* values;
}
read_int_registers(integer tid, integer reg, integer num)
```

Reads **num** floating point registers of process **tid** starting at register **reg** and returns their contents. The register numbers and their bit-length depend on the node architecture; they are defined in the node processor's ABI.

6. **loader_information** .................... return the loader information of a process.

```
typedef struct {
   string  path_name;       // The path name of that load module
   string  member_name;     // Member name, if module is an archive
   integer code_start;      // Start address of code segment
   integer code_len;        // Length of code segment in Bytes
   integer data_start;      // Start address of data segment
   integer data_len;        // Length of data segment in Bytes
   integer bss_start;       // Start address of BSS segment
   integer bss_len;         // Length of BSS segment in Bytes
} Loader_info;
struct {
   integer status;
   Loader_info* loader_info;
}
loader_information(integer tid)
```

On some parallel computers, e.g. the Parsytec GC/PowerPlus, programs are relocated when they are loaded by the operating system. Debuggers therefore need to know the start addresses and the lengths of the program's segments. The service **loader_information** that will return this information for each load module of the process. This service may also be used with operating systems like AIX where code (e.g. libraries) can be loaded dynamically.

### Asynchronous Services

The following services notify the caller on certain process related events that occur in the monitored application. In all of these services, an empty **tid_list** denotes "all application processes on the node(s) where the request is sent to".

1. **new_process** ..................................... A new process has been created.

```
integer new_process()
--> integer tid;    // tid of the new process
```

This event is raised whenever a new process is created. The event is raised immediately before the new process starts executing. The parameter passed to actions consist of the new process' **tid**.

2. **process_terminated** ................................... A process has terminated.

```
integer process_terminated(integer* tid_list)
--> integer tid;   // tid of terminating process
```

Event **process_terminated** is raised when a process in **tid_list** has terminated. It passes the **tid** of the terminated process to its actions.

3. **process_signal** ........................................ A process received a signal.

```
integer process_signal(integer *tid_list, integer *sig_list)
--> struct {
   integer tid;    // process receiving the signal
   integer sig;    // signal number
}
```

This event occurs whenever a process in **tid_list** receives a signal in **sig_list**. It will pass the **tid** and the signal number to its actions.

4. **process_blocked** ........................................ A process gets blocked.

```
integer process_blocked(integer* tid_list)
--> integer tid;    // tid of process that got blocked
```

This event is raised when a process in **tid_list** is blocked in a blocking communication or synchronization call. It passes the process' **tid** to its actions.

5. **process_unblocked** .................................... A process gets unblocked.

```
integer process_unblocked(integer* tid_list)
--> integer tid;    // tid of process that got unblocked
```

This event is raised when the blocking condition (due to a communication or synchronization call) of a process in **tid_list** is removed again. It passes the process' **tid** to its actions.

6. **process_stopped** ........... A process has been stopped by the monitoring system.

```
integer process_stopped(integer* tid_list)
--> integer tid;     // tid of process that has been stopped
```

This event is raised when a process in **tid_list** is stopped by the monitoring system (using the **stop** service). It passes the process' **tid** to its actions.

7. **process_continued** ........ A process has been continued by the monitoring system.

```
integer process_continued(integer* tid_list)
--> integer  tid;    // tid of process that has been continued
```

This event is raised when a process in **tid_list** is continued by the monitoring system (using the **continue** service). It passes the process' **tid** to its actions.

8. **scheduling** ............................................... Process scheduling event.

```
integer scheduling()
--> struct {
   integer old_tid;  // tid of process that has been descheduled
   integer new_tid;  // tid of process that is going to be scheduled
}
```

On machines where process scheduling is observable, the event **scheduling** will be raised each time process scheduling takes place. The parameters available for the actions contain the the **tids** of the descheduled and the scheduled processes. The **tid** is -1, if the process doesn't belong to the monitored application.

9. **breakpoint** ................................ A process reaches a given code address.

```
integer breakpoint(integer* tid_list, integer address)
--> integer       // tid of process that reached the breakpoint
}
```

Breakpoints for debugging purposes also are asynchronous services, since they can be reached arbitrarily often. The **breakpoint** event will occur each time a process in **tid_list** reaches the specified **address**. The result parameter for actions is the **tid** of the process.

10. **step** ......................................... Single step a process (step into calls).

```
integer step(integer tid)
-->               // no result parameters
```

This service resumes the stopped process **tid** for the execution of one machine statement. It will invoke its actions after the single step has finished. This is an asynchronous service, since the process may block during the single step, so there is no guaranteed response time.

11. **next** ....................................... Single step a process (step over calls).

```
integer next(integer tid)
-->               // no result parameters
```

This service resumes the stopped process **tid** for the execution of one machine statement, treating subroutine calls as a single statement. It will invoke its actions after the single step has finished. This is an asynchronous service, since the process may block during the single step, so there is no guaranteed response time.

12. **finish** ..................................................... Finish current procedure.

```
integer finish(integer tid)
-->               // no result parameters
```

This service continues process **tid** until it leaves the currently active procedure. Again, this is an asynchronous service, since the process may block.

13. **call_function** ......................................... call a function in a process.

```
integer call_function(integer tid, integer addr, any* params)
```

This service will call the function at address **addr** with parameters **params** in the current context of process **tid**. The result will be the same as if the process had called the function at the point where it was stopped when this service has been invoked. This means, the parameters will be written to the proper locations, the return addess will be saved, the process' PC will be set to the specified address, and the process will be continued until the function returns. Again, since the function may block, this is an asynchronous service.

14. **ready_to_migrate** ............................. A process is going to be migrated.

```
integer ready_to_migrate(integer* tid_list)
--> struct {
   integer tid;        // tid of process that will be migrated
   integer dest_node;  // destination node of migration
}
```

This event will be raised when the system is going to migrate a process in **tid_list**. The result parameters passed to actions include the process' **tid** and its destination node.

15. **migration_finished** .................. A process has been migrated to another node.

```
integer migration_finished(integer* tid_list)
--> integer;           // tid of migrated process
```

This event is raised on a node, when a process in **tid_list** has been migrated to that node. The information available for actions if the process' **tid**.

16. **start_lib_call** .................. A process invokes a call to the programming library.
**end_lib_call** .............. A process returns from a call to the programming library.

```
integer start_lib_call(integer* tid_list, string lib_call_name)
--> struct {
   integer tid;   // process doing the library call
   ...            // the following results are the input parameters of
                  // the library call
}

integer end_lib_call(integer* tid_list, string lib_call_name)
--> struct {
   integer tid;   // process doing the library call
   ...            // the following results are the result parameters of
                  // the library call
}
```

These events are raised whenever a process in **tid_list** is calling the specified routine of the parallel programming library (e.g. PVM). **start_lib_call** is raised just after the routine is entered, while **end_lib_call** occurs just before the call returnes. The services are provided for all routines in the programming library; the value of the **lib_call_name** parameter specifies the routine's name. In addition to the **tid** of the calling process, the values of the input or output parameters of the library call are passed to the actions. The number and the types of these parameters depend both on the programming library used and the selected library call. They will be defined later in an extra part of this document.

The reason for this two-step approach is to avoid dependencies between the on-line monitoring interface specification and the supported programming library. It clearly separates the real task of these services, namely to detect calls to the programming library, from library specific aspects. Moreover, we will work towards an automatic generation of these services for a specific programming library from a specification of the library's function prototypes (see known problem no. 4 in Chapter 13).

### 8.1.2 Messages

#### Manipulation Services

The services in this group will allow to modify message queues and single messages. They return a status value indicating whether or not the service could be executed correctly.

1. **insert_message** .................... Insert a message into a process' message queue.

   ```
   integer insert_message(integer tid, integer* msg, integer pos)
   ```

   For debugging message passing errors, it may be profitable to have a service **insert_message** that allows to insert a message at a given position **pos** in a process **tid**'s message queue. The message is specified as a list of bytes.

2. **remove_message** ................ Remove a message from a process' message queue.

   ```
   integer remove_message(integer tid, integer pos)
   ```

   This service will delete the message at position **pos** in process **tid**'s message queue. It may be most useful for debugging message passing errors.

3. **tag_message** ............................................... Add a tag to a message

   ```
   integer tag_message(integer tid, integer msg_ptr, integer tag)
   ```

   To support debugging of message passing programs, OMIS defines services that operate on a special message tag. This tag is independent of any message tag defined by the programming model. In facs, it is invisible for both the programming library and the application. However, there are monitoring services to set this tag, to read its value and to trigger some actions when a message with a given monitor tag is received by a process.

   These tags may be used for different purposes. For example, during debugging, the user may be interested in how a message that is sent by a process is processed by another one. By tagging the message, the monitor can stop the receiving process, when it receives that message, regardless of the number of preceding messages in the receiver's message queue. When the receiver is stopped, the user can examine how the message is processed, e.g. by single stepping. In addition, tags can also be used to implement distributed event detection.

   The **tag_message** service must be called immediately before process **tid** issues a message send operation. It will then put the specified **tag** into the message being sent. **msg_ptr** is the pointer to the message buffer.

#### Synchronous Services

Currently, this group contains only two services:

1. **queue_info** ........................ Return information on a process' message queue.

   ```
   typedef struct {
      [integer  sender;]         // tid of process that sent this message
                                 // present if bit 0 is set in flags
      [integer  msgtag;]         // the PVM message tag
                                 // present if bit 1 is set in flags
   ```

```
    [integer  size;]             // size of the message
                                 // present if bit 2 is set in flags
    [integer* contents;]         // the message buffer as a list of bytes
                                 // present if bit 3 is set in flags
    [integer tagged_by_monitor;] // is the message tagged by the monitoring
                                 // system
                                 // present if bit 4 is set in flags
    [integer  monitor_tag;]      // the tag added by the monitoring system
                                 // present if bit 4 is set in flags
} Message_info;
struct {
    integer status;
    integer queue_length;        // number of messages in queue
    Message_info* messages;
}
queue_info(integer tid, integer flags)
```

This service will return information on the message queue of process **tid**. In analogy to **process_info** we provide a bit set mechanism allowing to select the kind of information to be retrieved. A component in **Message_info** will only be returned, if the corresponding bit in **flags** is set.

2. **message_buffer** ............... Get the address of a process' current message buffer.

```
struct {
    integer status;
    integer buffer_address;      // address of message buffer
    integer buffer_size;         // length of message in buffer
}
message_buffer(integer tid)
```

The service **message_buffer** will return the address of a process' current message buffer and its length. It will mainly be used by debugging tools.

### Asynchronous Services

The following events are provided specifically for the monitoring of message passing. Other events, such as start and end of send or receive calls, can be monitored using the **start_lib_call** and **end_lib_call** services (see Section 8.1.1).

1. **message_arrived** ...................... A message is inserted into a message queue.

```
integer message_arrived(integer *tid_list)
--> struct {
    integer  tid;               // process where message is inserted
                                // into queue
    integer  sender;            // tid of process that sent this message
    integer  msgtag;            // the PVM message tag
    integer  tagged_by_monitor; // is the message tagged by the monitoring
                                // system
    integer  monitor_tag;       // the tag added by the monitoring system
}
```

We will provide a service **message_arrived** that reports when a message is inserted into the message queue of a process in **tid_list**. This event is essential if you want to build a tool based on event traces that needs information on the size or contents of the message queue.

2. **tagged_message_recv** ........................ A process receives a tagged message.

```
integer tagged_message_recv{integer* tid_list, integer monitor_tag}
--> struct {
   integer  tid;             // process receiving the message
   integer  sender;          // tid of process that sent the message
   integer  msgtag;          // the PVM message tag
}
```

This service reports the receipt of a message by a process in **tid_list**, iff the message contains the given **monitor_tag**.

### 8.1.3 Hardware

#### Manipulation Services

Since the hardware usually cannot be manipulated, there are no services in this category.

#### Synchronous Services

These services return static and dynamic information on an application's virtual machine, i.e. the part of a parallel computer or a distributed computing environment currently used by the monitored application:

1. **number_of_nodes** .................. Determine number of nodes in virtual machine.

```
struct {
   integer status;
   integer num_nodes;
}
number_of_nodes()
```

   This service returns the number of processing nodes currently in the virtual machine, regardless to which node the service request is sent.

2. **list_hosts** ............................. Return host names and their node numbers.

```
typedef struct {
   integer node_number;  // a host's node number
   string  host_name;    // the host's machine name
} Host_info;
struct {
   integer status;
   Host_info* hosts;     // list of hosts
}
list_hosts()
```

   This service returns a list of the names of all nodes in the virtual machine together with their assigned node numbers, regardless to which node the service request is sent.

3. **node_info** ................................ Return hardware information on a node.

```
typedef struct {
  [string  architecture;]    // architecture type (as in pvmgetarch)
                             // present if bit 0 is set in flags
  [integer number_of_cpus;]  // number of available CPUs
                             // present if bit 1 is set in flags
  [integer used_cpus;]       // number of CPUs used by the application
                             // present if bit 2 is set in flags
  [integer avail_memory;]    // available amount of main memory
                             // present if bit 3 is set in flags
  [integer used_memory;]     // main memory used by application
                             // present if bit 4 is set in flags
  [integer avail_disk;]      // available disk space
                             // present if bit 5 is set in flags
```

```
    [integer cpu_speed;]        // relative speed of CPU
                                // present if bit 5 is set in flags
    ...                         // more information not yet specified
  } Node_info;
  struct {
     integer status;
     Node_info info;
  }
  node_info(integer flags)
```

Detailed information on nodes is provided by the **node_info** service. As with the other information services, the bit-vector **flags** defines which kind of information has to be retrieved. In addition to the data specified above, we currently think towards providing detailed information on the communication network:

- number of links to other nodes and their estimated or measured speed,
- link usage,
- type of network, e.g. ATM, Ethernet, etc.,
- network topology, e.g. bus, 2D-grid, etc.

### Asynchronous Services

Since the number of nodes in an application's virtual machine may be dynamic (as in PVM), the monitoring system will adapt itself to such changes. This means that if a new node is added to the virtual machine, a monitor will be started on that node just before the application starts to use that node. Likewise, if a node is removed, the monitor on that node will terminate. The monitoring interface provides two events giving notice of these situations:

1. **new_host()** .................... A new host has been added to the virtual machine.

   ```
   integer new_host()
   --> integer node_number;      // node number assigned to the new host
   ```

   The service **new_host** will report that a new host has been added to the virtual machine. The event will be raised on any node where it is defined, just after the monitor on the new node has been initialized. It returns the number of the new node, so the event's actions can be redirected to the new node by specifying $1 in their node list.

2. **host_removed** ........................ A host is removed from the virtual machine.

   ```
   integer host_removed()
   --> integer node_number;      // number of the node that will be removed
   ```

   This service will give notice that a host is removed from the virtual machine. Like **new_host**, this event will be raised on any node where it has been defined. It is raised just before the monitor on the node to be removed terminates, so actions can still be executed on that node.

### 8.1.4 Enhancement: Parallel I/O

In contrast to message passing interfaces, which are rather standardized now, parallel I/O is still an active research issue. It is not only difficult to implement parallel I/O efficiently, but it is still not decided which features a parallel I/O system must have, and how the programming interface must look like. Therefore, it is not possible at the moment to specify a standard for the corresponding monitoring services. This is one reason why the monitoring of parallel I/O is defined as an extension to OMIS.

The other reason, which is related to the one above, is of more practical nature: Since there is no agreement on the functionality of parallel I/O systems, there are different I/O libraries that can be used with various programming libraries. Examples are PIOUS and PFSLib. Defining the monitoring services for parallel I/O as an extension to OMIS leaves everyone free to provide the services most appropriate for a specific I/O library.

In the following sections we will present some basic considerations for these services.

#### Manipulation Services

It is not yet clear, which manipulation services could be needed by different kinds of tools, if they are needed at all. Similar to processes, a service for migrating parts of a file from one disk to another could be useful for load balancing strategies.

#### Synchronous Services

These services could provide additional information about files, that are not available via the programming interface of the parallel I/O system. Examples of this kind of information include:

- The way, how a given file is distributed across different disks.
- The amount of disk space used by a file on the different disks.

Typically, a parallel I/O system is used because it hides the distribution from the programmer. However, if you want to do performance analysis (of either the application or the I/O system), you will be interested in this kind of information.

#### Asynchronous Services

There are two levels of events related to I/O that should be observable using the monitoring system:

1. For the higher level, there will be extensions to the basic services **start_lib_call** and **end_lib_call** that report the start and end of each I/O related library call. These extensions support debugging, visualization and performance analysis of an application with respect to I/O.

2. However, we may also want to use the monitoring system for performance analysis of the I/O system itself, or for an automatic performance improvement (e.g. some kind of load balancing for files). For this purpose, there should also be services that report lower level events, e.g. access to local and remote disks during the execution of an I/O request for a distributed file.

## 8.2 Monitor Objects

Currently, we have defined two types of objects in the monitoring system: user defined events and asynchronous service requests. The latter are monitor objects, since they have to be stored in the monitoring system, and they can be manipulated by other services. Other monitor objects may be added by distributed tool extensions, e.g. timers, counters, etc.

In addition, there are some services that cannot be associated with a specific monitor object. They are introduced in Section 8.2.3.

### 8.2.1 Asynchronous Service Requests

#### Manipulation Services

1. **enable** .......................................... Enable a previously defined event.
   **disable** .......................................... Disable a previously defined event.

   ```
   integer enable(integer request_id)
   integer disable(integer request_id)
   ```

   These services will enable or disable the monitoring of an event previously defined with the specified request identifier **request_id** in an asynchronous service request. Since events always are initially disabled, they must enabled explicitly. The services can also be used for temporarily disabling breakpoints or performance measurements. Since they can be used as actions, it is possible to start and stop monitoring of an event based on the occurrence of another event. In this way, the detection of complex distributed events is possible.

2. **delete** ........................ Delete a previous request for an asynchronous service.

   ```
   integer delete(integer request_id)
   ```

   This service deletes the event-action definition with the specified request identifier **request_id**.

### 8.2.2 User Defined Events

#### Manipulation Services

1. **define_user_event** .................................... Create a user defined event.

   ```
   integer define_user_event(integer event_no)
   ```

   The monitoring system will allow to use user defined events via this service. The parameter **event_no** specifies the number which is used to address this event in the other services.

2. **raise_event** ........................................... Raise a user defined event.

   ```
   integer raise_event(integer event_no, any* params)
   ```

   This service will raise the user defined event **event_no** that must have been created previously using the **define_user_event** service. **params** is a list of parameters that will be passed to the event's actions. The size of this list, i.e. the number of parameters is arbitrary.

3. **destroy_user_event** ................................ Destroy a user defined event.

```
integer destroy_user_event(integer event_no)
```

This service is used to destroy a used defined event **event_no** when it is no longer needed. In addition, all asynchronous services using this event will be deleted.

### Asynchronous Services

1. **user_event** ................................ A user defined event has been raised.

```
integer user_event(integer event_no)
--> ...        // parameters specified in the call to raise_event
```

The service **user_event** gives notice that the specified user event has been raised using **raise_event** service. It passes all parameters specified in the invocation of **raise_event** to its actions. Thus, an action can refer to the first element in the **params** list of **raise_event** via $1, to the second one via $2, and so on.

User defined events can for instance be used to realize action lists that can be triggered by different events, without having to define the action list twice. If you want to trigger a list **A(x)** of actions with event **E1** or event **E2**, you can specify[3]:

```
define_user_event(5)
E1: raise_event(5, [$1])
E2: raise_event(5, [$1])
user_event(5): A($1)
```

In addition, user defined events allow to chain action call-backs. For instance, you could provide some call-back **filter(var, val, event_no)** that raises a user defined event, iff **var == val**. If we assume that the second parameter returned by the **start_lib_call(tid_list, "pvm_send")** service is the destination **tid**, then in the following situation

```
define_user_event(6)
start_lib_call([12], "pvm_send"): filter($2, 15, 6)
user_event(6): A()
```

action **A()** will be executed, iff process 12 sends to process 15.

Finally, user defined events can of course also be used for additional code instrumentation.

## 8.2.3 Miscellaneous

### Synchronous Services

1. **print** ........................................................ Return arguments.

```
struct {
   integer status;
   any* args;
}
print(any* args)
```

[3]For clearness reasons, we don't show the request identifiers and the receiver node lists in this example

Sometimes a tool wants to be directly notified about an event and its parameters. For this purpose, the service **print** is available. It simply returns its arguments in its result structure.

2. **extensions** .................................... Return a list of available extensions.

```
struct {
   integer status;
   string* extension;   // the prefix used for this extension
}
extensions()
```

This service allows to ask which monitor extensions or distributed tool extensions are available in a monitor. It will return the list of the prefixes used for the services of the available extensions. Thus, tools can decide whether the necessary extensions are available, and they may handle the cases where less important extensions are missing.

# Part IV

# Concepts for an Implementation

# Chapter 9

# The Software Module Structure

This chapter is intended for readers who are interested in details on an implementation of an OMIS compliant monitoring system which is integrated with an already existing parallel programming environment. People interested exclusively in the specification part of the document or in just using OMIS for connecting tools to it may skip this chapter.

We will show a potential working together of all components in an environment for workstation clusters where we use PVM as parallel programming library and PFSLib as parallel I/O library. In contrast to the logical module structure presented in Chapter 4 we find things to be more complex, as the monitoring system turns out to be at least partially integrated with many other modules.

Let us have a closer look at these modules and the way they cooperate. PVM at the software module level consists of one daemon per node (pvmd) and a PVM library being linked to each task of the application. The same holds for PFSLib and may be true for many other runtime support components. The situation is even worse for the monitoring system itself: besides having stand-alone components as (at least) one main monitor process per node we have to integrate monitoring components in form of libraries into all other software components from which we want to have information which can not directly be accessed. Integrating parts of the monitor with other software is called instrumentation. We need an instrumentation of the library codes and of all daemons involved in the running system[1]. With the operating system we have to choose other means, as instrumentation without vendor support is not feasible here. The integration of a monitoring system into an existing HW/SW-system is a complex task. Therefore, any concept for a monitoring system must also discuss the consequences for all other components of the system.

Figure 9.1 shows the software module structure for our approach. On the top we can see the monitor process. It is a monolithic block being linked together with the libraries of monitor extensions, distributed tool extensions, and distributed tool components. The reason for having at least one process for the monitor lies in the necessity for the monitor to react autonomously. It must accept commands and invoke actions concurrently to other activities in the system. The visible part of it serves for communication with other modules; thus, all information flow goes through a communication library where calls and answers are transferred to message exchange[2]).

Again we identify three data and control flow paths from the monitor/program-interface to other components. However, the meaning of these lines is slightly different. With task and daemon processes this interface has been moved directly into the components. The thick solid line between them and the monitor process denotes monitor internal communication between this process and some monitor libraries. The real monitor/program-interface in these two cases

[1]It would be possible to access all library related information via the operating system. However, this path is much too inefficient with respect to overhead evoked. Thus, special library instrumentation is inevitable.

[2]See Known Problems No. 2.

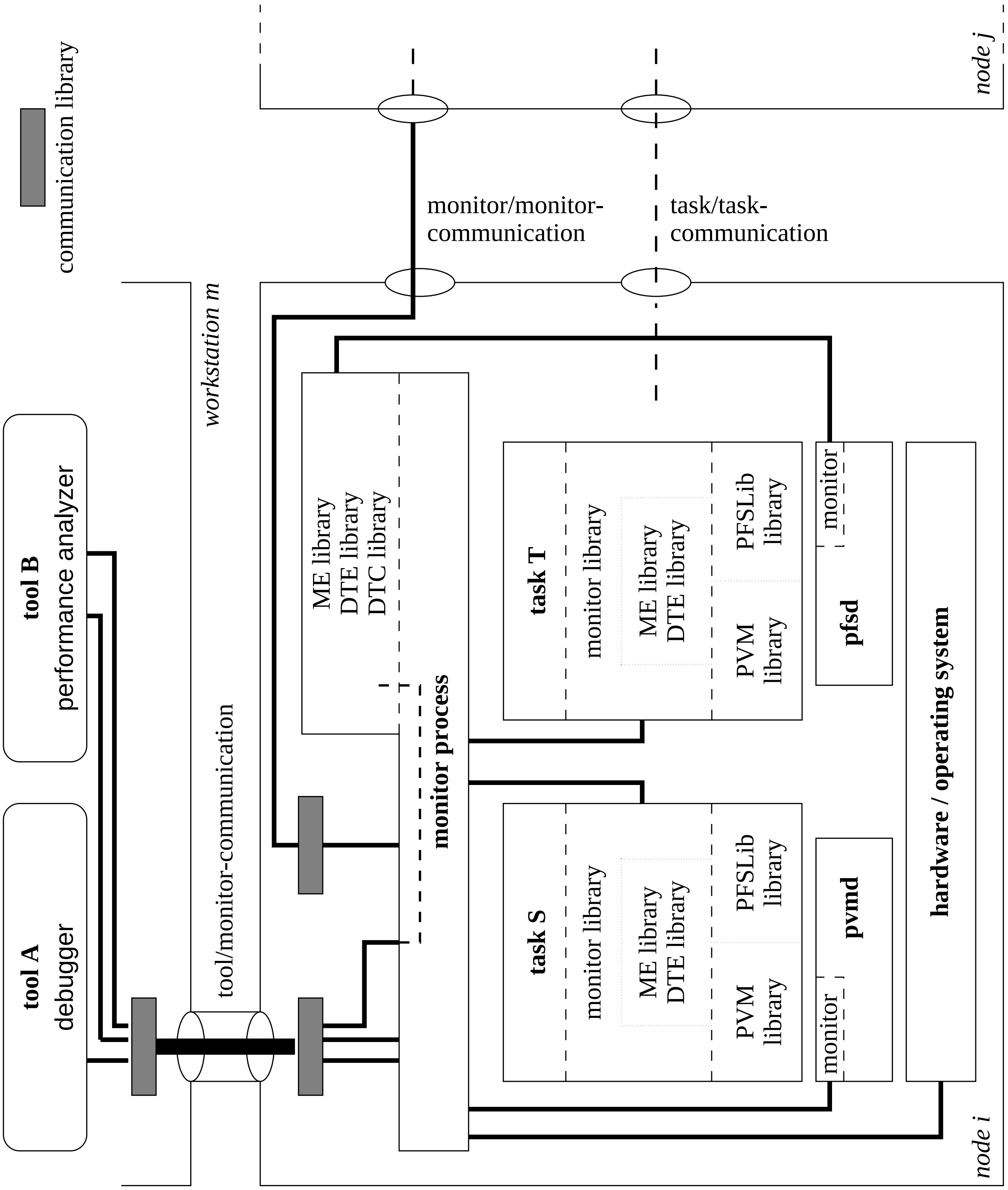


Figure 9.1: Software module structure of an environment with tools and PVM as a basic programming library layer and PFSLib a a parallel I/O library

is the interface between the monitor libraries and the rest of the task or daemon process. The situation is slightly different for the third case, the cooperation with the operating system. Here we can only use what is directly provided. In more detail this means that we will use system-calls, access to special file-systems (`/proc`) or special devices (`/dev/kmem`) to collect necessary information. No instrumentation of the operating system will be done. Although this would be a nice feature, e.g. to supervise scheduling activities, it would nevertheless ruin the portability of the approach.

Let us come back to the connection between the monitor process and the PVM daemon. Two aspects are important. First, the monitoring system will take profit of the already provided tasker/hoster interface of PVM. As these interfaces allow only one third party software product to connect to the daemon, it is inevitable to mirror this functionality at the tool/monitor-interface. Whenever it will be used by some extensions of PVM their activities can now also be monitored by tools.

A second source of information is a monitor library integrated into the PVM daemon. Its purpose is to yield data about the daemon's internal state, such as state of message queues and current message transfer, process group informations, etc. Thus, this monitor library will mainly be realized by access functions which can read the appropriate informations in the daemon's address space. The design of this interface is of great importance not only for the OMIS project but also for other tool developers as it will give them easy access to PVM daemon runtime informations. This is useful for designing their own tools, not necessarily on-line tools, e.g. management facilities for traces or batch jobs.

The final components to be discussed are the processes which represent the tasks of the application program. These tasks include as regular components their own code as well as all necessary libraries for execution: the PVM library and as an example of an extension also the PFSLib library for parallel file access. In addition we need a monitor component in order to survey and manipulate the task's execution. All activities could also be handled directly in the monitor process but the splitting into several components increases efficiency. Manipulations can be done directly during task code execution. In several cases no context switch and no communication appear when the monitor has to react to a certain condition (e.g. increasing some event counter). This block includes functionality which can also be found in the monitor process: a monitor library and occasionally a monitor extension library and a DTE library. All these entities, from which only the monitor library is imperative, serve for providing the monitor process with the necessary details. The libraries will contain routines that wrap around PVM (or PFSLib etc.) functions, in order to get events upon entry and exit of these functions and in order to invoke task manipulations. They may also contain routines implementing actions that should be called in the context of the task raising an event. For instance, if we want to measure the mean time spent in a PVM function, it is prohibitive to make a UNIX context switch just for reading the clock and updating an integrating timer. This has to be done in the application's context. The data exchange between these parts of the monitor and the monitor process will be implemented via a shared memory block, in order to be efficient. If asynchronous notification is necessary, UNIX signals will be used.

# Chapter 10

# The Monitor/Program-Interface

The monitor/program-interface will be divided into three different parts: one collecting information from the program itself, another-one being in contact with the PVM daemon, and finally an access path to operating system and hardware related informations.

The functionality of this three-partite interface is not yet specified. However, it is clear that the lowest level information and manipulation routines will take profit of the ptrace and system-call interface to UNIX. The design of the interface to the PVM daemon is of crucial importance but will be deferred until the functionality of the tool/monitor-interface is fixed. Finally, the interface part to the monitor library which gets linked to the application tasks has to be defined. As this interface does not interfere with other people's software products we will make a specification only when making concrete plans for an implementation of the monitoring system.

# Chapter 11

# Time Schedule

### OMIS Version 1.0: February 1, 1996

The first version of OMIS serves as a basis for the design of an OMIS compliant monitoring system and for adaption of OMIS based tools.

### Start of design phase: January, 1996

A group of six researchers and students at TUM is currently designing an OMIS compliant monitoring system based on PVM as programming paradigm and networks of workstations as target architecture. Implementation will start mid of 1996.

# Part V

# Diverse

# Chapter 12

# Requests for Comments

In this chapter we summarize important open question of the OMIS project. We would like to encourage any reader of this document to send us his/her comments, ideas, suggestions etc. Please refer to open questions by indicating their number.

**1** Should the tool/monitor-interface be designed in an object oriented manner? This could have several advantages:

- By defining a hierarchy of observable objects, the interface is structured much more clearly. The hierarchy also defines an abstract model for the kind of system observable by the monitoring system. A first idea of such an object model together with some of the available methods (services) is shown in Fig. 12.1.
- The on-line monitoring interface specification could be defined in terms of abstract object classes. The actual object classes needed for a specific parallel programming library would then be derived from these ones and inherit their methods (i.e. services). For instance, we could have abstract processes with services like **stop**, **continue**, **user_time**, **read_memory**, and so on. A UNIX process, derived from this abstract process class, would inherit these services and could define additional ones, e.g. **kill** or **nice**. A PVM task could be derived from a UNIX process and provide additional services such as **message_buffer**.

  This scheme would make it possible to separate OMIS into a large part that is independent of any concrete programming library and a small extension for each such library (e.g. PVM, MPI). It would also make tools more portable, since for instance a debugger could start and stop processes no matter whether they are PVM tasks or MPI processes.
- Finally, using object oriented techniques for the monitors' implementation could increase portability with respect to different programming libraries. If a significant part of the code for a concrete object could be inherited from the abstract base classes, support for different programming libraries could be implemented in relatively small monitor extensions.

**2** Should we consider to provide OMIS also for shared memory architecture environments? It might be not very difficult to do so.

**3** Should we have a more powerful request language? At the moment, we only provide the combination of events and actions, and concurrent and serial execution of action lists. A lot of other constructs could be useful, e.g.:

- a mix of parallel and serial execution within one action list,

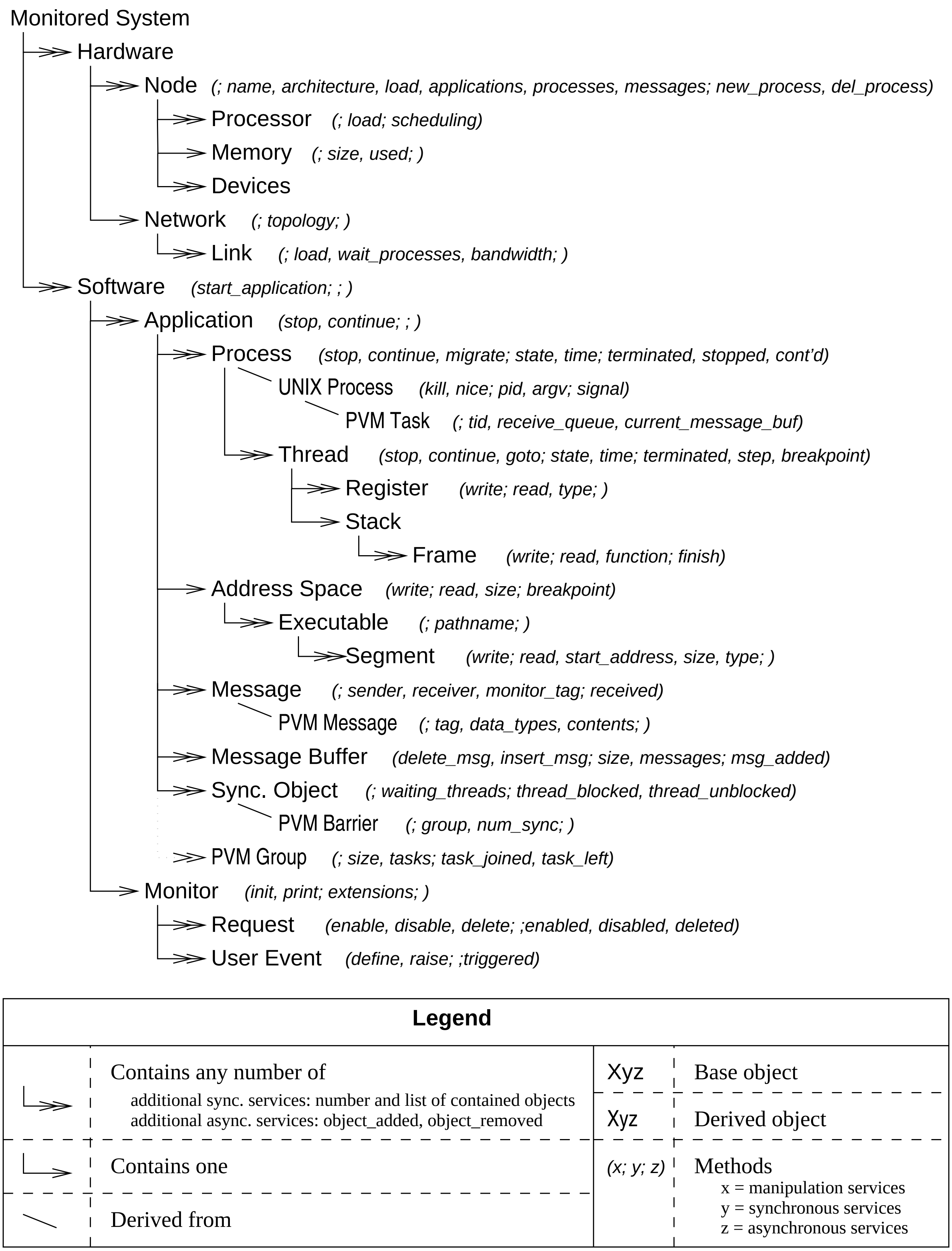


Figure 12.1: A possible object scheme for an on-line monitor. The inheritance relation between the base objects is not shown.

- definition of asynchronous services as an action, i.e. when an event occurs, define a new event-action pair, where the definition may use output parameters of the first event.
- parameter passing between actions.

However, each of these features adds significant complexity to the request parser, thereby increasing the monitor's intrusiveness. Thus, we have to find a suitable compromise.

# Chapter 13

# Known Problems

In this chapter we summarize important known problems of the OMIS project, which mostly concern about the implementation of an OMIS compliant monitoring system. We would like to encourage any reader of this document to send us his/her comments, ideas, suggestions etc. Please refer to known problems by indicating their number.

**1** The monitor must be able to work together with several application either of one or several users (e.g. load balancing). It is not yet clear how to design this.

**2** Both the communication between a tool and the monitor and the monitor/monitor communication could be based on either the communication mechanism provided by the parallel programming library (e.g. PVM), TCP sockets, or other mechanisms like RPCs. Using PVM communication may raise several complications. E.g., we have to separate monitor/monitor communication from task/task communication (PVM doesn't have the communicator concept of MPI). Another problem is that the tool must be a PVM task, but must not be included in the monitored set of tasks.

On the other hand, socket communication is much more difficult to use, and it is system dependent. It is also not yet known whether we can use both PVM and socket communication within one process.

**3** For efficiency reasons, some actions (e.g. updating a timer that measures the time a task spends in a receive call) must be executed within the context of the observed task. If we introduced a UNIX process switch here, the overhead would make useless the whole measurement. But, some actions can not be executed in the task that produces an event, e.g. most actions related to debugging that are based on the ptrace() system call.

In summary, there are four possible combinations:

- An event can be detected either in an application task or in the monitor process.
- An action can be executed either in the application task or in the monitor process.

It is not fully clear yet how to support all combinations with a uniform and orthogonal interface.

One solution would be to have additional flags for the registration of a new service. One of these flags must specify whether the service is an event (asynchronous basic service) or an action (synchronous or manipulation basic service). For an event, another flag could indicate whether it is detected in the monitor process or in the application process. For an action, the flags could indicate whether it has to be executed in the monitor process or in the application process, or can be executed in both of them. The base monitor could then automatically pass the event to another process, if necessary.

**4** The **start_lib_call** and **end_lib_call** services should be generic, in order to keep the amount of PVM dependent functions in OMIS acceptable. We plan to have a description file containing ANSI-C prototypes of all observable library calls. This file must also define which parameters are input and which are output parameters. In addition, it has to specify, how the values of these parameters are translated into the data types available in OMIS. This file could look like:

```
int pvm_recv(int tid, int msgtag)  // C prototype
start {                            // parameters passed to actions of
   integer tid;                    // start_lib_call
   integer msgtag;
}
end {                              // parameters passed to actions of
   integer pvm_recv;               // end_lib_call (result of function)
}

int pvm_...
```

This file could also contain routines for translation of parameters, or for providing extra information on implicit parameters of the library calls (e.g. the message buffer in **pvm_send**).

We intend to have some kind of translator that automatically generates the wrapping functions containing the instrumentation, and inserts them into the PVM library. In addition, the translator could also generate information for the monitoring process that could allow to monitor these library calls using traps, e.g. if an instrumented library is not available when debugging an application.

It is not yet known, whether these goals can be achieved and how this generic scheme can be implemented.

**5** In order to make an OMIS compliant monitor compatible with other tools using the hoster and tasker interfaces of PVM (e.g. resource managers), we have to mirror these interfaces. It is not yet known, how this can be achieved. A possible way would be to intercept the calls to **pvm_reg_hoster** and **pvm_reg_tasker**. In this way, the monitor knows to which tasks it must forward the information on new hosts or tasks.

# Glossary

Throughout the text of the specification we will use the following technical terms.

**Asynchronous Service** A service that may have have any number of responses, occuring at unpredictible points in time.

**Basic Service** A service of an OMIS compliant monitoring system that is defined in Part III of this specification or by a monitor extension. Basic services are the building blocks from which service requests are constructed.

**Composite Service** A service of an OMIS compliant monitoring system that is composed from several basic services. As an example, "detect a send event in process x" and "stop all processes" are basic services, while "stop all processes, when a send event is detected in process x" is a composite service.

**Distributed tool component (DTC)** Component of a distributed automatic tool which is located on one node. These tools have no user interface.

**Distributed tool extension (DTE)** Part of a centralized interactive tool which is replicated and runs on every node involved in direct cooperation with the monitor on that node. Used for preprocessing of data or organization of special distributed functionalities.

**Manipulation Service** A service that only returns an acknowledge or an error message. Its effect is to manipulate objects in the application or the monitoring system.

**Monitor** We call a (single) monitor that part of the monitoring system which is located on a single node of the parallel system (either a workstation or a node of a parallel machine). It may be composed of processes and libraries linked to various other software modules. Interaction between these parts may use all available mechanisms.

**Monitor Extension (ME)** An extension to a monitoring system offering new services for the monitoring of application objects not yet considered in that monitoring system.

**Monitoring System** A monitoring system is the collection of all software parts in a distributed system which observe and modify the running program and the underlying operating system and hardware architecture and communicate with one or more tools.

**Monitor/monitor-communication** Interaction between monitors on separate nodes of the system.

**Monitor/program-interface** This is the interface of a monitor with the object it should observe. For simplicity reasons this is called monitor/program-interface. However, the monitor interacts not only with the program but with all which gets the program running, i.e. the parallel programming library (e.g. PVM), the operating system, and the hardware. Access to other components like the PFSLib is controlled via monitor extensions.

**OMIS** The **O**nline **M**onitoring **I**nterface **S**pecification.

**Service** The functions offered by an OMIS compliant monitoring system, i.e. the commands that are accepted by its tool/monitor-interface.

**Synchronous Service** A service that returns exactly one result within a (at least theoretically) predictible response time.

**Tool/monitor-interface** The tool/monitor-interface is responsible for interaction between tools and monitors. This interface is the main subject of the on-line monitoring interface specification.

# History

### Version 1.0: February 1, 1996

Current version. Published as a technical report at TUM [LWSB96].

### Pre-Version 0.9 beta: November 30, 1995

Second draft version which served as a dicussion basis for a birds-of-a-feather meeting at the Supercomputing'95 conference in San Diego, California, USA, December 1995. The session was moderated by Arndt Bode and Vaidy Sunderam.

### Pre-Version 0.9 alpha: August 31, 1995

First draft version which served as a discussion basis for a birds-of-a-feather meeting at the European PVM Users' Group Meeting in Lyon, September 1995.

### Kick-off meeting: July, 1995

Initiators: Arndt Bode, Thomas Ludwig, Vaidy Sunderam, Roland Wismüller

# Bibliography


[BB91] T. Bemmerl and A. Bode. An Integrated Environment for Programming Distributed Memory Multiprocessors. In A. Bode, editor, *Distributed Memory Computing - 2nd European Conference, EDMCC2*, volume 487 of *LNCS*, pages 130–142, München, April 1991. Springer-Verlag.

[BB92] Arndt Bode and Peter Braun. Monitoring and Visualization in TOPSYS. In G. Kotsis and G. Haring, editors, *Proc. of Workshop on Monitoring and Visualization of Parallel Processing Systems*, pages 97 – 118, Moravany nad Váhom, CSFR, 1992. Elsevier, Amsterdam (1993).

[BBB+90] H.J. Beier, T. Bemmerl, A. Bode, et al. TOPSYS - Tools for Parallel Systems. Research report SFB 342/9/90 A, Technische Universität München, January 1990.

[BHL90] T. Bemmerl, O. Hansen, and T. Ludwig. PATOP for Performance Tuning of Parallel Programs. In H. Burkhart, editor, *Proceedings of the CONPAR 90 - VAPP IV Joint International Conference on Vector and Parallel Processing, Zurich, Switzerland*, pages 840-851, Berlin, September 1990. Springer. Lecture Notes in Computer Science, 457.

[BL90] T. Bemmerl and T. Ludwig. MMK — A Distributed Operating System Kernel with Integrated Dynamic Loadbalancing. In H. Burkhart, editor, *Proceedings of the CONPAR 90 - VAPP IV Joint International Conference on Vector and Parallel Processing, Zurich, Switzerland*, pages 744-755, Berlin, September 1990. Springer. Lecture Notes in Computer Science, 457.

[BLT90] T. Bemmerl, R. Lindhof, and T. Treml. The Distributed Monitor System of TOPSYS. In H. Burkhart, editor, *Proceedings of the CONPAR 90 – VAPP IV Joint International Con ference on Vector and Parallel Processing, Zurich, Switzerland*, volume 457 of *Lecture Notes in Computer Science*, pages 756-765, Berlin, September 1990. Springer. Lecture Notes in Computer Science, 457.

[BW95] T. Bemmerl and R. Wismüller. On-line Distributed Debugging on Scalable Multiprocessor Architectures. *Future Generation Computer Systems*, (11):375-385, November 1995. `http://wwwbode.informatik.tu-muenchen.de/~wismuell/publications/fgcs95.ps.gz`.

[Han94] O. Hansen. A Tool for Optimizing Programs on Massively Parallel Computer Architectures. In *High-Performance Computing and Networking, Volume II*, volume 797 of *Lecture Notes in Computer Science*, pages 350 - 356, München, April 1994. Springer Verlag.

[HKOW96] O. Hansen, J. Krammer, M. Oberhuber, and R. Wismüller. A Scalable Tool Environment for Observing the Runtime Behavior of Massively Parallel Applications.

*Parallel Computing*, To appear in Q1 1996.
`http://wwwbode.informatik.tu-muenchen.de/~wismuell/publications/pc95.ps.gz.`

[Lud93a] T. Ludwig. Load Management on Multiprocessor Systems. In A. Bode and M. Dal Cin, editors, *Parallel Computer Architectures — Theory, Hardware, Software, Applications*, pages 87-101. Springer, Berlin, 1993. Lecture Notes in Computer Science, 732.

[Lud93b] T. Ludwig. UPAS — Universally Programmable Architecture and Basic Software. In P. P. Spies, editor, *Euro-ARCH '93, Munich, Germany*, pages 660-671, Berlin, 1993. Springer. Informatik aktuell.

[LWB+95] T. Ludwig, R. Wismüller, R. Borgeest, S. Lamberts, C. Röder, G. Stellner, and A. Bode. THE TOOL-SET – An Integrated Tool Environment for PVM. In *Second European PVM Users' Group Meeting*, Lyon, France, September 1995.
`http://wwwbode.informatik.tu-muenchen.de/~wismuell/publications/europvm95.ps.gz.`

[LWSB96] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. OMIS — On-line Monitoring Interface Specification (Version 1.0). Technical Report TUM-NUMMER, SFB-Bericht Nr. XXX/XX/XX A, Technische Universität München, Munich, Germany, February 1996.
`http://wwwbode.informatik.tu-muenchen.de/~omis/XXXX/omis.ps.gz.`

[OW95] M. Oberhuber and R. Wismüller. DETOP - An Interactive Debugger for PowerPC Based Multicomputers. In P. Fritzson and L. Finmo, editors, *Parallel Programming and Applications*, pages 170-183. IOS Press, May 1995.
`http://wwwbode.informatik.tu-muenchen.de/~wismuell/publications/zeus95.ps.gz.`

[SMP95] T. Sterling, P. Messina, and J. Pool. Findings of the Second Pasadena Workshop on System Software and Tools for High Performance Computing Environments. Technical report, Center of Excellence in Space Data and Information Sciences, NASA Goddard Space Flight Center, Greenbelt, Maryland, 1995.