OMIS — On-Line Monitoring Interface Specification [1]

Version 2.0

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 -8243 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

July 15, 1997

[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 uniform 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** target platforms (consisting of hardware, operating system, programming libraries etc.) will be reduced in complexity from **n** x **m** to **n** + **m**. In addition, it will eventually be possible to simultaneously use tools of different developers and to compose uniform tool environments.

As a second step following this specification the research group at LRR-TUM has designed an OMIS compliant monitoring system (OCM) which is currently being implemented. It will be available for the PVM and MPI programming models running on networks of workstations. A set of interactive and automatic tools will be made available for OCM.

The present document defines the goals of the OMIS project and lists necessary requirements for such a monitoring system. It is an improved and enhanced version of OMIS 1.0 which was published in February 1996. 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 further 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 (omis@informatik.tu-muenchen.de). If you would like to participate in the implementation or would like to use OCM for your own tool development please feel free to ask us!

Contents

Preface to this Document Version

This document version is an enhancement and improvement of OMIS 1.0 which was published in February 1996. Critical feedback from various persons and recognition of several problems have lead to OMIS 2.0. There is no change of the basic principles of the interface specification. However, services were renamed for better consistency and sometimes extensions were made to meet requirements with specific tool scenarios.

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 detailed description of services provided by the monitoring interface.

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 (omis@informatik.tu-muenchen.de).

Please see also chapter History for the document history.

Part I

The OMIS Project

Chapter 1

Motivation

Parallel processing is a key technology of 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 clusters and networks of workstations (COWs, NOWs) play an important role, as often only their aggregate power can 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.

In order to reduce the complexity of software development we need powerful tools throughout the life cycle of parallel and distributed software. Ideally, they should cover all issues from specification to production runs of the programs. They are essential in order to achieve a high efficiency with software engineering, where quality of code and time to market are the most important factors.

The present report refers to tools where the program is already operational, i.e. where the first prototype is running. Starting from this phase we usually apply tools that cooperate with the running program by observing and modifying its behavior. Debuggers are examples for such tools. The tools get their information by a so called monitoring system. It establishes a software layer that connects the running program system (consisting of hardware, operating system, and application program processes) to the tools or tool environments and guarantees observation and manipulation capabilities.

Due to its intermediate position, the monitoring system has two interfaces: one towards the tool, the other one towards the running program. Up to now, not much attention was put on these interfaces, especially on the issue of proposing standards for them. There are some few approaches undertaken to fix at least the tool/monitoring system interface. The p2d2 debugging concept by Hood/Becker/Cheng [Hoo96] and the Universal Measurement Architecture (UMA) approach by the X/Open group [Gro95a, Gro95b] are examples for this.

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. Also, 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 adaptations of publicly available programming libraries. Usually, the programmer can use only one tool at a time or must even specially adapt his program before applying another tool. Finally, there are no uniform tool environments in the sense that an application developer could use the same set of tools on different target architectures. Thus, it is not astonishing that the results of the Second Pasadena Workshop on System Software and Tools for High Performance Computing Environments [SMP95] stress a considerable lack of sophisticated tools and tool environments.

Let us now have a closer look at the types of tools that would be required. Tools that 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 facilities 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 that 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 forwarding 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 the 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 report is to lay the basis for the design (and also the implementation) of on-line monitors for parallel environments with distributed memory. We would like to stress here that a monitoring system for on-line tools usually provides a superset of the functionality that is required by off-line tools.

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.

Without a reasonable standard, 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.

Having an agreed on on-line monitoring interface, different tool developer groups could design and implement new interactive and automatic 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** x **m** to **n** + **m**, thus bringing more tools onto more parallel and distributed systems which finally would ease software development on these architectures. A further implication of having standardized 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.

If we compare different tools we will find that they use a considerable part of the monitor interface that 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 monitoring system that can be adapted to different underlying hardware/software environments and can be expanded for connecting it to new tools.

The goal of the OMIS[1] project that will be presented in this report is to provide a viable basis for better tools with respect to integration of the individual tools into a single environment. It is an approach to define a powerful interface between tools for parallel and distributed computing and monitoring systems that control computers and applications. Eventually, OMIS will help to build unified tool environments, where the same set of tools runs on a big variety of target platforms. The goal of this report is to specify OMIS and to explain, how it can be applied to reach the described goals.

The approach presented in this document version will in its current phase concentrate on systems with distributed memory architecture and programs being implemented with message passing libraries. However, the concepts introduced are not contradictory to the shared memory programming paradigm. Necessary enhancements to our approach for covering also these environments will be included in the next document version.

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

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 nine 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 stands for TOols for Parallel SYStems; for detailed information please refer to [Bod94, 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 the frameworks of other projects and direct industry cooperations. Examples are an adaptation 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, Siemens, Genias, and others. (For more details please refer to the annual report of LRR-TUM[1].)

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, WOKH96]. Both tools are successors of former TOPSYS tools and became an integral part of the PARIX parallel programming system for PowerXplorer systems in 1994. In 1996 we adapted the debugging tool to the new CC systems running EPX, a special version of PARIX for embedded systems. Again, the main effort concerned an appropriate port of the underlying monitoring system to the now used AIX operating system.

Within the framework of the Prepare project we developed a special monitoring system for HPF programs together with a dedicated performance analysis tool.

[1]http://wwwbode.informatik.tu-muenchen.de/archiv/diverses/jber94/jb.ps.gz and .../jber95/jber.ps.gz

Currently, the research groups of Karl/Hellwagner at LRR-TUM are developing an SCI-based networking platform that will exploit the distributed shared memory programming paradigm (DSM) [AHKL96, HKL97a, HKL97b]. As soon as the architecture is in a stable state, tool development will start based on the OMIS approach.

In the last years a change in paradigm took place: distributed memory multiprocessors systems are no longer the only vehicles of parallel processing. In addition, workstation clusters, SMPs, clusters of SMPs, and DSM systems enjoy an increasing popularity. The style of programming 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. The implementation will initially be based on PVM which represents a 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. Adaptions to genuine parallel architectures and the MPI programming model will 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 program flow visualizer, a tool for deterministic program execution, a dynamic load balancer, a consistent check-pointing facility [Ste96a, Ste96b], a distributed file system [Lam97] and a trace data comparison tool.

2.2 Project Goals

The central research topic of the OMIS project 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 concentrate on the question of how to keep the interface specification abstract enough to guarantee its applicability to various other environments.

The detailed list of goals comprises research oriented issues, design and implementation issues, and standardization issues. The project is 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 major objective is to define a tool/monitor-interface that meets three main requirements: First, it should be extensive and complete in the sense that the functionality of all common types 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. Third, the approach should exhibit a high adaptability to current and future programming paradigms. The spectrum reaches from message passing to shared memory, remote procedure call, and client/server programming concepts.

From the discussion in section 2.1 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, on a regular basis, and on an event driven 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 detect events that are specified by the user; this implies a high degree of configurability.
- 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 define a standardized tool/monitor-interface and to provide means to efficiently design and implement monitoring systems that fulfill the above mentioned requirements. With an OMIS compliant monitoring system being connected to a running HW/SW system, several tools from possibly different developers can concurrently watch and manipulate the execution of application programs. Tool interoperability and integration of future tools into existing environments are the most important features OMIS is able to support. In addition, we will reach the goal of having uniform environments where identical tools exist for a variety of target architectures.

2.3 History and Future of OMIS

A first version of OMIS was published in February 1996 in newsgroups and at workshops [LWSB96, LWOB97]. We received valuable feedback from other tool designer groups giving us details what type of interface is necessary to meet their special requirements. From that we produced the current refinement of the document, i.e. OMIS 2.0 [LWSB97]. We would be grateful to also get feedback on the current project status.

Starting from a sophisticated specification document two further goals have to be achieved. First, we are currently implementing an OMIS compliant monitoring system (OCM) that serves as a basis for THE TOOL-SET. The design phase started in January 1996 and was completed by end of 1996 (see [Gei96, Uem96, Zel96] for details). The first implementation phase will concentrate on PVM as programming model and Solaris 2.5, HPUX, and LINUX as operating systems running on their corresponding target hardware. In the next step we will support MPI as further programming model and AIX and IRIX as additional operating systems. Thus, we will 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.

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.

2.4 Integration of OMIS with other projects

Since OMIS 1.0 was released it was considered to be integrated into the following projects:

- The DOSMOS project (Distributed Objects Shared MemOry System) by Lionel Brunie, Laurent Lefevre, Olivier Reymann at Ecole Normale Supérieure of Lyon, France [BLR96].

  DOSMOS provides the user with a distributed shared memory environment that is based on shared objects of the programming language C.

- The ⊲PBEAM project by Stefan Petri (formerly Technische Universität Braunschweig, now LRR-TUM) and Bettina Schnor (now at Universität Lübeck) [PSLS96].

  ⊲PBEAM is a system to support load balancing and fault tolerance on a cluster of workstations by migration and checkpoints of running processes. Thomas Gottschalk at Technische Universität Braunschweig made a first integration approach [PSL96].

- The GRADE project by Peter Kacsuk at KFKI MSZKI,Research Institute of the Hungarian Academy of Sciences, Budapest, Hungary [KCD+97, KDF96][2].

  GRADE is an integrated environment for development and debugging of parallel programs. It is based on the graphical design language GRAPNEL.

- The MAD project by Jens Volkert, Dieter Kranzlmüller, and Sigfried Grabner [KGV96]

  MAD is an evironment for debugging message passing programs that comprises several single tools for different aspects of this issue.

Discussions with participants of these projects and many other researchers were valuable contributions to OMIS 2.0. We would like to thank all of those people for their constant interest in our work.

2.5 Project Policy

The project policy will be to release all software products being developed by LRR-TUM 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.

[2]Funded by BMBF within the WTZ project framework under number UNG-053-96.

Chapter 3

Requirements

A general requirement is derived from the goals of the project itself. The monitoring interface must be powerful enough to give on-line and off-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) and programming paradigms.

In detail, the design of the monitoring interface imposes several requirements which the specification will have to meet. This chapter will summarize the most important of them. 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 our implementation of an OMIS compliant monitoring system (OCM) dedicated to a given system architecture. However, they will not influence the tool/monitor-interface but the monitoring system's internal structure and the monitor/program-interface.

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 optimal versatility the monitoring system not only has to offer a fixed functionality at its interface level. Instead it has to offer a language by which services can be combined to powerful service requests. By following the event/action paradigm of the monitoring system the user may compose complex requests by specifying a certain event to be observed and a list of actions to be performed when the event takes place. The monitoring system acts as an intelligent controller and manipulator that gets its instructions via the tool/monitor-interface. The programming language of the monitoring system is defined by the form of the requests that are sent from the tools. Similar to ordinary languages commands are expressed in form of strings. They may be composed of several substrings with individual meanings. 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 that we might like to monitor we can state the following requirements: As implementations of monitoring systems will in the first phase be realized for distributed memory environments we will definitely be interested in execution objects and communication objects. Functions of the interface should give access to these object types on several levels of abstraction: e.g. with execution objects we are interested in processes and threads and would like to know more about procedures and individual statements of their code. With communication objects such as 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 monitoring system also creates 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.

The monitoring interface is now divided into two logically separate parts, one providing the user with basic services, the other one with extensions necessary for different programming environments (e.g. PVM, MPI). 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. Later chapters will however show how to introduce these extensions.

Any of the tools will finally interact with all of 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 of the type of functionality that is required by individual tools, using The Tool-set as an example:

- The Debugger[1]

  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 accessible via the tool/monitor-interface is of crucial importance.

[1]For a brief description of what the functionality of the following four Tool-set-components will be, please refer to [LWB+95]

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. In addition, new programming paradigms need to be covered that differ from the message passing paradigm we are currently concentrating on. Shared memory programming will be the next step but also object oriented programming and client/server programming using interface definition languages (IDLs) might be of interest. This conceptional requirement will be fulfilled with our specification by designing means of how to enhance the tool/monitor-interface. The specification provides means to plug in new components that handle these future concepts.

A second conceptional requirement is imposed by the variety of tools that will use this interface in a environment of parallel and distributed machines. We distinguish tools with and without a graphical user interface. The first group will usually have a single point of control, e.g. a tool environment on a workstation. From that single point of control the tool will communicate with the monitoring system, i.e. the individual monitors on the nodes. On the other hand we will have tools without a 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.

Furthermore, different types of architectures have to be handled as well. Especially SMP workstations and clusters of SMP workstations will be considered. With these architectures OMIS based tools work at the same level of resolution as the operating system does: if the assignment of a thread to an individual processor is not controllable by the program it will neither be controllable by a tool. In this case the additional complexity is hidden and the SMP workstation acts just like a single processor node.

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 look at this issue here. Interaction between a monitor and a tool must be handled by a kind of communication mechanism (e.g. message passing or RPCs). 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 combine a sequence of requests of available basic services into a single request. E.g. to get an overview over the system utilization, the tool could ask each monitor for the idle time percentage. Instead, it will issue a combined request to one monitor that thereupon sends requests to all others and returns a collective response to the calling tool. These types of composite services are of special interest in situations where the requested information is the result of a computation based on information gathered from different computing nodes. In this case the monitor itself can do some pre-calculation in order to reduce communication overhead. The syntax of requests is kept simple to keep the overhead necessary for parsing low. Complex commands that express e.g. loops or alternatives can be realized in the extension part of the interface specification.

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 for example a performance analysis system wants to measure the mean time between 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 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 execution objects (processes, threads) which usually spawn over a set of nodes. A node from our point of view is a computing device that offers a single system image to the user. It could e.g. be a single processor workstation, an SMP workstation, or a cluster of workstations providing a single system inage. Observations and manipulations below the level of the system image are only possible if this is supported by the operating system[1].

Thus, this paper does not only present an interface specification, but also specifies the kind of environment that can be handled by the interface. As the specification exhibits a high degree of flexibility, we expect to be able to integrate it also into other architectural environments, even into those that differ considerably from architectures found with the message passing paradigm. A future version of OMIS will cover shared memory environments; other architectures will be carefully considered.

Let us now look at the components participating in such an environment. Figure 4.1 shows an abstract view in which the individual nodes of the parallel system are not yet visible. The application programs consist of a certain number of execution objects that cooperate by means of communication objects provided by the parallel programming library or the operating system (e.g. message passing between nodes, shared memory on one node). The management of the execution objects and other organizational work is performed by special modules of the programming environment (e.g. daemons) or directly by the operating system. Technically, this means that there might be libraries and daemons with which the program cooperate. The complete complex runs on the target architecture, i.e. on top of the operating system and the hardware. The complex consisting of target architecture, programming libraries, and daemons used is called target platform.

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 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 distributed tools. Two

[1]For example, inquiry services might be able to determine the number of CPUs in an SMP machine although none of them can be accessed directly to measure its load.

[2]Note, that with many modern parallel computers and with workstation clusters the host may actually be part of the target machine.

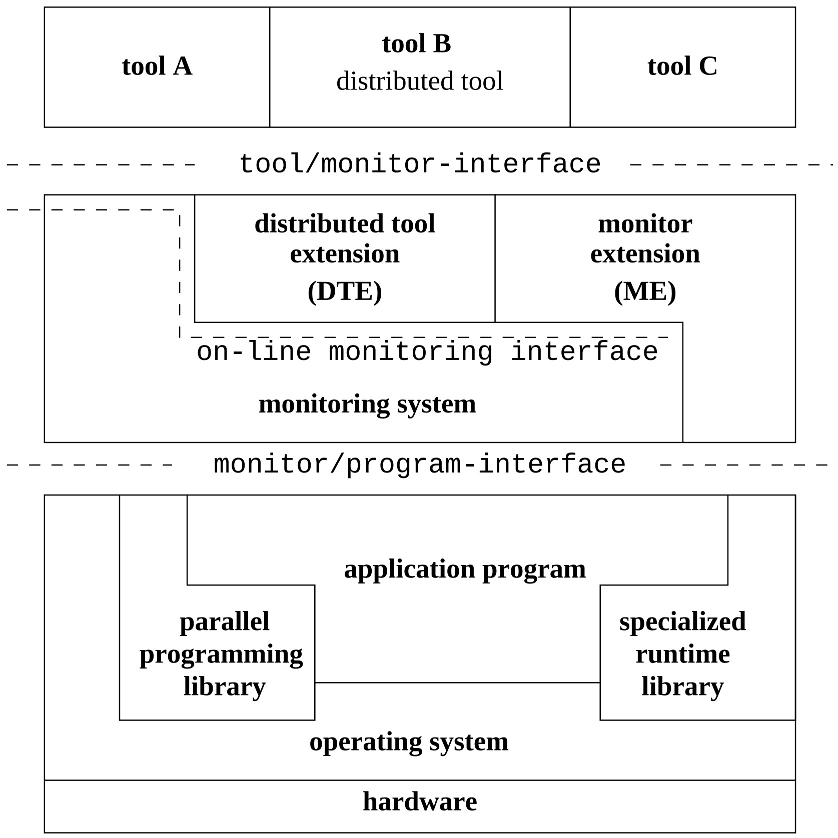
tool A
tool B
distributed tool
tool C
tool/monitor-interface
distributed tool extension (DTE)
monitor extension (ME)
on-line monitoring interface
monitoring system
monitor/program-interface
application program
parallel programming library
specialized runtime library
operating system
hardware

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

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. 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 distributed tool extension because many activities can be handled more efficiently directly on the nodes (e.g. preprocessing of information data).

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 the monitoring system will have to handle different information types depending on the low-level module with which it inter-acts. Details will be discussed in later chapters.

The tool/monitor-interface is what the developer of a tool finally will use. OMIS specifies semantics only for the on-line monitoring interface and provides means to extend this interface. We end up with a definition of the tool/monitor-interface where the basic services provided by

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

the on-line monitoring interface are defined with their syntax and semantics whereas all of the extensions are only defined with respect to their syntactical structure.

The interaction between tools and the monitoring system is handled via asynchronous procedure calls. The tool invokes a service request and either waits for results coming back from the monitoring system or specifies a call-back to be invoked when results are available.

The interaction between the monitoring system and the monitor extensions and distributed tool extensions takes place via function calls and activation of call-back functions.

Let us finally have a look at requirements that must be met by the target architecture in order to successfully support the OMIS approach. First of all, the operating system must support multi tasking. This is obligatory for the various activities that are performed by the monitoring system while the program is running[4]. The target architecture also must provide means for the manipulation of execution objects (processes and threads)[5]. Otherwise only observation would be feasible and the approach would lose its most important part. Finally, we need means for communication between the modules belonging to monitoring system, the tools, and the underlying target platform.

[4]It would be possible to link all components together and run tools and program on e.g. a mono tasking parallel computer environment. However, there is not much sense in setting up an approach like OMIS for such a limited architecture.

[5]With modern operating systems this is achieved via the `proc`-file-system or the `ptrace`-interface.

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 a language in order to achieve the needed flexibility. The tool/monitor-interface basically consists of a single procedure that can be invoked by both centralized tools and components of distributed tools. In addition, a few other procedures are available for maintenance purposes (see Section 7.1). The main procedure receives a string as an input parameter, interprets this string, and returns a result that is represented as a data structure. 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 matching the event definition 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 nodes the monitoring system is responsible for; therefore, a distributed tool component can request services not only for objects on its local node, but also for any object. Likewise, the actions associated with an event can be requests for services on objects located on a node different from the one where the event occurs. The monitoring system automatically takes care of forwarding the requests to the proper nodes. In addition, we allow services that are global, i.e. that involve more than one node.

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. In section 5.5 we discuss the the scope of tools, i.e the set of objects that they control. Section 5.6 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 can classify services according to their input/output behavior:

1. **Information services**. These services are exclusively meant for observation of the program and the monitoring system. The result of their invocation will contain information about the current state of the monitored system (including the monitoring system itself). Examples are "return the value of a variable x of process y" or "return the CPU time of a process".
2. **Manipulation services**. These services manipulate the objects that are listed in the parameters of the service call. Thus, they change the internal state of the program or the monitoring system or both. Examples are "stop a process", "set a variable of a process to a given value" or "raise a user[1] defined event".
3. **Event services**. In contrast to the two classes already introduced, event services do not have a direct reply. Instead they are used to trigger lists of manipulation services and information services which are called action lists. An event service defines a class of events to be observed. We call an 'event' the situation where a specific change in the state of an observed object occurs. Whenever an event belonging to that class occurs it triggers the specified action list. The replies to the latter are collected and sent back to the tool. In fact there are also reply values of an event service. They are called event context parameters and their values are handed over to the manipulation services and information services to be invoked. By that the latter are parameterized and may react on events in different manners depending on the actual values of the event context parameters.

   We do not know in advance how many times an event of the given class will occur nor when this might happen. Thus, the tool does not know how many replies might arrive and when.

   Obviously, an event service cannot be used as a complete request as its result can only be observed in the form of results of action lists triggered by this event.

The description of services in Chapter 8 and Chapter 9 is structured according to these classes.

The interface specification defines the way requests can be composed by using above mentioned categories of services. Details on the syntactical structure of requests will be given in the next section.

Logically we have to distinguish two situations: requests that are unconditional and yield results immediately and conditional requests where something has to be done whenever an event of a certain class occurs. Unconditional requests are composed by a well-defined set of manipulation services and information services of any size. Rules for the sequence of invocation of the individual services will be introduced later. With conditional requests we simply add one event service to an unconditional request. It specifies the condition under which the request will be triggered.

- We already mentioned that the versatility of OMIS lies in the mechanism of extending the specification. This divides services into two categories.

  Basic services are independent of the concrete programming libraries employed for the parallel program. They establish the core of OMIS and thus of every OMIS compliant implementation of a monitoring system. The set of basic services in OMIS 2.0 is defined in Chapter 8.

  Extension services do not belong to the OMIS core. Instead we need an extension for every programming library we want to use with our programs. Chapter 9 defines an extension

[1]The user here is a tool, not the application programmer.

for PVM 3.3. If a tool developer wants to implement tools for e.g. the parallel file system PIOUS he will have to provide an extension for this purpose. However, one implementation of these additional services will be sufficient for others to also develop tools for PIOUS.

The OMIS research group will take over the task to coordinate the design and implementation of extensions being made by third parties.

- Services can be classified according to the type of objects they refer to. On the first level, we distinguish between system objects and monitor objects. System objects are those objects being part of the monitored application or of the hardware the application is executed on. Currently, four different types are supported by the basic services:

  1. **processes**,
  2. **threads**,
  3. **messages** and **message queues**, and
  4. **nodes** of the parallel or distributed computing system.

  This selection reflects of course our decision to concentrate on the message passing paradigm in the current version of the specification. Although not yet included here we are working towards extending OMIS for the shared memory programming paradigm. The integration will add new objects to the above list, e.g. memory regions.

  Additional objects for specific message passing libraries to be monitored appear with OMIS extensions. Chapter 6 will discuss that in detail.

  Monitor objects are those objects that are introduced by the monitoring system itself. For example, each conditional service request 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 object oriented paradigms for the tool/monitor-interface instead of the current one, that is object based but does not apply inheritance. By exploiting inheritance, object oriented techniques could provide a way to define the interface at an abstract level independent of the supported programming paradigms and concrete libraries. Services specific to a certain programming library could then be realized by an extension to the generic monitor that implements object classes (e.g. PVM task, MPI processes) derived from the interface's base classes (e.g. abstract process). See item 1 in our requests for comments (Chapter 12).

5.2 Introduction to Monitoring Services

This section is intended to give an impression of how the syntax of service requests and service replies looks like and how unconditional and conditional requests are composed. For a complete formal description of the request and reply syntax and semantics, please refer to Chapter 7. The discussion here will not cover all relevant aspects. Instead it will try to introduce the concepts that will be used.

5.2.1 Unconditional Requests

An unconditional request, also called action list, is syntactically composed of a list of information services and manipulation services. A simplified definition looks as follows[2]:

```
action_list     ::=  action   |   action [ ';' ] action_list
action          ::=  service_name '(' parameters ')'
```

The individual actions of an action list are concatenated either directly or by means of a semicolon. This distinction gets important as soon as individual actions refer to different nodes of the target system. In this case it is possible to execute actions in parallel. A direct concatenation by definition expresses the possibility for parallel execution whereas a semicolon forces the monitors on the individual nodes to globally synchronize before they continue with the next action. For example an action list "a b ; c" expresses that actions a and b can be executed in parallel (if possible) but both of them have to be completed before action c can be invoked.

The *service_name* identifies the requested service. It may either belong to the class of basic services or of extension services. As extensions are optional it is the task of the monitoring system to check for available extension service identifiers and provide the tools with the necessary information.

Of course we also need parameters for the individual actions of our list. Parameters are defined as follows:

```
parameters      ::=  parameter_list   |  ε
parameter_list  ::=  parameter   |  parameter ',' parameter_list
parameter       ::=  integer  |  floating  |  string  |  token  |  binary
                     |  list  |  ev_ctx_param
ev_ctx_param    ::=  '$' identifier
token           ::=  identifier
list            ::=  '[' parameters ']'
```

Each parameter is either of the standard data type **integer**, **floating**, **string**, or **binary** or of the special data type **token**, **list**, or **ev_ctx_param**. The meaning of the standard data types should be intuitively clear. Detail will be given in section 7.2.2. The special data type **token** is used for addressing in a platform independent way objects of any type, i.e. system objects and monitor objects. Finally, the data type **list** allows to compose lists of all other data types including the list type itself.

Where the service name specifies the action to be invoked it is the role of the token to determine the objects that are to be used. In the most simple case the token is directly an identifier for an object the service works on. E.g. the token might specify a process where the service is "stop a process". More complex situations can be mastered by using token lists. For the same service a token list might specify a set of processes to be stopped.

Tokens also offer an expansion mechanism. A node token used with the above service means: stop all processes on the specified node. The node token is expanded to a list of process tokens before the service is activated. The other direction is called localization. If we use a thread token with the same service, the system replaces the token with the corresponding process token, i.e. with the token of the process that runs the thread.

[2]A special syntactical construct to lock the monitoring system during the execution of an action list is left out here for simplicity.

Expansion and localization can also be mixed in a token list: a complex list might comprise node tokens and thread tokens. It is translated into the list of tokens of the corresponding process before activation of the service.

A special semantics is defined for empty token lists. They are expanded to a non-empty list of tokens refering to all currently monitored objects of the type the service works on[3].

Let us assume that a service needs to address all currently monitored processes in the system. The process token input parameter of the service gets an empty token list when the request is sent from the tool to the monitoring system. The empty token list is expanded at the moment where the service is to be performed. By means of this just in time expansion we are able to handle dynamically changing sets of objects (nodes, processes, etc.). A more detailed description of the token data type can be found in section 7.2.2.

Finally, the event context parameter **ev_ctx_param** is used to transfer information from event services to information services and manipulation services. A parameter of this type describes an output parameter of an event service (e.g. $node, $time etc.). A fixed set of event context parameters is provided by OMIS itself. Extensions are free to introduce additional parameters.

5.2.2 Conditional Requests

We will now discuss the conditional request that adds some further rules to our request syntax. We already mentioned that a conditonal request is composed of an event service definition and an action list[4]. We define:

request	::=	[*event_definition*] ':' *action_list*
event_definition	::=	*service_name* '(' *parameters* ')'

With conditional requests the protocol between tools and monitoring system is more complicated. The tool gets a reply whenever an event matching the specification occurs and the action list was performed. However, when the tool gets the replies, how can it identify them? A further question is: how does the tool know whether or not the event detection was successfully installed for this request? An answer that solves these two problems is to send a first reply to the tool when the request is analyzed. This reply carries information on the instantiation of the corresponding event detection as well as a token that identifies the request. When the event finally is triggered the corresponding request is identified by the callback function invoked and its actual result parameters.

Conditional requests can be enabled and disabled. By definition, a conditional request is initially disabled, i.e. there must be an explicit request for activation.

For a useful semantics of the tools it is crucial that any object under investigation does not change its state after an event has been detected and before all actions were performed (unless — of course — actions change that state by themselves). This is ensured by defining that the execution of an object that triggered an event has to be suspended immediately after event recognition. It is resumed after the completion of all actions in the list.

A detailed description of issues relevant to conditional requests is given in section 7.2.3.

5.2.3 Service Replies

Service replies are very complex by nature. In case of a successfull completion of an action list they carry result values of all the individual actions of the list being performed on the specified

[3]There are some more details concerning the final list of tokens that will only be introduced in later chapters.

[4]Thus, a request from a tool to the monitoring system is conditional or unconditional depending on the fact whether or not an event service is specified.

objects. E.g. one service of the action list might collect information of all processes on all nodes. The result of this action is a list of object tokens together with the result values of the specific service issued. Another service of the list might manipulate all processes on all nodes. As manipulations might be successful or erroneous the service reply must also be able to transfer this information back to the tool.

Replies of the individual services of the action list are put together to one reply for the complete list which finally is sent back to the tool. Obviously, the reply is hierarchically structured as many action and objects will be referenced.

With conditonal requests we find two additional situations that have to be covered. The first one concerns the requests by themselves: definition, enabling, disabling, and deletion result in a special reply sending back to the tool information on the success of these functions. The second one refers to the occurence of an event that matches an event definition. The reply must be assigned to the correct request issued before. This is achieved via the callback function triggered and/or the parameters passed via this call.

Details on the reply data structure and its individual components are discussed in depth in section 7.3.

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 and extension services will not be defined until later in this document, their semantics should be intuitively clear in the examples. The primary goal is to give an impression of the interface's structure and expressiveness, rather than of its concrete services.

5.3.1 Performance Analysis of PVM Programs

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

No.	request string	result token

1	thread_has_started_lib_call([p_4178],"pvm_send") :
	timer_start([pa_t_1])
	counter_add([pa_c_2],$par5)
		c_1

2	thread_has_ended_lib_call([p_4178],"pvm_send") :
	timer_stop([pa_t_1])
		c_2

3	thread_has_started_lib_call([],"pvm_barrier") :
	print(["pvm_barrier entered",$node,$proc,$time])
		c_3

4	thread_has_ended_lib_call([],"pvm_barrier") :
	print(["pvm_barrier left",$node,$proc,$time])
		c_4

5	: csr_enable([c_1,c_2,c_3,c_4])	

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

The event services used in this example are of special importance as they bridge the gap between the universal monitoring interface and an interface adapted for the PVM programming model. These services allow to monitor library calls of any programming library, in our example of the PVM package[5]. The service **thread_has_started_lib_call** is matched by any event where an appropriate library call is started. Accordingly, **thread_has_ended_lib_call** is matched by any completion of such a call.

Each event service is defined for objects on a certain level of abstraction (i.e. processes, messages etc.). Tokens are automatically adapted to the appropriate level of abstraction by two mechanisms called expansion and localization. For details on token hierarchies please see section 7.2.2.

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

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

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

While the events used in the example are basic services defined by this document, all the actions (with the exception of **csr_enable** and **print**) come from a distributed tool extension. Thus the performance analyzer can use its own semantics for integrators and counters.

5.3.2 Debugging of MPI Programs

The following example shows how the basic service requests defined by this document can be used during debugging. Assume that the debugger wants to be notified whenever process p_1123 starts sending a message, or whenever process p_1234 reaches instruction address 0xfe08, and that it wants to know the process identifier, the procedure stack, and the contents of all general purpose registers. This can be achieved by using a user defined event:

No	request string	result token
1	: user_event_create()	e_1

2	thread_has_started_lib_call([p_1123],"MPI_Send") :
	user_event_raise([e_1],[],0)
		c_1

3	thread_reached_addr([p_1234],0xfe08) :
	user_event_raise([e_1],[],0)
		c_2

4	user_event_has_been_raised([e_1]) :
	print([$proc])
	thread_get_backtrace([$proc],0)
	thread_read_int_regs([$proc],0,32)
		c_3

5	: csr_enable([c_3]) ; csr_enable([c_1,c_2])	

At first we define a new user event which subsequently can be used for both triggering a synthetic event and detecting the very same synthetic event. The definition yields a token for this user defined event by which it can be referenced in the next requests.

The second request raises this event whenever an MPI send operation is performed. Similarly, request number three defines the same event to be raised on the condition of reaching a certain address with a thread object. Both requests act as a kind of logical OR operator for event: whenever sending is performed or a certain address is reached a user defined event will be raised.

[5]Note: This can not be achieved without modification of the PVM library. However, we expect to have this modifications handled automatically either at link-time or during run-time.

Request 4 defines what has to be done in this case: we want to get the process identifier of the triggering object and its node identifier, get its procedure stack, and read its register values.

Up to now all conditional requests are disabled. At first we enable the handling of our user defined event by activating **c_3**. The possible invokation of this event is activated by enabling **c_1** und **c_2**. Now the complete request sequence is active and triggers identical actions by different events.

After the actions have been executed, the process continues execution. To stop the process with the occurrence of the event, simply add **thread_stop($proc)** to the action list of request 4.

5.4 Interface Procedures

We will now give a first short introduction into the concepts of interface procedures that are used between the tools and the monitoring system. An exact description of this topic will follow in section 7.1.

Essentially, OMIS defines one function to handle the cooperation between tools and the monitoring system. This function is called **omis_request** and has the following ANSI-C prototype when provided for a C language binding:

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

The function is used to issue both unconditonal and conditional requests. Its behavior in terms of blocking or non-blocking is defined by its parameter values and may be tuned to fit various situations.

The function definition involves four basic concepts:

1. The request sent to the monitoring system is always represented by a character string.

2. A call-back function provides means for the monitoring system to transfer result data back to the tool. This is essential for conditional requests which result in any number of replies. The call-back function is activated whenever a reply from the monitoring system has to be transferred.

3. Flags define behavioral variants of the **omis_request** function. Most important they allow to e.g. suppress simple ok-replies or wait for for acknowledgement of proper event activation.

4. A reply parameter is used whenever we get direct results as opposed to indirect results via the call-back function. This reply parameter might e.g. denote reply data from unconditional requests. Any reply sent by the monitoring system occupies memory in the address space of the tool. It can be freed via the **omis_reply_free** function.

For centralized tools, for which the **omis_request** function will be realized by some kind of remote procedure call mechanism, we need several additional functions. Their main purpose is to initialize a connection with the monitoring system and to wait for replies to be received. Details are given in section 7.1.

5.5 The Tools' Scope

A very important question is the scope that every tool has, i.e. the objects it can observe and manipulate. If we were faced with static systems only, i.e. single user single application environments with only one tool, the situation would be simple: the tool's scope would just be all objects of the running system. However, OMIS wants to provide means to handle dynamic systems where the number of objects varies during time. New execution objects may be created or old ones may no longer exist. Likewise nodes may be added to the environment or be deleted from it. Also, several tools might co-exist in a tool environment and have to be managed independently by the monitoring system. Finally, OMIS based tools should be multi application capable, i.e. they should support a tool's access to more than one currently running application program.

The main concept of OMIS is that every tool at every moment has a well defined scope, i.e. can observe and manipulate a specific set of objects. The scope of different tools may be different. They may handle object sets that are not identical.

Concerning the ability to monitor concrete objects we define the following rules:

- After the start of a tool and its successful initialization of the cooperation with the monitoring system (via **omis_init()**) the tool has not yet an access to individual objects.
- In order to control nodes and processes the tool has to explicitly attach to every individual object. Only after an attach operation the tool can monitor the object.
- Transient objects like threads and messages can be monitored if the tools is attached to the corresponding 'container' object (i.e. a process, a message queue). No attach operation is necessary as this would be in conflict with the objects' short life time.
- New instances of nodes and processes are not monitored automatically. Instead, the tool has to issue a conditional service request to get informed whenever new objects of these types are instantiated. If it wants to monitor these objects it explicitly has to attach to them.

Above rules fix a tool's scope. We can see that the set of objects to be monitored in the target system is well defined. Every tool sees only what it wants to see.

Whenever two or more tools monitor the same object the monitoring system has to ensure to properly distinguish these tools. Imagine that two tools measure the CPU time of a process. If one of them is no longer interested in that value and disables the measurement, the monitoring system has to guarantee that it will remain activated for the second tool. On the other hand, as long as there are two or more identical measurements defined, the monitoring system should be able to recognize this fact and to keep the additional overhead by this multiple definition as low as possible.

The varying set of objects produces some logical problems for conditional service requests especially in cases where we use object tokens like "all nodes" or "all processes" in event and action definitions. We have to fix a semantics how to handle the situation where the object set changes during an event definition and the corresponding action list activation. These problems will be discussed in detail in section 8.1.2 and 8.1.1.

5.6 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. The behavior of the interface might be blocking or non-blocking depending on the actual parameters. This will be discussed in detail in the next chapter.

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

- 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.

  The debugging example already demonstrated how to combine two events in order to have a logical OR operation between them. Likewise it is possible to realize a "happened after" relation. Let us reconsider the debugging example. Now, we would like to have the action list executed when first process p_1123 started sending and afterwards process p_1234 reaches instruction address 0xfe08.

No	request string	result token

1	thread_reached_addr([p_1234],0xfe08) :
	print([$proc])
	thread_get_backtrace([$proc],0)
	thread_read_int_regs([$proc],0,32)
		c_1

2	thread_has_started_lib_call([p_1123],"MPI_Send") :
	csr_enable([c_1])
		c_2

3	: csr_enable([c_2])	

  The difference now is that conditional service request does not trigger a user event. Instead it enables c_1. By that we have successfully specified a "happened after" relation.

  Other combinations (e.g. an **and** operator) can be realized in a distributed tool extension as a service that triggers 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 and also new types of objects 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 and the additional functions for object token conversion 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. 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, by providing e.g. counters and interval timers, whose detailed semantics depend on the specific tool.

   Therefore, it is desirable to define new services for this kind of tool-specific data processing and also new token types for the monitor objects provided by these services. We will call this kind of extension a distributed tool extensions (DTE). They can also include specialized, more complex services that are based on the basic services defined by this document including also new events that are derived from existing events via filtering.

   The principal property of distributed tool extensions is that they only use the interfaces defined by OMIS. Therefore, they are independent of the target platform or a specific implementation of an OMIS compliant monitoring system.

2. Some tools may also want to observe additional aspects of application objects or even new application objects not covered by this document. These new objects may be implemented in some specific target platforms or may come from specialized runtime libraries. For instance, if an application uses a parallel I/O system, 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). The interfaces provided by MEs have to comply with OMIS, however, due to their nature, monitor extensions also have to make use of interfaces specific to a target platform or an implementation of an OMIS compliant monitoring system. Thus, these extensions may not be portable.

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 (DT). This kind of distributed tool will typically consist of a set of specialized actions that are triggered either periodically or by events in the monitored application. It is similar to a DTE with the difference, that in a DT there is a set of requests that is defined during the initialization of the monitoring system, while in a DTE all requests are defined by the centralized tool.

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 and new object types to be linked to the monitoring system, the question is how to make them accessible at the monitoring interface. The approach we are using is as follows: Each service is addressed by a unique service name, i.e. a unique identifier string passed to the tool/monitor-interface, which is mapped to the implementing function(s) using a mapping data structure built during initialization of the monitoring system. Likewise, the type of a token is indicated by a string prefix that can be mapped to the functions implementing token conversion using a similar data structure. We now have a couple of requirements that must be fulfilled:

- The service names and token prefixes used by different extensions must be disjunct, otherwise we will not be able to use several 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 of the tools available in a certain environment.
- 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 the 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 every new service at a central location.

We have decided to use the following strategy for extensions:

- Each service name, each token prefix, 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 the above strategy, the data structure 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 currently not linked to the monitor.

The procedure of requesting a new prefix has to be done only once by those groups that plan 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 get 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.

6.3 The Extension Interface

From the above, it is clear that for the extension scheme to work there must be a well defined interface for extensions. Thus, there has to be a specification covering:

- the interface of the functions implementing manipulation, information and event services,
- the interface of token conversion functions,
- the interface used to register new services and token types.

These interfaces will be specified in a later version of this document, after we have collected more experience with the implementation of an OMIS compliant monitoring system for PVM on workstation clusters.

Note that the interface allowing extensions to use other services is already defined: services in an extension can call **omis_request** to invoke other services, in just the same way as a tool does.

Part III

Interface Specification

Chapter 7

General Description

7.1 Interface Procedures

OMIS defines six procedures to access the monitoring interface:

1. **omis_request**: Sends a service request to the monitoring system.
2. **omis_reply_free**: Frees the memory occupied by a reply structure.
3. **omis_init**: Initializes the connection to the monitoring system.
4. **omis_finalize**: Shuts down the connection to the monitoring system.
5. **omis_fd**: Auxiliary function: returns a file descriptor to wait at for incoming messages.
6. **omis_handler**: Auxiliary function: polls for incoming

The first two functions are available for both centralized tools and for distributed tools, distributed tool extensions, and monitor extensions. The other ones are only usable for centralized tools or distributed tools that are implemented as processes separate from the monitoring system, since only these tools have to set up a communication connection to the monitoring system.

The following paragraphs specify the C language version of these procedures. Other language bindings may be specified when needed. The rest of this chapter and Chapters 8 to 9 specify in detail the requests accepted by **omis_request** and the resulting replies.

7.1.1 omis_request

```
typedef unsigned int Omis_flags;

#define OMIS_WAIT_FOR_FIRST_REPLY 1  /* Even if a callback is specified, */
                                     /* block until the first reply is */
                                     /* received and return this reply */
                                     /* as the function's result, */
                                     /* without calling the callback */
#define OMIS_DONT_RETURN_OK       2  /* Don't call the callback for */
                                     /* replies that only contain an OK */
                                     /* status */
#define OMIS_DONT_RETURN_EN_DIS   4  /* Don't call the callback for */
                                     /* replies indicating that the */
                                     /* request has been enabled or */
                                     /* disabled, if the status is OK */
#define OMIS_BUFFER_REQUEST        8  /* This request may be buffered on */
                                     /* the sender side. The next */
                                     /* request without this flag will */
                                     /* flush the buffer. */
#define OMIS_BUFFER_REPLIES       16  /* The replies for this request may */
                                     /* be buffered locally on the sender */
                                     /* side. The buffer will be flushed */
                                     /* after an implementation dependent */
                                     /* period of time */
#define OMIS_DEBUG                32  /* This flag can be used to switch */
                                     /* on debugging output for this */
                                     /* request. The kind of debugging */
                                     /* output produced is */
                                     /* implementation dependent */

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

This function provides a tool's only access to the tool/monitor-interface. Basically, it accepts a service request as a string in parameter **request**, sends it to the monitoring system and passes the reply back to the calling program. Thus, if tool and monitoring system are implemented as different processes, a call to **omis_request** is actually a remote procedure call.

The detailed behaviour of **omis_request** depends on the value of **flags** and whether or not **callback** is a **NULL** pointer. If **callback** is **NULL**, the function sends the provided service request to the monitoring system and blocks until the reply to this request is available. The reply structure is then passed back to the caller as the function's result. As only one reply can be passed back in this way, this mode of calling **omis_request** only makes sense for unconditional service requests.

If **callback** is not **NULL**, the behavior of **omis_request** is determined by the bits set in **flags**. If **flags** is zero, the provided service request is sent to the monitoring system and the function immediately returns to its caller with a **NULL** result. Later, whenever a reply for this request is sent back by the monitoring system, the specified call-back function is invoked, getting the reply structure and the value of **param** as its parameters.

Setting the **OMIS_WAIT_FOR_FIRST_REPLY** flag in **flags** results in the function to block until the first reply is received by the monitoring system and to return this reply as the function's result. Subsequent replies will be passed to the call-back function. This mode is useful for conditional service requests, since the first reply indicates whether or not the event definition could be processed correctly, while the other replies contain the results of the request's action list.

If **OMIS_DONT_RETURN_OK** is set in **flags**, the call-back function should not be invoked when a reply only consists of status values indicating that the service has been executed correctly, but does not provide any further information. When this flag is set, the monitoring system can drastically optimize its internal communication, especially for conditional service requests where the action list only contains manipulation services. However, an implementation is free to ignore this flag.

If **OMIS_DONT_RETURN_EN_DIS** is set in **flags**, the call-back function should not be invoked for replies indicating that the service request has been enabled or disabled, except in those cases where an error occured. This flag only has an effect if **request** is a conditional service request. If the tool needs not be notified on the enabling and disabling of the request sent, setting this flag can decrease the amount of communication between monitoring system and tool.

The remaining flags control whether buffering of requests and replies is allowed. If the flag **OMIS_BUFFER_REQUEST** is set, the request may be buffered until **omis_request** is called with this flag being reset. This allows an implementation to transfer several service requests to the monitoring system in a single communication step. If **OMIS_BUFFER_REPLIES** is set, replies may be buffered by the monitoring system. An implementation must ensure that the buffers are flushed within an adaquate interval of time. Both flags may be ignored by an implementation.

Finally, a flag **OMIS_DEBUG** is provided for debugging. Its intended purpose is to generate debugging output for that sepcific request. However, whether or not such output will be produced, and which information will be presented depends on the specific implementation of the monitoring system. Debugging output will be passed to the error handler (see **omis_init** below) in a reply structure with a status field equal to **OMIS_OK**.

The exact data structure of **Omis_reply** and the syntax of the **request** strings are specified in separate sections throughout the rest of this document part.

7.1.2 omis_reply_free

```
void omis_reply_free(Omis_reply reply)
```

This function returns the memory occupied by **reply** to the system. It is safe to pass a **NULL** pointer to this function.

7.1.3 omis_init

```
Omis_status omis_init(int *argc, char ***argv,
                      void (* error_handler)(Omis_reply reply),
                      int *tool_id)
```

omis_init establishes a connection between a tool process and the monitoring system. Its parameters are pointers to the tool's argument count (**argc**) and argument vector (**argv**), a pointer to a handler function for asynchronous errors (**error_handler**), and pointer to an integer identifier used for tools consisting of more than one process (**tool_id**). The result is an error code as defined in Section 7.3.1.

The parameters **argc** and **argv** are used to pass the tools command line options to **omis_init**. The function will then extract all options from **argv** that are needed for correctly starting the distributed monitoring system. The arguments used will be removed from the argument vector pointed to by **argv**, ***argc** will be adjusted correspondingly. Since the start-up of the monitoring system heavily depends on the target platform, and there is currently no standardized way to perform a connection between parts of a distributed application, we follow the approach taken by the MPI forum and do not standardize the startup procedure. This means that whether or not **omis_init** starts the monitoring system (if it is not already running) and the specific meaning of the arguments in **argv** will be implementation dependent. The only requirement is that after a successful completion of **omis_init**, the tool can issue service requests to the tool/monitor-interface using **omis_request**. **omis_init** only establishes the connection to the monitoring system, it does not attach the montoring system to any node or process.

If non-NULL, **error_handler** specifies an error handling function, which will be invoked when some error occurs that is not correlated with a specific request (request specific errors are reported in the request's reply). The handler will receive a reply structure with only one **Omis_service_result** element containing an error code and a description (see Section 7.3 for details on reply structures). The handler will also receive debugging output, if the flag **OMIS_DEBUG** is set for a request (see **omis_request** above). Debugging output is indicated in the reply structure by a status field equal to **OMIS_OK**.

The last argument, **tool_id** is an in-out parameter that is only necessary for distributed tools implemented as a set of processes separate from the monitoring system. In this case, one of these processes has to play the role of a master. It has to pass a pointer to a variable containing a 0 for **tool_id**. After completion of the **omis_init** call, this variable contains a unique tool identifier. The master then has to pass this identifier to all the slaves, which in turn call **omis_init** with the supplied identifier. In this way, the monioring system knows that the processes form a single tool rather than different ones. Note however, that the ability for different tools to attach to the monitoring system at the same time is not required by this specification. A centralized tool can simply pass a **NULL** pointer to **tool_id**.

Typically, the **main** function of a centralized tool will pass pointers to its argument count and argument vector and an error handler to **omis_init**:

```
    main(int argc, char **argv)
    {
      if (omis_init(&argc, &argv, err_handler, NULL) != OMIS_OK) {
         << Error handling >>
      }
      ...
    }
```

The startup of a distributed tool implemented as separate processes will look like this:

```
    master process:                      slave processes:

    main(int argc, char **argv)          main(int argc, char **argv)
    {                                    {
      int id = 0;                          int id;
      if (omis_init(&argc,&argv,           << receive id from master >>
                     err_handler,&id)      if (omis_init(&argc,&argv,
          != OMIS_OK) {                                   err_handler,&id)
         << Error handling >>                  != OMIS_OK) {
      }                                       << Error handling >>
      << send id to slaves >>              }
      ...                                  ...
    }                                    }
```

7.1.4 omis_finalize

```
Omis_status omis_finalize()
```

This function must be called by any tool before it exits in order to shut down the connection between the tool and the monitoring system in a well defined way. When **omis_finalize** is called, the monitoring systems deletes all conditional service requests defined by that tool. It also detaches from all objects to which it has been attached by that tool.

Whether or not **omis_finalize** also terminates the monitoring system and/or the monitored application depends on the specific implementation, but should be controllable by the arguments passed to **omis_init**.

7.1.5 omis_fd

```
int omis_fd()
```

omis_fd returns a file descriptor that can be used to block a tool process until a message from the monitoring system arrives, e.g. by passing the file descriptor to a **select** system call. If the function cannot execute correctly, it returns -1. A rationale and an example for this function is given in the next subsection.

Note that on non-UNIX systems the return type of this function may vary.

7.1.6 omis_handler

```
void omis_handler()
```

omis_handler polls the communication channel to the monitoring system for incoming messages and handles them properly. Although in an ideal world, this function (and also **omis_fd**) should not be visible at the interface, it is nevertheless necessary, since tools usually provide their own main loop waiting for input events and handling them. The message handling, including the invocation of call-back functions must somehow be included into this main loop, if the tool uses call-back functions for **omis_request**. If the tool always uses **NULL** for the **callback** argument of **omis_request**, there is no need to ever use **omis_fd** or **omis_handler**.

The following two examples show the proper use of **omis_fd** and **omis_handler**. The first example considers a simple command line oriented tool:

```
#include <sys/types.h>
#include <sys/select.h>
#include "omis.h"

int main(int argc, char **argv)
{
   fd_set fds;
   int mon_fd;
   char input_buf[80];

   /* Initialize connection to monitoring system */
   omis_init(&argc, &argv);

   /* Get OMIS file descriptor */
   mon_fd = omis_fd();

   do {
      /* Build file descriptor set containing the OMIS file
         descriptor and the stdin file descriptor */
      FD_ZERO(&fds);
      FD_SET(mon_fd,&fds);
      FD_SET(0,&fds);

      /* Block until there is some input */
```

```
        select(mon_fd+1, &fds, NULL, NULL, NULL);

        /* Read from stdin, if there is something to read */
        if (FD_ISSET(0,&fds)) {
           scanf("%s",input_buf);
           /* decode input_buf and execute commands */
           ...
        }

        /* Handle OMIS messages */
        omis_handler();

     } while (strcmp(input_buf,"quit"));

     omis_finalize();           /* shut down connection */
  }
```

The **select** system call and the **if** statement in this example ensure that OMIS messages can be handled whenever they arrive, since they prevent the tool process from blocking in the **scanf** function. Such a blocking would mean that replies cannot be passed to call-back functions until the blocking is released, which may cause a problem to the tool.

The next example shows how a tool based on X windows can use OMIS:

```
  void cb(XtPointer closure, int *source, XtInputId *id)
  {
     omis_handler();
  }

  int main(int argc, char **argv)
  {
     int omis_fd;
     XtAppContext app;

     /* initialize X window stuff */
     ... = XtAppInitialize(&app, ..., &argc, &argv, ...);

     omis_init(&argc, &argv);   /* initialize mon. system */

     /* Add OMIS file descriptor as an additional input source
        to the X window main loop. */
     XtAppAddInput(app, omis_fd(),
                   (XtPointer)XtInputReadMask,
                   (XtInputCallbackProc)cb, NULL);

     XtAppMainLoop(app);        /* X window main loop */
  }
```

Here, the OMIS file descriptor is passed as an additional input source to the X window system, that will call the **cb** function when there is some input to process. Note that since **XtApp-MainLoop** never returns, some X window call-back function must invoke **omis_finalize** before the tool exits.

7.2 Service Requests

7.2.1 String Syntax

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

```
request          ::=  [ event_definition ] ':' action_list
event_definition ::=  service_name '(' parameters ')'
action_list      ::=  unlocked_al  |  locked_al
locked_al        ::=  '{' unlocked_al '}'
unlocked_al      ::=  action  |  action [ ';' ] unlocked_al
action           ::=  service_name '(' parameters ')'

service_name     ::=  identifier
parameters       ::=  parameter_list  |  ε
parameter_list   ::=  parameter  |  parameter ',' parameter_list
parameter        ::=  integer  |  floating  |  string  |  token  |  binary
                        |  list  |  ev_ctx_param
ev_ctx_param     ::=  '$' identifier
token            ::=  identifier
list             ::=  '[' parameters ']'
```

The symbols *integer*, *floating*, *string*, and *identifier* represent integers, floating point numbers, quoted strings, and identifiers. The syntax of these elements follows the syntax of the corresponding elements in the C programming language.

The symbol *token* represents an abstract object identifier. *binary* is a binary data string consisting of ASCII coded number (the data length), immediately followed by a '#' character, immediately followed by the indicated number of Bytes in 8-Bit binary format.

list denotes an (untyped) list of entities; *ev_ctx_param* can be used in actions to refer to event context parameters. There is a set of standard parameters specified in Section 7.2.4 and a set of event specific parameters for each event service which is defined in Chapters 8 and 9.

The semantics of the different data types used in the request language is specified in Section 7.2.2. The semantics of the input *parameter_list* of a service depends on the concrete service and is specified in Chapters 8 and 9.

7.2.2 Data Types

This section specifies the data types that values in OMIS request and reply strings may have. Note that these are not data types of any programming language, and that the values of these data types only exist in string form. They have already shortly been introduced in Section 7.2.1.

OMIS defines five primitive data types: **integer**, **floating**, **string**, **binary**, and **token**. The first three types represent integer numbers, floating point numbers and quoted strings. Since the numbers only exist as strings, there is no need to specify the precision; it may in principle be arbitrary. However, except for a few target systems, integers will fit into 32 Bits and floating point numbers into a 64-Bit IEEE floating point format.

The **binary** data type has been included to support tools such as visualizers that have to acquire very large amounts of data, where a conversion into ASCII strings would cause an inacceptable performance problem. A value of this data type consists of an ASCII coded number (the data length), immediately followed by a '#' character, immediately followed by the indicated number of Bytes in 8-Bit binary format. Since the binary data may contain NUL characters, you have to be careful when operating on request or reply strings that contain binary data.

Note, however, that only services that may optionally be implemented in an OMIS compliant monitoring system use this data type.

The **token** data type is an abstract data type used to identify objects, e.g. processes, threads, or messages. It is specified in more detail in the next subsection.

Based on these primitive data types, OMIS defines one structured data type, the **list**. A list is an ordered collection of elements, which may have different primitive data types. However, most lists are homogeneous, i.e. contain only elements of a single primitive type.

The Token Data Type

The token data type is used in OMIS to provide a platform independent way of addressing objects being observed. Any object that can be observed or manipulated is represented by a token. OMIS defines seven basic classes of objects:

1. nodes
2. processes
3. threads
4. messages
5. message queues
6. user defined events
7. conditional service requests

Objects belonging to the first five classes (system objects) have a natural hierarchy, which is, however, not based on inheritance, but on the relation 'contains'. This hierarchy is shown in Fig. 7.1. Note that the relation between message queues and processes or threads depends on the target system, i.e. on whether the threads in a process have individual message queues or share a single queue. There may also be platforms, where message queues are first class objects (e.g. mailboxes). In this case, message queues are only related to nodes.

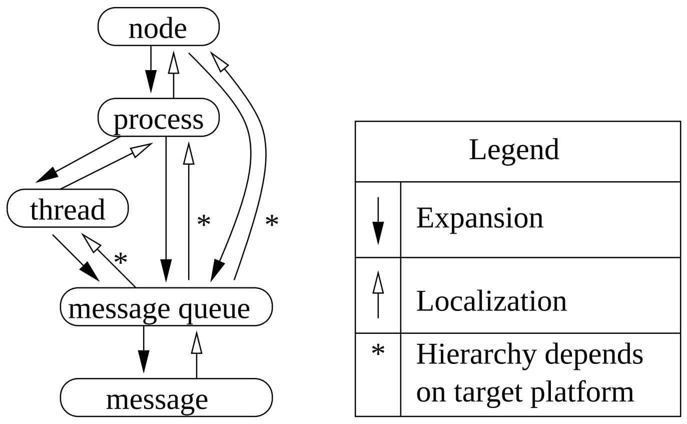

Figure 7.1: Object hierarchy in OMIS

According to this hierarchy, tokens and lists of tokens are implicitly converted to the token class a service works on. There are two types of conversions: localization and expansion. Localization converts a token a of class A into a token b of class B, where B is on an upper level in the hierarchy than A. This means that b refers to an object that contains the object addressed by a. Expansion converts a token a of class A into a set of tokens b of class B, where

B is on a lower level than A in the hierarchy. This set contains the tokens of all objects of class B contained in the object referred to by a to which the monitoring system is attached. Since expansion may result in a token being replaced by several new tokens, it is only performed for tokens contained in a list. Empty token lists are used as a universal object: if converted to a list of tokens of class B, an empty list is expanded to the set of tokens of all objects of class B to which the monitoring system is attached.

Since objects may be created and deleted dynamically, the point in time when the conversion is performed has an influence on the result. The following constraints are defined by OMIS:

1. Conversions required by action lists must be performed each time the action list is executed. For the action lists of conditional requests this means that conversion is not performed when the request is defined, but when the monitoring system has detected the occurrence of an event matching the request's event definition and executes the action list.

2. From a logical point of view, conversions required by the arguments of the event definition part of a conditional service request are performed each time the monitoring system detects the occurrence of some event and matches it againes the request's event definition[1].

3. All conversions done for the same execution of a request (including the event definition) must be consistent. So in the following example

   ```
   node_get_info([],1) proc_get_info([],1)
   ```

   if a new process on a new node is added to the monitored system between the execution of the two services, **proc_get_info** must not return information on that process, since otherwise the two conversions of the empty token list would be inconsistent. If, however, a new process on an already attached node is added, the expansion of the second token list may include that process.

   This could be achieved by performing all conversions in an atomic way at the beginning of the action list's execution, but can also be realized by a more efficient caching strategy: A conversion between two adjacent object classes in Fig. 7.1 is performed when it is needed for the first time. The result is then reused each time the same conversion step is required again.

   Note that if an object terminates during the execution of an action list, there is a chance for a service to receive a token of a nonexistent object. However, due to the chosen method of error handling (see Section 7.2.3) this does not induce a problem, since the service will only fail for that particular object.

The token data type is an abstract data type, so tools should not make any assumption on the structure and the contents of the token string. In particular, tools should not assume that the tokens are identical to the identifiers used within the parallel programming library. The only structural detail specified by OMIS is the encoding of the token class which is as follows: The class of a token is encoded in a prefix terminated by an underscore ('_'). The followig prefixes are defined:

[1]In an implementation, however, this matching and the conversion need not be performed explicitly, but can be achieved by a proper instrumentation of the monitored application.

nodes:	'n_'
processes:	'p_'
threads:	't_'
message queues:	'q_'
messages:	'm_'
user defined events:	'e_'
conditional service requests:	'c_'
undefined token:	'u_'

Extensions may define additional token classes. These tokens start with the prefix of the extension, followed by an underscore, a one-character token class specifier and another underscore. For example, a group token as defined in the PVM extension starts with 'pvm_g_'.

7.2.3 Semantics of Requests

In contrast to other monitoring interfaces, OMIS allows a service to be invoked on an object regardless of the state of that object. This means that e.g. the registers of a thread can be read without having to explicitly stop the thread in advance. On some target platforms, however, invoking a service on a running thread may include a temporary suspension of that thread, so explicitly suspending the thread may increase the performance when a larger number of services is requested.

The following subsections provide more details on the execution semantics of unconditional and conditional service requests.

Unconditional Requests

Each service contained in an action list will be executed on all the objects passed to the service as a token or token list, after proper conversion of the token(s). Since OMIS is used in a distributed environment, the executions of an action list's services on the different objects cannot be ordered totally, i.e. some of these executions may be concurrent or unordered. The following conditions hold for all action lists:

1. If **a1** and **a2** are services operating on lists of objects[2] and **l1** and **l2** are already converted lists of the proper token class, then the following ordering relation is required for each sequence **a1(l1) a2(l2)** in an action list:

   if **o1** is an object in **l1** and **o2** an object in **l2** and **o1** and **o2** are located on the same node, then **a1** is executed on **o1** before **a2** is executed on **o2**.

2. A semicolon (';') in an action list acts as a barrier, i.e. all actions left to the semicolon are completely executed before the execution of any action right to the semicolon is started.

3. Services do not have delayed side effects. This means that when a service has been executed, all the modifications it performs on the monitored system have been fully completed. This ensures, for instance, that services following a **thread_suspend** service will find the threads already suspended.

Tools should not assume any other ordering constraints to hold. Especially, a service may be executed on the objects passed to it in a token list in an order different from that indicated by the order of tokens in this list. Note that an object should not occur twice in a token list (which might also happen due to token expansion, see Section 7.2.2). In this situation, it is undefined whether the service will be executed once or twice for that object.

[2]Services that operate only on single objects may be viewed as working on a one-element object list for the purpose of this discussion

By default, action lists are interruptible, i.e. the execution of different action lists may be interleaved by the monitoring system. This allows action lists triggered by different events or sent by different tools to be executed concurrently in the parallel or distributed system. To enforce mutual exclusion of action lists when necessary, OMIS defines a locking mechanism, which is activated by enclosing an action list in braces ('{' and '}'). Locking an action list A ensures that on the subset of nodes that is touched by A no other action list is active while A is executed.

Conditional Requests

In contrast to unconditional service requests, the action list of a conditional request is not executed immediately, but each time when the monitoring system detects an event matching the event definition given in the request. The following items define the semantics of conditional requests:

1. When a conditional service request is passed to the monitoring system, it will immediately return a reply indicating whether or not the request could be installed properly. This reply will contain the token identifying the request.

2. Conditional service requests are disabled by default, i.e. they will be ignored by the monitoring system until they are explicitly enabled using the **csr_enable** service.

3. When a conditional request is deleted, or when it is enabled or disabled and the flag **DONT_RETURN_EN_DIS** is not set in **OMIS_request**, the monitoring system will send a reply indicating this change of state.

4. When the monitoring system detects some event and the conditional request is enabled, the event is matched against the event definition of the request. If the match succeeds, the following steps are performed:

   (a) The monitoring system takes measures to ensure that the object associated with the event will not undergo relevant state changes between the detection of the event and the completion of the action list execution. Usually, this is achieved by a temporary suspension of execution objects, i.e. threads, which is released upon completion of all action lists associated with that event. In order to permanently stop any threads, the action list must invoke the **thread_stop** service.

   A detailed specification, which threads will be suspended during the execution of action lists is given in the specification of event services for the different object classes.

   (b) Information on the detected event is stored in the event context parameters, which are specified in Section 7.2.4 and in the specifications of the individual event services.

   (c) The request's action list is executed as described in the previous subsection.

Error Handling

Since OMIS requests can result in a list of services being executed on a (possibly distributed) set of objects, there is a chance that some of these executions fail, while others succeed. In principle, there are three possible strategies to handle partial failure of a request:

1. The execution of a request is abandoned as soon as the first error is detected and a single error code is returned. This is the most simple strategy. However, the severe disadvantage is that in the case of an error, the system being observed is left in an unknown state.

2. Each request behaves as a transaction, i.e. it either is executed completely successfully or fails without having modified the state of the system being observed. Although from the users' point of view this is the most desirable strategy, its implementation would result in an overhead making the monitoring system useless.
3. Therefore, OMIS uses a mixed approach: when an error is detected while executing a single service on a single object, an error code and an error description is appended to the reply. Execution of this service on this object is then abandoned, trying to undo all manipulations that may already have been done to the object. If undoing is not possible in some cases, this must be indicated by setting a special bit (**OMIS_FATAL**) in the error code. The execution of the request is then continued. This means that a single service on a single object should behave as a transaction, whenever possible.

The used strategy avoids the overhead of global transactions, while ensuring that a tool issuing a request can always infere which modifications to the system's state have been performed and which have not (except in the case of fatal errors).

7.2.4 Event Context Parameters

The following list specifies the general event context parameters, i.e. parameters that are set on occurrence of any event. They may be used to pass information on the event to the action list of a conditional service request. In addition to the parameters defined below, individual event services may provide also other parameters, which are defined in the description of the event service.

token node; Contains the token of the node where the event took place.

token proc; The token of the process where the event took place. If the event cannot be attributed to a process, **proc** contains an undefined token.

token thread; The token of the thread where the event took place. If the event cannot be attributed to a thread, **thread** contains an undefined token.

floating time; This parameter contains a wall-clock time stamp indicating when the event has happened. An exact comparison between time stamps is only possible for events that occured on the same node, however, two successive events may have equal time stamps due to the limited clock resolution. Clocks on different nodes are at least to be synchronized at start-up-time up to a precision in the order of the message transfer time between nodes. The time stamps do not represent absolute time, i.e. the absolute time for which the time stamp is zero is not specified. The unit of the time stamp is seconds; the resolution of time stamps is platform dependent.

token csr; This parameter contains a token providing a self-reference to the conditional service request. It can be used to manipulate the service request from within its action list (e.g. to delete the request after it has been executed).

7.3 Service Replies

7.3.1 Reply Structure

The reply returned by **omis_request** (either as a function result of by passing it to the callback function) is a data structure conforming to the following C type declaration:

```
/*
** Status values
*/
typedef int Omis_status;

#define OMIS_OK                    0  /* no error */

#define OMIS_CSR_DEFINED           2  /* cond. request has been defined */
#define OMIS_CSR_ENABLED           4  /* cond. request has been enabled */
#define OMIS_CSR_DISABLED          6  /* cond. request has been disabled */
#define OMIS_CSR_DELETED           8  /* cond. request has been deleted */
#define OMIS_CSR_TRIGGERED        10  /* cond. request has been triggered */

#define OMIS_FIRST_ERROR          16  /* lowest status code for error */

#define OMIS_SYNTAX_ERROR         16  /* syntax error in request string */
#define OMIS_UNKNOWN_SERVICE      18  /* service name is unknown */
#define OMIS_UNSUPPORTED_SERVICE  20  /* service is not supported */
#define OMIS_UNKNOWN_ECP          22  /* event context par. is unknown */
#define OMIS_UNKNOWN_OBJECT       24  /* object is unknown/not existent */
#define OMIS_TYPE_MISMATCH        26  /* type mismatch in parameters */
#define OMIS_PARAMETER_ERROR      28  /* illegal parameter value */
#define OMIS_OS_ERROR             30  /* error from operating system */
#define OMIS_NO_PERMISSION        32  /* operation not permitted */
#define OMIS_NO_MEMORY            34  /* out of memory */
#define OMIS_INTERNAL_ERROR       36  /* internal error in mon. sys. */

#define OMIS_UNSPECIFIED_ERROR  1000  /* generic error code */

#define OMIS_FATAL                 1  /* object state modified */

/*
** Result for a single object or identical results for a set of objects
*/
typedef struct {
    char *obj_list;      /* List of objects where result belong to */
    Omis_status  status; /* Status for this set of objects */
    char *result;        /* Result for this set of objects */
} Omis_object_result;

/*
** Result of a single service
*/
typedef Omis_object_result *Omis_service_result;

/*
** Full reply of a service request
*/
typedef Omis_service_result *Omis_reply;
```

The reply is of type **Omis_reply**, which is a pointer to a **NULL**-terminated array, whose element type is **Omis_service_result**. The first element (with index 0) of this array is a reply for the request as a whole where the request is regarded as a passive object. This element is used for the following purposes:

1. It returns errors that occur during any steps needed to prepare the execution of the request's action list. This includes any syntactical and semantical checks done when the request is defined.
2. For conditional requests, it contains the conditional request token needed for the services manipulating the request.
3. It indicates any change in state of conditional service requests.

The other elements of a reply contain the results of those parts of the request that have been executed. The element with index 1 points to the results of the first action in the request's action list, element 2 to the results of the second one (if any) and so on.

There are three different types of replies, that can be distinguished by the status field of the **Omis_object_result** structure pointed to by the reply's first element.

1. If the status field contains a value above or equal to **OMIS_FIRST_ERROR**, an error has been detected during checks made prior to the request's execution. In this case, the reply has only one non-**NULL** element.
2. If the status field is equal to either **OMIS_CSR_DEFINED**, **OMIS_CSR_ENABLED**, **OMIS_CSR_DISABLED** or **OMIS_CSR_DELETED**, the reply contains the results of the definition, enabling, disabling or deletion of a conditional service request. In this case, the reply has a second element that contains the status message generated by the event service. For **OMIS_CSR_DEFINED**, the **result** element of the first **Omis_object_result** structure will contain the conditional request's token. However, if the event service failed and the conditional request therefore has not been stored by the monitoring system, the **result** element will be **NULL**.

   Note that **OMIS_CSR_ENABLED** and **OMIS_CSR_DISABLED** will be generated both when the request is explicitly enabled or disabled, and when the set of actually monitored objects changes by attaching to or detaching from objects.
3. If the status is **OMIS_OK** or **OMIS_CSR_TRIGGERED**, the following elements of the reply contain the results of the execution of all actions in the action list of the unconditional or conditional request.

Since the results of actions may consist of several sub-results for different objects, the type **Omis_service_result** is a pointer to an array of **Omis_object_result** structures. The end of this array is marked by an entry with **obj_list** == **NULL**; the other fields of this end marker don't have any meaning. The principal semantics of the **Omis_object_result** structures is:

- **obj_list** points to a string containig a comma-separated list of object tokens that specifies the objects this part of the result belongs to.
- **status** is the status for the objects defined in **obj_list**. If the bit **OMIS_FATAL** is set in an error status, the state of the objects in **obj_list** has been changed in an inconsistent way due to the error. If an error message is returned where the flag is not reset, the objects' states have not been changed by the failing service.

- **result** points to a string containing the (identical) result of an action's execution for the objects in **obj_list**, if **status** doesn't indicate an error. Otherwise, **result** points to a string containing a more detailed error description that may be presented to the user. If there is no result or error description, **result** may be **NULL**.

If the reply does not belong to a service operating on objects (as it is the case for element 0 of the **Omis_reply** array), **obj_list** must point to an empty string and the rest of the above description holds analogously. For the other cases, **Omis_object_result** allows to unify identical results for different objects in order to save memory and communication time. This is important especially for the status replies which are usually the same for all objects. Having one **status** field for each single object a service worked on would result in a huge overhead. However each implementation of an OMIS compliant monitoring system is free to decide whether or not this unification is done. Since OMIS uses a hierarchical approach to identify objects, the **obj_list**s may in principle contain objects of different granularity. For instance, if a service is invoked for all threads in the observed system by specifying an empty token list as its parameter, and the result is the same for each thread, there are several possibilities for the result structure:

- Only a single entry with **obj_list** pointing to an empty string.
- Several entries with **obj_list** pointing to a single node token.
- Only a single entry with **obj_list** being the list of tokens of all threads in the observed system.
- ...

In order to not overly complicate the tools using OMIS, the scheme is restricted according to the following description:

1. In **Omis_object_result** structures returning a real result (i.e. those having **status** == **OMIS_OK** and either belong to an information service or a manipulation service with a non-void return type), **obj_list** must point to a non-empty list of tokens of the type the service works on. This ensures that any information returned is always accompanied with an explicit list of the objects it refers to.
2. In a reply generated when a conditional request is triggered successfully, the first structure **Omis_object_result** in the result has **status** == **OMIS_TRIGGERED**, and **obj_list** points to a one-element list defining the object where the event occured.
3. In all other cases, the **obj_list**s may be a partial expansion of the object list given as an argument of the service. Furthermore, the last **Omis_object_result** structure (prior to the end marker) may have **status** == **OMIS_OK** with **obj_list** pointing to an empty string and **result** == **NULL**), indicating that for all of the objects not mentioned in previous entries the service has been executed successfully.

Note that except for case 3, objects specified explicitely in the parameter list of a service will never be summarized by using tokens of containing objects in the **obj_list**.

The exact syntax for the strings pointed to by **obj_list** and **result** is specified in Chapter 7.2, the semantics of the **result** strings depends on the service and is specified in Chapters 8 and 9.

7.3.2 String Syntax

The syntax of the strings pointed to by the **obj_list** and **result** fields of an **Omis_object_result** strucure (see Section 7.3.1) is as follows:

obj_list	::=	*token_list* | ϵ
token_list	::=	*token* | *token* ',' *token_list*
result	::=	*parameter_list* | *error_description*

The symbol *error_description* denotes an arbitrary character sequence, which can provide a detailed description of an error situation.

7.3.3 Reply Examples

Fig. 7.2 to Fig. 7.4 illustrate different forms of replies. For the sake of clarity, the illustrations don't show pointers to strings. Instead, the strings are directly shown in the pointer fields in the representation used in the C language.

First, consider an unconditional request of the form

```
: proc_get_info([], ...) thread_manipulate([], ...)
```

where **proc_get_info** is an information service for processes, while **thread_manipulate** is a manipulation service for threads. When no errors are encountered during the execution of this request, a possible reply may look as those shown in Fig. 7.2. Note that the empty token list passed to **proc_get_info** must be expanded in the reply, while the empty token list passed to **thread_manipulate** need not. The reason for this is the fact that the first service returns a real result, i.e. has a non-**void** return type, whereas the latter one has a return type of **void** and therefore only returns a status value. Fig. 7.3 shows how a reply may look like when errors occur.

Now, consider a conditional request:

```
proc_has_done_something([], ...) : node_get_name([$node])
```

When this request is passed to the monitoring system, three different kinds of replies may be received as shown in Fig. 7.4. The reply generated when the event is detected indicates the object (i.e. process in this example) where the event occurred in the **obj_list** component of the reply's first **Omis_service_result** structure.

When the tool now attaches to a new process, say **p_32**, a reply as shown in Fig. 7.5 will be generated, except for the case when the **OMIS_DONT_RETURN_EN_DIS** flag has been set when defining the above conditional request. If the preparations necessary to monitor the event in the new process fails, this reply contains a corresponding status and an error message.

7.4 Description Method for Services

Throughout the service descriptions in Chapters 8 and 9, we will use ANSI-C-like prototypes to define the input parameters and the result strings, 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*, *binary* and *token*, we will use the type-identifier *any* which stands for any of these types. To define more complex parameters, we will use **typedef**'s and/or C-**struct**'s[3]. Lists of values are specified by the type name followed by a '*', denoting zero or more repetitions of that type. 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 '*') corresponds to several values of its component type, that are again separated by commas, but are enclosed in brackets ('[' and ']').

[3]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.

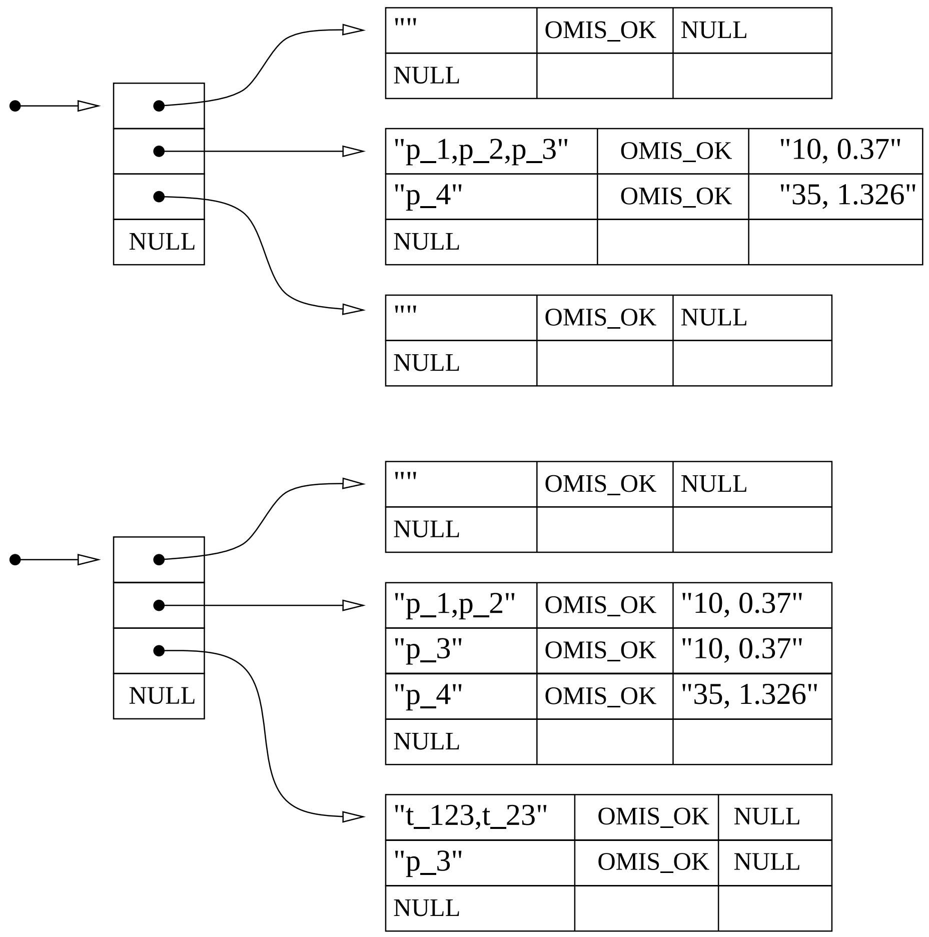

Figure 7.2: Two possible replies for an unconditional request without errors

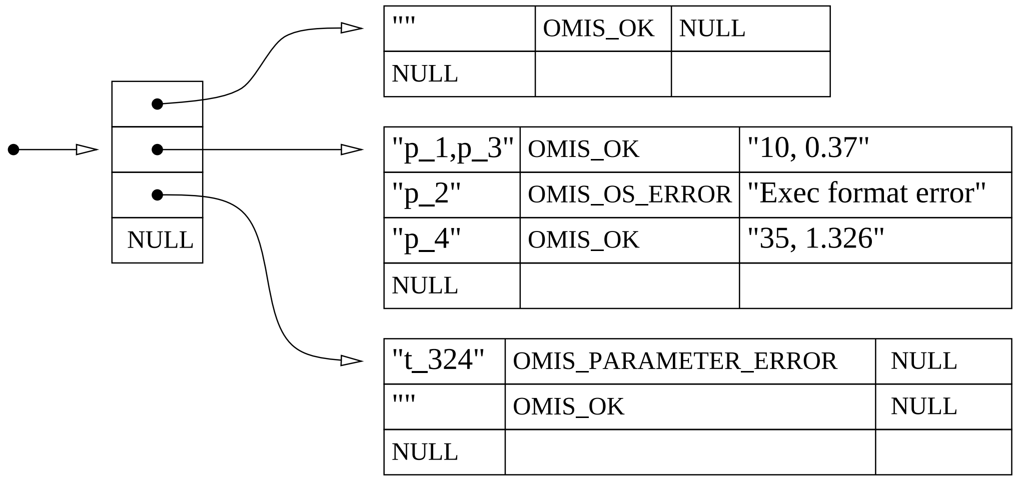

Figure 7.3: Reply for an unconditional request with errors

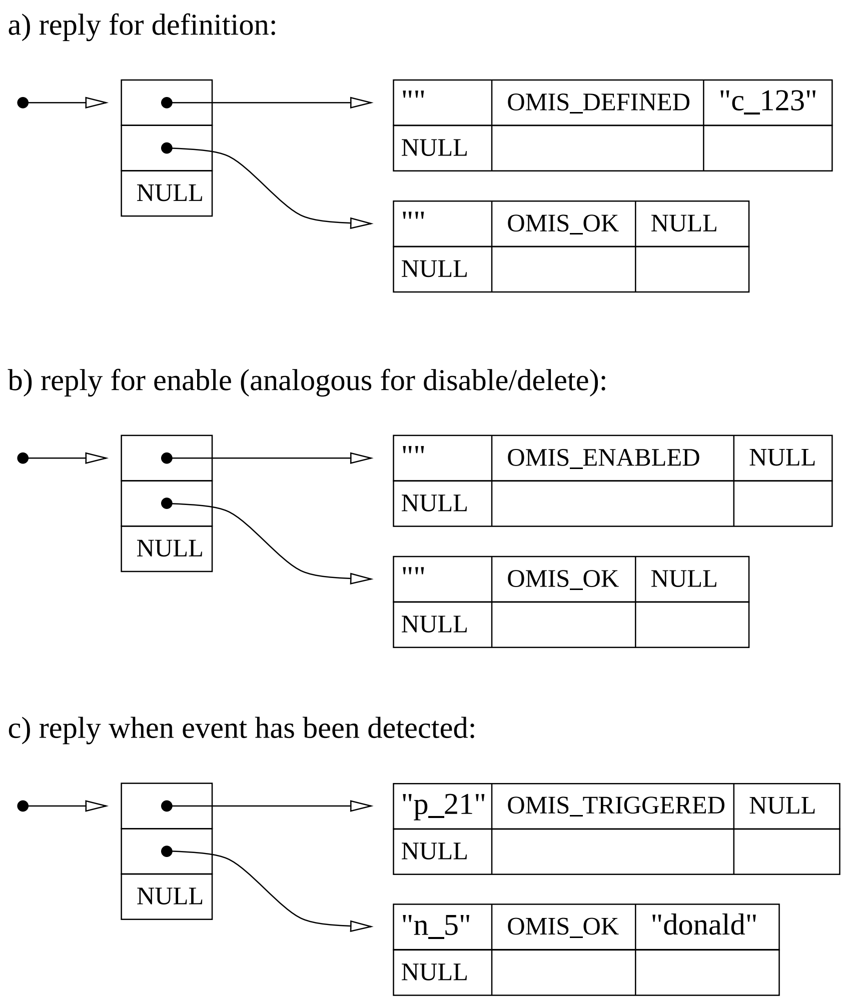

Figure 7.4: Replies for a conditional request

The return type given in the prototype specifies the result of the service for a single object, i.e. the information contained in the **result** field of a single **Omis_object_result** structure in the service's reply.

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

```
struct {
    string name;
    integer state;
    integer priority;
    floating cpu_time;
}
proc_get_info (token* proc_list, integer flags);
```

This says that **proc_get_info** has two input parameters, the first one is a (probably empty) list of tokens, the second one is a single integer. For each object, i.e. process the service operates on, a string consisting of four comma-separated elements is returned, where element 1 is a quoted string, elements 2 and 3 are integers and elements 4 is a floating point number. Therefore, a correct request for this service could be:

```
proc_get_info([p_31,p_45,p_54], 5)
```

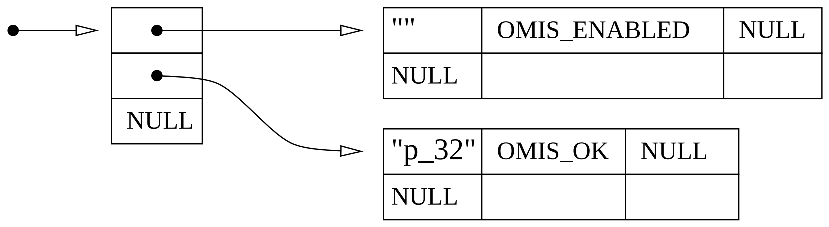

Figure 7.5: Reply for a conditional request when new object is monitored

A valid reply could look as shown in Fig. 7.6.

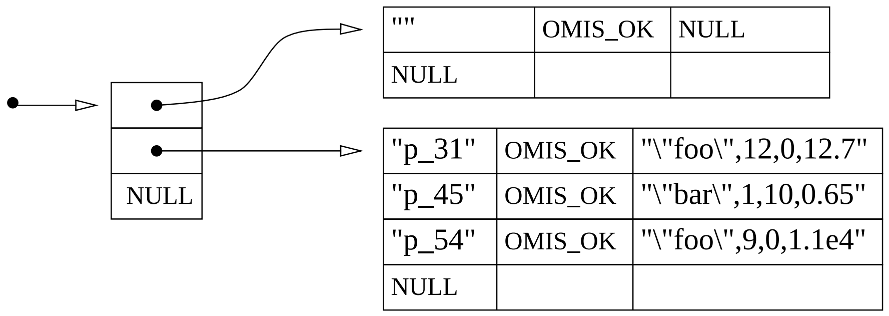

Figure 7.6: Reply example

A few synchronous services return results 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 then determines which components will actually be present. The real **proc_get_info** service is of this type, i.e. the definition of this service really looks more like:

```
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
}
proc_get_info (token* proc_list, integer flags);
```

This means that all components of **proc_get_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

```
proc_get_info([p_31,p_45,p_54], 5)
```

could be (since bits 0 and 2 are set in **flags**) as in Fig. 7.7.

Notice that due to the filtering, the replies for the processes with tokens **p_31** and **p_54** are now the same, so they have been unified to a single reply string. However, this behavior is not required, i.e. it is also allowed that separate (identical) replies are generated.

Event services always have only a status value as their normal result, which is returned when a conditional service is defined, enabled, disabled or deleted. However, event services also have an additional type of results, namely the parameters that can be accessed by the actions when a matching event is detected. To define the types of these results, we use a notation that looks like this:

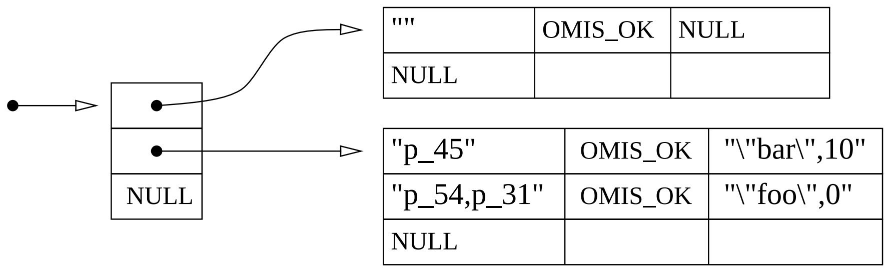

Figure 7.7: Reply example

```
void proc_has_done_something(token* proc_list, integer param)
--> struct {
   integer first_result;
   string  second_result;
   integer third_result;
}
```

The type of data returned as the normal result of the event service is given in the C-like prototype (it is always **void** since event services return only a status). The so called *event context parameters*, which contain information on a detected event, are defined as a **struct** after the `-->` symbol. These event context parameters can be passed to the actions in the request's *action_list* by specifying $*event_context_parameter_name* as an input parameter for the action. Note that event context parameters are never defined as being optional. However, an implementation may choose not to compute an event context parameter, if it is not used in the action list.

Chapter 8

Specification of the 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 for writing the events to a peripheral as a distributed tool extension, 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.

8.1 System Objects

System objects are those objects in the monitored system that do not belong to the monitor itself. Currently, we distinguish between

- nodes (Section 8.1.1),
- processes (Section 8.1.2),
- threads (Section 8.1.3), and
- messages and message queues (Section 8.1.4).

The following section specify the services for these classes of objects. In these sections we distinguish between required services that have to be provided by any implementation and optional services which need not be implemented by an OMIS compliant monitoring system. The services are marked with (R) and (O), respectively. In addition, there are some services that are marked with a (P), denoting that they are partially required, i.e. only some of the services functionality is required.

Additional system objects and services are defined in an extension for the PVM programming model (see Chapter 9).

8.1.1 Nodes

OMIS is intended to be usable for a wide range of hardware platforms. The coarse grain model of the monitored hardware platform is a set of nodes interconnected by some network. However, a node need not be a single processor. A node may also be a multiprocessor system. The criterion to draw the borderline between processors and nodes is that for a user or a programmer nodes are distinguishable from each other, while processors on the same node are not. In other words: nodes are those components of the hardware platform that have a single system image. This implies that processors on the same node have a (at least virtually) shared memory, but the reverse implication is not necessarily true.

Manipulation Services

The services in this section of course do not manipulate the hardware, but change the monitored hardware system by adding or removing nodes to be monitored. These services not only allow an incremental start-up of the monitoring system, but also provide support for programming environments that allow applications to extend the set of nodes they are currently executing on. An application thread may extend the set of nodes either explicitly, as in PVM, where it must use a library call **pvm_addhosts**, or implicitly by creating a process on a node not used before. In any case, this event can be monitored using the **thread_adds_node** service. By using the services specified below, the monitoring system can be programmed to automatically extend itself to the new nodes.

Likewise, an application's node set may also shrink due to an explicit request from an application thread (e.g. by calling **pvm_delhosts**). This event can be monitored using the **thread_removes_node** service. An action associated with that event may detach the monitoring system from that node.

1. **node_attach** .... attach to a node. (O)

   ```
   void
   node_attach(token* node_list)
   ```

   Calling this service results in the monitoring system attaching itself to the nodes specified in **node_list**. The node token(s) required as an argument may be obtained from the event context parameters of **node_has_been_added**, from a service in an extension (e.g. **pvm_vm_get_nodelist**), or from some other OMIS based tool, using a communication mechanism outside the scope of OMIS. Note that there is also a service **node_attach2** to attach to a node specified by its name.

   This is the only service that may legally receive a token of an unattached node as its parameter. If the node is already attached, the service does nothing.

2. **node_attach2** .... attach to a new node. (O)

   ```
   token
   node_attach2(string node_name)
   ```

   Calling this service results in the monitoring system attaching itself to a new node specified by its name. The exact meaning of **node_name**'s contents is platform dependent. Usually, the string will either contain an internet address (for workstation clusters) or a node number (for parallel computers). If the node is already attached, the service does nothing.

   Note that since the service is a constructor in the sense of object oriented programming, it does not operate on node tokens although it is a node service.

3. **node_detach** .... detach from a node. (O)

   ```
   void
   node_detach(token* node_list)
   ```

Calling this service results in the monitoring system detaching itself from the nodes specified in **node_list** and in turn also from all processes located on these nodes. Note, that conditional requests defined for the detached objects will not be deleted automatically – the events simply will no longer be detected. Any subsequent service request (other than **node_attach** or **node_attach2**) for a detached node will result in an error.

Information Services

Currently, there is only a single service that returns static and dynamic information on the nodes to which the monitoring system is attached:

1. **node_get_info** .... Return information on a node. (P)

```
typedef struct {           // Information on network link
  string   net_type;       // Type of network interface, e.g. Ethernet,
                           // ATM
  string   net_ipaddr;     // IP address of the node in this network
  integer  net_bandwidth;  // Vendor defined total network bandwidth
                           // (in KBytes/s)
  floating net_bench;      // Relative network performance measured
                           // by some implementation specific
                           // loopback benchmark
} netinfo;

typedef struct {
                             // Static node information

                             // These components are present if bit 0
                             // is set in flags:
                             //
  [string   name;]           // Name of this node. Usually, this is the
                             // host name (R)

                             // These components are present if bit 1
                             // is set in flags:
                             //
  [string   os_name;]        // Name of node's operating system (R)
  [string   os_version;]     // OS version (R)
  [string   os_release;]     // OS release (R)
  [string   os_nodename;]    // Host name of this node (R)
  [integer  os_boottime;]    // OS boot time in seconds since Jan. 1st,
                             // 1970, 0:00 (O)

                             // These components are present if bit 2
                             // is set in flags:
                             //
  [string   cpu_arch;]       // CPU architecture (R)
  [integer  cpu_num;]        // Number of installed CPUs on this
                             // node (R)
  [integer  cpu_maxproc;]    // Max. number of processes on this node
                             // (O)
  [integer  cpu_clock;]       // CPU clock frequency (in MHz) (O)
  [floating cpu_intbench;]    // Relative integer performance measured
                              // by some implementation specific
                              // benchmark (O)
  [floating cpu_fpbench;]     // Relative floating point performance
                              // measured by some implementation
                              // specific benchmark (O)

                              // These components are present if bit 3
                              // is set in flags:
                              //
  [integer  mem_numpages;]    // Number of physical memory pages (O)
  [integer  mem_pagesize;]    // Size of a page in Bytes (O)
  [floating mem_bench;]       // Relative memory performance measured by
                              // some implementation specific
                              // benchmark (O)

                              // These components are present if bit 4
                              // is set in flags:
                              //
  [integer  dsk_num;]         // Number of local disks (O)
  [integer  dsk_size;]        // Total size of disk space (in pages of
                              // size mem_pagesize) (O)
  [integer  dsk_tmpsize;]     // Total size of tmp disk space (in
                              // pages) (O)
  [integer  dsk_swapsize;]    // Total size of swap space (in pages) (O)
  [floating dsk_bench;]       // Relative disk performance measured by
                              // some implementation specific
                              // benchmark (O)

                              // These components are present if bit 5
                              // is set in flags:
                              //
  [integer  net_numlinks;]    // Number of high-speed network links (O)
  [netinfo* net_info;]        // Additional information for each network
                              // link. The number of entries in this
                              // list equals net_numlinks (O)

                              // These components are present if bit 6
                              // is set in flags:
                              //
  [integer  usr_maxlogins;]   // Max. number of user logins (O)
} Node_static_info;
typedef struct {
                              // Dynamic node information

                              // These components are present if bit 7
                              // is set in flags:
                              //
  [integer  os_ctxtswitch;]   // Number of context switches per
```

```
                             // second (O)
[integer  os_execs;]         // Number of calls to 'exec' per
                             // second (O)
[integer  os_syscalls;]      // Number of system calls per second (O)

                             // These components are present if bit 8
                             // is set in flags:
                             //
[integer  cpu_rql;]          // Length of process run queue (R)
[integer  cpu_dwj;]          // Number of processes waiting for disk
                             // I/O (O)
[integer  cpu_pwj;]          // Number of processes in page wait (O)
[integer  cpu_slj;]          // Number of processes sleeping in
                             // core (O)
[integer  cpu_swj;]          // Number of runnable processes swapped
                             // out (O)
[floating cpu_rql1;]         // CPU load (run queue length) avaraged
                             // over one minute (R)
[floating cpu_rql5;]         // CPU load avaraged over 5 minutes (R)
[floating cpu_rql15;]        // CPU load avaraged over 15 minutes (R)

                             // These components are present if bit 9
                             // is set in flags:
                             //
[integer  mem_freepages;]    // Number of free physical memory
                             // pages (O)
[integer  mem_usedpages;]    // Number of used physical memory
                             // pages (O)
[integer  mem_freeswap;]     // Number of free pages in swap area (O)

                             // These components are present if bit 10
                             // is set in flags:
                             //
[integer  vm_swap;]          // Total number of process swaps per
                             // second (O)
[integer  vm_swapin;]        // Number of processes swapped in per
                             // second (O)
[integer  vm_swapout;]       // Number of processes swapped out per
                             // second (O)
[integer  vm_page;]          // Total number of paging per second (O)
[integer  vm_pagein;]        // Number of pages paged in per second (O)
[integer  vm_pageout;]       // Number of pages paged out per
                             // second (O)

                             // These components are present if bit 11
                             // is set in flags:
                             //
[integer  dsk_rawrd;]        // Number of physical reads to raw disk
                             // device per second (O)
[integer  dsk_rawwr;]        // Number of physical writes to raw disk
```

```
                                  // device per second (0)
    [integer  dsk_nfsrd;]         // Number of NFS block reads per
                                  // second (0)
    [integer  dsk_nfswr;]         // Number of NFS block writes per
                                  // second (0)
    [integer  dsk_sysrd;]         // Number of 'read' system calls per
                                  // second (0)
    [integer  dsk_syswr;]         // Number of 'write' system calls per
                                  // second (0)

                                  // These components are present if bit 12
                                  // is set in flags:
                                  //
    [integer  net_lpkt;]          // Number of local packets per second (0)
    [integer  net_fpkt;]          // Number of fast packets (ETH, ATM, ...)
                                  // per second (0)
    [integer  net_fpktrcv;]       // Number of fast packets received per
                                  // second (0)
    [integer  net_fpktsnd;]       // Number of fast packets sent per
                                  // second (0)
    [integer  net_spkt;]          // Number of slow packets (Modem, ...)
                                  // per second (0)
    [integer  net_spktrcv;]       // Number of slow packets received per
                                  // second (0)
    [integer  net_spktsnd;]       // Number of slow packets sent per
                                  // second (0)

                                  // These components are present if bit 13
                                  // is set in flags:
                                  //
    [integer  usr_numlocal;]      // Number of local user logins (0)
    [integer  usr_localact;]      // Is a local user active? (0)
    [integer  usr_numremote;]     // Number of remote user logins (0)
    [integer  usr_remoteact;]     // Is a remote user active? (0)
  } Node_dynamic_info;
  struct {
     Node_static_info statinfo;
     Node_dynamic_info dyninfo;
  }
  node_get_info(token* node_list, integer flags)
```

Detailed information on nodes is provided by the **node_get_info** service. As with the other information services, the bit-vector **flags** defines which kind of information has to be retrieved. By calling **node_get_info([],0)**, the tokens for all nodes currently observed by the monitoring system can be retrieved.

The values contained in **Node_dynamic_info**, which are measured in some units per second may be averaged over a couple of seconds. The exact interval used for averaging is implementation dependent, but should be between one and fifteen seconds.

For those components labelled as optional, an implementation should return the value -1 or an empty string, if the information is not available on the specific target system.

8.1.2 Processes

Since OMIS aims at defining a very general monitoring interface, it is based on a multithreaded execution model. In this model, threads are the entities actually executing code, while processes only serve as a container for threads. The term 'container' indicates the twofold role of a process: It defines the execution environment, e.g. the address space, for its threads, but it may also be viewed as an active component defined by the union of all its threads. A process cannot exist alone; there is always at least one thread in the process. So a process creation always implies also a thread creation. Threads executing in the same process share a common address space.

This model is applicable to a wide range of platforms. Currently, three different types of platforms can be distinguished:

1. Platforms without support for multithreading: On these systems, each process contains exactly one thread. The token conversion rules defined in Section 7.2.2 allow process and thread tokens to be used interchangeably, so a tool does not have to distinguish between threads and processes.
2. Multithreaded, multiprocess platforms: They exactly fit into the process model of OMIS.
3. Multithreaded platforms without the notion of a process: There are some platforms that only support threads, but don't provide virtual address spaces (e.g. the Parsytec computers running the PARIX operating system). On these platforms, a process in the sense of OMIS is defined by all threads created due to the invocation of a single executable program, and the memory areas allocated for that program. The service **proc_get_loader_info** has been provided especially for these platforms.

Manipulation Services

The following services are provided to manipulate the behavior of processes.

1. **proc_create** .... create a new process on a node. (O)

   ```
   token
   proc_create(token* node_list, string exec, string* argv,
               string* envp, string* io)
   ```

   The **proc_create** service creates a new process (e.g. a PVM task) on each node in **node_list** and returns its process token. The monitoring system automatically attaches to the process, which is created in a stopped state, so you have to use the **thread_continue** service to start it. The parameter **exec** defines the executable's path name, **argv** is the vector of command line arguments (not including the name of the executable again).

   envp is a list of strings defining the environment seen by the new process. Each string consists of the name of an environment variable, immediately followed by a '=' sign and the value of that variable. If **envp** is the empty list, the process inherits the environment from the monitoring system.

   io is a list of strings defining file names used to redirect the standard IO-streams of the new process. The first element in **io** defines the file to be used for the process' first IO-stream (stdin for C/UNIX), the second element for the next IO-stream and so on. An empty string denotes that the corresponding stream will not be redirected, i.e. will be inherited from the monitoring system. If **io** is the empty list, no IO-stream will be redirected.

   This service is optional, since some programming libraries (e.g. MPI-1) do not allow processes to be created dynamically. Note that since the service is a constructor in the sense of object oriented programming, it operates on node tokens although it is a process service.

2. **proc_attach** .... attach to a process. (O)

```
void
proc_attach(token* proc_list)
```

The **proc_attach** service attaches the monitoring system to all processes specified in **proc_list**. This may also include attaching the monitoring system to new nodes. The process token(s) required as an argument may be obtained from the event context parameters of **proc_has_been_created**, from a service in an extension (e.g. **pvm_vm_get_proclist**), or from some other OMIS based tool, using a communication mechanism outside the scope of OMIS. Note that there is also a service **proc_attach3** to attach to a process specified by its local process identifier on a specific node.

This is the only service that may legally receive a token of an unattached process as its parameter. If the process is already attached, the service does nothing.

Attaching to a process implies attaching to all existing and future threads of that process. The rationale for this behavior is as follows: Thread creation should be an extremely lightweight operation. Forcing an explicit attach operation for each thread would result in an unacceptable overhead. In addition, in virtually all multithreaded systems, a standard implementation of a monitoring system will result in an automatic attachment to all threads in an attached process, so defining that new threads are unattached by default would put a huge burden on OMIS implementations. Finally, threads share a common execution environment, while processes do not, so it may be even more natural to regard them as a unit for the purpose of monitoring, although the different handling of processes and threads may seem to be inconsistent.

3. **proc_attach3** .... attach to a process on a node. (O)

```
token
proc_attach3(token* node_list, integer pid, string exec)
```

The **proc_attach3** service attaches the monitoring system to the process given by its local identifier **pid** on each node in **node_list**. Note that for operating systems like UNIX specifying a node list with more than one node may be not useful, nevertheless, a token list is used as a parameter for conistency reasons.

exec specifies the path to the executable of the process to be attached. This information is needed to access symbol tables or linker information of the process to be attached. When an empty string is passed for **exec**, the monitoring system will try to determine the path by itself. Depending on the concrete plattform, this may or may not be possible.

Like **proc_create**, this service is a constructor in the sense of object oriented programming, so it operates on node tokens although it is a process service.

4. **proc_detach** .... detach from a process. (O)

```
void
proc_detach(token* proc_list)
```

The **proc_detach** service detaches the monitoring system from the processes specified in **proc_list** and all of its threads. When a process is detached, it is no longer included in the set of monitored processes and the monitoring system removes any instrumentation, i.e. any modifications done to the process in order to detect events. Note however, that conditional requests defined for the detached processes will not be deleted automatically — the events simply will no longer be detected. Any subsequent service request (other than **proc_attach** or **proc_attach3**) for a detached process will result in an error.

5. **proc_send_signal** .... send a signal to a process. (O)

   ```
   void proc_send_signal(token* proc_list, integer sig)
   ```

   Sends the signal **sig** to all processes in **proc_list**. According to the Posix semantics of signals, the signal may be delivered to an arbitrary thread in each of these processes.

6. **proc_set_priority** .... change priority of a process. (O)

   ```
   void
   proc_set_priority(token* proc_list, integer val)
   ```

   Changes the scheduling priority of the processes in **proc_list** to **val**. On systems where there is more than one scheduler, the service refers to the priority defined by the parallel programming library, which also defines the range and semantics of **val**.

   The service usually should not allow to raise a process' priority above its initial one, however, if the priority has been lowered using **proc_set_priority** it should allow to raise it to the initial one again. However, the exact behavior can be platform dependent.

7. **proc_write_memory** .... write into the memory of a process. (R)

   ```
   void
   proc_write_memory(token* proc_list, integer addr,
                     integer blocklength, integer stride, integer* val)
   ```

   Writes the contents of the integer list **val** into the memory of all processes defined by **proc_list**. **val** contains the raw data to be written, i.e. a list of Bytes; the monitoring system does not perform any processing (e.g. byte swapping) of this data. The data will be written in contiguous blocks of Byte-size **blocklength**, where the address of the first block is **addr** and **stride** is the separation between the start of two subsequent blocks in Bytes. Thus, the first **blocklength** Bytes in **val** will be written to addresses **addr** ... **addr**+**blocklength**−1, the next **blocklength** Bytes to addresses **addr**+**stride** ... **addr**+**stride**+**blocklength**−1 and so on. **stride** must not be smaller than **blocklength**.

8. **proc_write_memory_bin** .... write into the memory of a process. (O)

   ```
   void
   proc_write_memory_bin(token* proc_list, integer addr,
                         integer blocklength, integer stride, binary val)
   ```

   This service is identical to **proc_write_memory** with the exception that it expects the data to be written in a binary format.

9. **proc_migrate** .... migrate a process to another node. (O)

   ```
   void
   proc_migrate(token* proc_list, token node)
   ```

This service migrates all of the processes defined by **proc_list** to an other node, which is given by its node token.

10. **proc_checkpoint** .... save checkpoint of a process. (O)

```
void
proc_checkpoint(token* proc_list, string name)
```

Creates a consistent checkpoint of all processes in **proc_list** and saves it to disk. **name** is an identifier for this checkpoint used for a later restore. The format of the checkpoint file(s) and the way they are stored on disk(s) is implementation dependent.

11. **proc_restore** .... restore a process from a checkpoint. (O)

```
struct {
    token* procs;        // the process tokens of the restored processes
    integer num_procs;   // number of processes restored
}
proc_restore(string name)
```

Restores a set of processes from a previously saved checkpoint with the identifier **name**. The processes will be restored on their original hosts, but they will get new process tokens. The list of these tokens is returned as the result of the service. The monitoring system will automatically attach to all of these processes..

Information Services

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

1. **proc_get_info** .... get information on processes. (P)

```
typedef struct {                // Static info. Independent of time.
                                //
                                // Required components:
                                //
   [integer global_id;]         // Global id of this process, as
                                // defined by the programming library
                                // used, e.g. the PVM task id
                                // present if bit  0 is set in flags
   [string* argv;]              // Argument vector,
                                // i.e. pathname and parameters
                                // present if bit  1 is set in flags
                                //
                                // Optional components:
                                //
   [integer uid;]               // ID of the user owning this process
                                // present if bit  2 is set in flags
   [integer gid;]               // ID of the group owning this process
                                // present if bit  3 is set in flags
   [string* user_argv;]         // Argument vector provided by the
                                // user at start of this process. It may
                                // differ from argv if the programming
```

```
                              // library removes or adds options when
                              // starting a process.
                              // present if bit  4 is set in flags
   [string* envp;]            // Environment of this process.
                              // See proc_create for a description.
                              // present if bit  5 is set in flags
   [token parent;]            // Token of the process' parent process
                              // present if bit  6 is set in flags
   [token message_queue;]     // Token of the process' (logical)
                              // message queue if the process' threads
                              // share a single queue, undefined token
                              // otherwise (see message services). See
                              // the section on message queues for a
                              // discussion.
                              // present if bit  7 is set in flags
} Proc_static_info;

typedef struct {              // Dynamic info. Changes over time.
                              // All times are returned in seconds,
                              // all sizes in Bytes.
                              //
                              // Required components:
                              //
   [token node;]              // Token of the node where this
                              // process is located.
                              // Note that this may be dynamic due
                              // to process migration.
                              // present if bit  8 is set in flags
   [integer local_id;]        // Local id of this process,
                              // e.g. UNIX pid.
                              // Note that this may be dynamic due
                              // to process migration.
                              // present if bit  9 is set in flags
   [integer scheduling_state;] // Current scheduling state:
                              // 0: running, 1: sleeping/blocked,
                              // 2: ready/runnable, 3: zombie,
                              // 4: stopped/suspended
                              // present if bit 10 is set in flags
   [floating total_time;]     // Sum of user and system time of the
                              // process since it started
                              // present if bit 11 is set in flags
                              //
                              // Optional components:
                              //
   [integer priority;]        // Current scheduling priority
                              // see proc_set_priority
                              // present if bit 12 is set in flags
   [floating system_time;]    // System time of the process since
                              // it started
                              // present if bit 13 is set in flags
```

```
    [integer memory_size;]       // Current process size
                                 // present if bit 14 is set in flags
    [integer resident_size;]     // Amount of main memory currently used
                                 // present if bit 15 is set in flags
    [integer max_resident_size;] // Maximum amount of main memory used
                                 // since process started
                                 // present if bit 16 is set in flags
    [integer int_resident_size;] // Integral of main memory size used
                                 // since process started
                                 // (unit: Bytes * seconds)
                                 // present if bit 17 is set in flags
    [integer minor_page_faults;] // Number of page faults since process
                                 // started that did not require physical
                                 // I/O
                                 // present if bit 18 is set in flags
    [integer major_page_faults;] // Number of page faults since process
                                 // started that did require physical I/O
                                 // present if bit 19 is set in flags
    [integer swaps;]             // Number of times the process has been
                                 // swapped out of main memory since it
                                 // started
                                 // present if bit 20 is set in flags
    [integer file_input;]        // number of file inputs since process
                                 // started
                                 // present if bit 21 is set in flags
    [integer file_output;]       // number of file outputs since process
                                 // started
                                 // present if bit 22 is set in flags
    [integer vol_cont_switch;]   // number of voluntary context switches
                                 // since process started
                                 // (e.g. wait for resource)
                                 // present if bit 23 is set in flags
    [integer invol_cont_switch;] // number of involuntary context
                                 // switches since process started
                                 // (time slice exceeded)
                                 // present if bit 24 is set in flags
  } Proc_dynamic_info;
  struct {
     Proc_static_info statinfo;
     Proc_dynamic_info dyninfo;
  }
  proc_get_info(integer* proc_list, integer flags)
```

Detailed information about a set of processes can be obtained by the **proc_get_info** service. **proc_list** defines the processes that have to be inspected. The second parameter **flags** is a bit set that allows to mask each kind of information individually. E.g. if **flags** is equal to 514 (0x202), the processes' argv vector (i.e. name and command line parameters, bit 1) and their node local identifiers (bit 9) will be returned. So it is possible to get all relevant process information with a single service request, but still a monitoring system only needs to retrieve the information that is really needed.

When specifying **proc_get_info([],0)**, no further information about processes is returned. However, the reply will contain a full expansion of the empty process list, i.e. will provide the tool with the tokens of all monitored processes.

2. **proc_read_memory** .... read the memory of a process. (R)

```
integer*
proc_read_memory(token* proc_list, integer addr,
                 integer blocklength, integer stride, integer count)
```

For each process in **proc_list**, this service reads **count** contiguous memory blocks from the process' memory and returns the contents as a list of bytes. The result is a raw memory image, i.e. a sequence of Byte values; the monitoring system does not perform any processing (e.g. Byte swapping) of this data. Each memory block read is of Byte-size **blocklength**, **addr** is the address of the first block, and **stride** is the separation between the start of two subsequent blocks in Bytes. Thus, the first **blocklength** Bytes of the result will be read from addresses **addr** ... **addr+blocklength**−1, the next **blocklength** Bytes from addresses **addr+stride** ... **addr+stride+blocklength**−1 and so on. **stride** must not be smaller than **blocklength**.

3. **proc_read_memory_bin** .... read the memory of a process. (O)

```
binary
proc_read_memory_bin(token* proc_list, integer addr,
                     integer blocklength, integer stride, integer count)
```

This service has exactly the same semantics as **proc_read_memory**, however, it returns its result in a binary format.

4. **proc_get_loader_info** .... return the loader information of a process. (P)

```
typedef struct {
   string  path_name;        // The (full) 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 num_load_modules;  // Number of load modules
   Loader_info* loader_info;  // loader info for each load module
}
proc_get_loader_info(token* proc_list)
```

On some parallel computers, e.g. the Parsytec GC/PowerPlus running PARIX, 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 **proc_get_loader_info** returns this information for each load module of a process contained in **proc_list**. This service may also be used with operating systems where code (e.g. libraries) can be loaded dynamically.

The first element in the **loader_info** list will contain the information on the process' main module (i.e. the statically linked part of its executable). Any implementation must at least provide this part of the information; additional list elements providing information on dynamically loaded libraries are optional. If the target system does not relocate the code of a process (as it is the case with UNIX), a result with all start addresses equal to zero and all lengths equal to the total address space of the target system is permissible.

Event Services

The following services notify the caller on certain events related to the monitored processes. When one of these events is detected, all threads in the process that generated the event are temporarily suspended until all action lists associated with the event have been executed.

Note that only the service specific event context parameters are specified in this section. The common event context parameters contain further information on the detected event, especially the tokens of the node and process where the event occurred.

1. **proc_has_terminated** .... A process has terminated. (R)

```
void
proc_has_terminated(token *proc_list)
--> void             // no service specific event context parameters
```

   Event **proc_has_terminated** is raised when a process in **proc_list** has terminated. There is no guarantee that the process is still accessible when the event is detected. However, implementations should try to detect this event before the process becomes inaccessible, whenever possible.

2. **proc_has_been_stopped** .... A process has been stopped by the monitoring system. (R)

```
void
proc_has_been_stopped(token *proc_list)
--> void             // no service specific event context parameters
```

   This event is raised when all threads of a process in **proc_list** have been stopped by the monitoring system (using the **thread_stop** service).

3. **proc_has_been_continued** .... A process has been continued by the monitoring system. (R)

```
void
proc_has_been_continued(token* proc_list)
--> void             // no service specific event context parameters
```

   This event is raised when all threads of a process in **proc_list** has been continued again by the monitoring system (using the **thread_continue** service).

4. **proc_has_been_scheduled** .... Process has been scheduled. (O)

```
void
proc_has_been_scheduled(token *proc_list)
--> void            // no service specific event context parameters
```

On machines where process scheduling is observable, this event is raised each time a process in **proc_list** is scheduled by the operating system, i.e. it gets the CPU for one time-slice.

5. **proc_has_been_descheduled** .... Process has been descheduled. (O)

```
void
proc_has_been_descheduled(token *proc_list)
--> void            // no service specific event context parameters
```

On machines where process scheduling is observable, this event is raised each time a process in **proc_list** is descheduled by the operating system, i.e. releases the CPU, either since its time-slice elapsed or the process voluntarily blocked for some reason.

6. **proc_will_be_migrated** .... A process is going to be migrated. (O)

```
void
proc_will_be_migrated(token* proc_list)
--> struct {
   integer dest_node;  // destination node of migration
}
```

This event will be raised before the system is going to migrate a process in **proc_list**. The event context parameters include the process' destination node. This service allows a tool to perform some cleanup work before a process is migrated.

7. **proc_has_been_migrated** .. A process has been migrated to another node. (O)

```
void
proc_has_been_migrated(token* proc_list)
--> void            // no service specific event context parameters
```

This event is raised, after a process in **proc_list** has been migrated to another node. This service allows a tool to perform some initialization work after a process has been migrated.

8.1.3 Threads

Threads are the only objects in the monitored system that actually execute code. See Section 8.1.2 for a discussion of the relation between threads and processes.

Manipulation Services

The following services are provided to manipulate the behavior of threads. Note that there are no services to attach to a thread, since threads are automatically attached. See **proc_attach** for a rationale.

1. **thread_detach** .... detach from a thread. (O)

   ```
   void
   thread_detach(token* thread_list)
   ```

   The **thread_detach** service detaches the monitoring system from the threads specified in **thread_list**. When a thread is detached, it is no longer included in the set of monitored threads. Note however, that conditional requests defined for the detached threads will not be deleted automatically – the events simply will no longer be detected. Any subsequent service request for a detached thread will result in an error.

   Note that there is no way to re-attach to the detached thread, except for detaching and re-attaching the thread's process. The rationale for this service is that some programming libraries create threads for their internal usage, which should not be influenced by the monitoring system. Thus, it should be possible to detach from these threads.

2. **thread_stop** .... stop a thread. (P)

   ```
   void
   thread_stop(token* thread_list)
   ```

   This service stops all threads specified by **thread_list**. This is achieved by putting each thread into a stopped state, where it no longer gets any CPU cycles. From this state, the thread can only be released with the **thread_continue** service. The operation is idempotent, i.e. invoking it on an already stopped thread will in no way change the thread's state.

   Every implementation of OMIS must provide the ability to stop all threads of a process. Stopping individual threads of a process is an optional feature.

3. **thread_continue** .... continue a thread. (P)

   ```
   void
   thread_continue(token* thread_list)
   ```

   This service removes all threads specified by **thread_list** from the stopped state. The operation is idempotent, i.e. invoking it on a thread that is not stopped will in no way change the thread's state. Note that the thread only starts executing again when there is no other reason preventing its execution (e.g. it may still be suspended due to a call to **thread_suspend** or because action lists associated with an event that occured in this thread are executed).

   Every implementation of OMIS must provide the ability to continue all threads of a process. Continuing individual threads of a process is an optional feature.

4. **thread_suspend** .... temporarily suspend a thread. (R)

```
void
thread_suspend(token* thread_list)
```

This service suspends all threads specified by **thread_list**. Like **thread_stop**, the threads are put into a suspended state where they no longer get any CPU cycles. However, the suspended state is logically different from the stopped state, i.e. a suspended thread cannot be resumed with **thread_continue**. Furthermore, **thread_suspend** is not idempotent, i.e. there is a suspend count which means that a thread suspended twice must also be resumed twice. Finally, when threads are to be suspended, **thread_suspend** may actually prevent an arbitrary superset of these threads from executing, provided that the implementation guarantees that they will again get the CPU when there is no longer any thread in the suspended state.

The rationale for having both **thread_stop** / **thread_continue** and **thread_suspend** / **thread_resume** is the different use of these services. **thread_stop** and **thread_continue** are used e.g. for debugging when a thread has to be stopped for a longer period of time (visible for the user of a tool and for other tools), while **thread_suspend** and **thread_resume** are used e.g. to ensure that the state of some threads do not change during the execution of an action list or a sequence of unconditional requests (invisible for the user of a tool and for other tools). Since on some platforms suspending a single thread may be extremly complicated if not impossible, an implementation is free to suspend more than the specified threads, e.g. all threads in the same process. High quality implementations should of course try to be as little intrusive as possible.

5. **thread_resume** .... resume a thread. (R)

```
void
thread_resume(token* thread_list)
```

This service decrements the suspend count for all threads specified by **thread_list**. When the count is zero for a thread, that thread is removed from the suspended state. However, an implementation is free to still prevent these threads from executing until there is no longer any thread in a suspended state. In any case, execution of the thread may still be prevented by a previous call to **thread_stop** or the execution of an action list associated with an event in that thread.

See **thread_suspend** for a rationale.

6. **thread_send_signal** .... send a signal to a thread. (O)

```
void
thread_send_signal(token* thread_list, integer sig)
```

On systems that allow signals to be sent to individual threads, this service sends the signal **sig** to all threads in **thread_list**.

7. **thread_set_priority** .... change priority of a thread. (O)

```
void
thread_set_priority(token* thread_list, integer val)
```

On systems providing a priority based thread scheduling, this service changes the scheduling priority of the threads in **thread_list** to **val**.

8. **thread_write_int_regs** .... write into a thread's integer registers. (R)

   ```
   void
   thread_write_int_regs(token* thread_list, integer reg, integer* val)
   ```

   Writes the values in **val** into the integer registers **reg**, **reg+1**, ... of all threads specified in **thread_list**. The register numbers and their bit-length depend on the node architecture; they are defined in the node processors' ABI (Application Binary Interface).

9. **thread_write_fp_regs** .... write into a thread's floating point registers. (R)

   ```
   void
   thread_write_fp_regs(token* thread_list, integer reg, floating* val)
   ```

   Writes the values in **val** into the floating point registers **reg**, **reg+1**, ... of all threads specified in **thread_list**. The register numbers and their bit-length depend on the node architecture; they are defined in the node processors' ABI (Application Binary Interface).

10. **thread_goto** .... set the PC of a thread. (O)

    ```
    void
    thread_goto(token* thread_list, integer addr)
    ```

    Sets the program counter (PC) of the threads specified by **thread_list** to the value **addr**. On some processors (e.g. Sparc) this also includes initialization of pipeline registers. The service has the effect of executing a jump instruction to that address in the current context of the specified threads. Note that this service only manipulates the threads' program counter, it does not continue the threads if they are stopped.

11. **thread_call** .... save and set the PC of a thread. (O)

    ```
    void
    thread_call(token* thread_list, integer addr)
    ```

    Like **thread_goto**, this service sets the program counter (PC) of the threads specified by **thread_list** to the value **addr**. However, it first saves the current PC (i.e. the return address) in a way conforming to the node processors' calling conventions, effectively performing a subroutine call in the current context of the specified threads. The tool must ensure that the proper parameters for the called subroutine (if any) are already loaded into the proper registers and/or memory locations. Note that this service only manipulates the threads' registers and stack, it does not continue the threads if they are stopped.

Information Services

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

1. **thread_get_info** .... get information on threads. (P)

```
typedef struct {                // Static info. Independent of time.
                                //
                                // Required components:
                                //
   [token process;]             // Token of the process containing
                                // this thread
                                // present if bit 0 is set in flags
   [integer global_id;]         // Global id of this thread, as
                                // defined by the programming library
                                // used
                                // present if bit 1 is set in flags
                                //
                                // Optional components:
                                //
   [integer root_funct;]        // Address of the thread's root function
                                // present if bit 2 is set in flags
   [token parent;]              // Token of the thread's parent thread
                                // present if bit 3 is set in flags
   [token message_queue;]       // Token of the thread's (logical)
                                // message queue, if threads in the same
                                // process have separate queues,
                                // undefined token otherwise. See
                                // the section on message queues for a
                                // discussion.
                                // present if bit 4 is set in flags
   [integer stack_size;]        // Stack size of the thread
                                // present if bit 5 is set in flags
} Thread_static_info;

typedef struct {                // Dynamic info. Changes over time.
                                // All times are returned in seconds,
                                // all sizes in Bytes.
                                //
                                // Required components:
                                //
   [token node;]                // Token of the node where this
                                // thread is located.
                                // Note that this may be dynamic due
                                // to process/thread migration.
                                // present if bit 6 is set in flags
   [integer local_id;]          // Local id of this thread, as defined
                                // by the node operating system.
                                // Note that this may be dynamic due
                                // to process/thread migration.
                                // present if bit 7 is set in flags
   [integer scheduling_state;]  // Current scheduling state:
                                // 0: running, 1: sleeping/blocked,
                                // 2: ready/runnable, 3: zombie,
                                // 4: stopped/suspended
                                // present if bit 8 is set in flags
   [floating total_time;]       // Sum of user and system time of the
                                // thread since it started
                                // present if bit 9 is set in flags
                                //
                                // Optional components:
                                //
   [integer priority;]          // Current scheduling priority
                                // see thread_set_priority
                                // present if bit 10 is set in flags
   [floating system_time;]      // System time of the thread since
                                // it started
                                // present if bit 11 is set in flags
} Thread_dynamic_info;
struct {
   Thread_static_info statinfo;
   Thread_dynamic_info dyninfo;
}
thread_get_info(token* thread_list, integer flags)
```

Detailed information about a set of threads can be obtained by the **thread_get_info** service. **thread_list** defines the threads that have to be inspected. The second parameter **flags** is a bit set that allows to mask each kind of information individually. E.g. if **flags** is equal to 5, for each thread the address of its root function (i.e. the address of the top level function executed by that thread) and the token of the process containing that thread will be returned. So it is possible to get all relevant thread information with a single service request, but still a monitoring system only needs to retrieve the information that is really needed.

When specifying **thread_get_info([],0)**, no further information about threads is returned. However, the reply will contain a full expansion of the empty thread list, i.e. will provide the tool with the tokens of all attached threads in the monitored processes.

2. **thread_get_backtrace** .... determine a thread's procedure stack backtrace. (R)

```
typedef struct {
   integer pc;          // Program counter / return address
   integer fp;          // Frame pointer
} Stack_element;
struct {
   integer num_frames;
   Stack_element* stack;
}
thread_get_backtrace(token* thread_list, integer depth)
```

The service **thread_get_backtrace** returns the current procedure stack backtrace of all threads specified by **thread_list**. The backtrace 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. The parameter **depth** determines the maximum depth of the backtrace, i.e. at most the first **depth** entries of the backtrace will be returned. If **depth** is zero, the complete backtrace will be returned. This service is mainly used for debugging or performance analysis based on sampling.

3. **thread_read_int_regs** .... read integer registers of a thread. (R)

   ```
   integer*
   thread_read_int_regs(token *thread_list, integer reg, integer num)
   ```

   Reads **num** integer registers starting at register **reg** in each of the threads specified by **thread_list** and returns their contents. The register numbers and their bit-length depend on the node architecture; they are defined in the node processors' application binary interface (ABI).

4. **thread_read_fp_regs** .... read floating point registers of a thread. (R)

   ```
   floating*
   thread_read_fp_regs(token *thread_list, integer reg, integer num)
   ```

   Reads **num** floating point registers starting at register **reg** in each of the threads specified by **thread_list** and returns their contents. The register numbers and their bit-length depend on the node architecture; they are defined in the node processors' application binary interface (ABI).

Event Services

The following services notify the caller on certain thread related events that occur in the monitored application. When one of these events is detected, the thread that generated the event will be temporarily suspended until all action lists associated with the event have been executed. When this thread is suspended, most implementations will at the same time also suspend all other threads within the same process, since this is the default behavior of the underlying operating system mechanisms. However, OMIS does not guarantee this behavior, since there are platforms where it cannot be achieved. Since threads use a common memory, this results in a chance that other threads change the memory contents before or while the action lists are executed. To avoid this behavior, enclose the action list in a **thread_suspend($proc) / thread_resume($proc)** pair. A high quality implementation should try to stop all thread in the process as soon as possible in this case, thus reducing (but not totally eliminating) this problem.

Note that only the service specific event context parameters are specified in this section. The common event context parameters contain further information on the detected event, especially the tokens of the node, process and thread where the event occurred.

1. **thread_adds_node** A thread adds a new node to its application's node set. (O)

   ```
   void
   thread_adds_node(token* thread_list)
   --> struct {
      token new_node;      // token of the node that is being added
   }
   ```

   The service **thread_adds_node** reports when a thread in **thread_list** adds a processing node to its application's node set. The event is raised before the thread can perform any operations concerning the new node (e.g. start a process). The event context parameter **new_node** contains the token of the new node. Thus, **node_attach** may be used in the action list to attach the monitoring system to that node.

2. **thread_removes_node** A thread removes a new node from its application's node set.

   ```
   void
   thread_removes_node(token* thread_list)
   --> struct {
      token rem_node;      // token of the node that is being removed
   }
   ```

   The service **thread_removes_node** reports when a thread in **thread_list** removes a processing node from its application's node set. The event is raised before the node is actually removed. Note that depending on the target platform, the monitor on the node may be killed when the node is removed, so the actions attached to this event service in a conditional request are the last services that can be safely executed for that node. Usually, the actions will include a **node_detach** service.

3. **thread_creates_proc** .... A thread creates a new process. (O)

   ```
   void
   thread_creates_proc(token* thread_list)
   --> struct {
      token new_proc;      // token of the process that is being created
   }
   ```

   The service **thread_creates_proc** reports when a thread in **thread_list** creates a new process. The event is raised immediately before the initial thread of the new process starts its execution. The event context parameter **new_proc** contains the token of the new process. Thus, **proc_attach** may be used in the action list to attach the monitoring system to that process.

   If the programming model of the monitored thread allows to create processes on a remote node without a need to first explicitly add this node to the application's node set (as it has to be done for instance in PVM), the creation of a process on a node not yet used before will first raise a **thread_adds_node** event, followed by a **thread_creates_proc** event.

   The **thread_creates_proc** is only guaranteed to be raised when the thread creates the process by 'legal' means of the programming library used. This means that if e.g. a PVM task creates a new process via a **fork** system call, process creation may not be detected.

4. **thread_creates_thread** .... A thread creates a new thread. (O)

   ```
   void
   thread_creates_thread(token* thread_list)
   --> struct {
      token new_thread;      // token of the thread that is being created
   }
   ```

   The service **thread_creates_thread** reports when a thread in **thread_list** creates a new thread. The event is raised immediately before the new thread starts its execution. The event context parameter **new_thread** contains the token of the new thread. Note that if the thread is created in an already attached process, the monitoring system will automatically attach to the new thread. See **proc_attach** for a rationale.

   This event is also raised for the initial thread of a new process. Thus, if both events are monitored and a thread creates a new process, first **thread_creates_process** will be raised, followed by **thread_creates_thread**.

5. **thread_has_terminated** .... A thread has terminated. (R)

```
void
thread_has_terminated(token* thread_list)
--> void            // no service specific event context parameters
```

Event **thread_has_terminated** is raised when a thread in **thread_list** has terminated. There is no guarantee that the thread is still accessible when the event is detected.

6. **thread_received_signal** .... A thread received a signal. (O)

```
void
thread_received_signal(token *thread_list, integer *sig_list)
--> struct {
   integer sig;    // signal number
}
```

This event is raised whenever a thread in **thread_list** received a signal in **sig_list**. It provides the signal number as an additional event context parameter.

7. **thread_has_blocked** .... A thread has blocked. (O)

```
void
thread_has_blocked(token* thread_list)
--> void            // no service specific event context parameters
```

This event is raised when a thread in **thread_list** has blocked e.g. in a blocking communication or synchronization call.

8. **thread_has_been_unblocked** .... A thread has been unblocked. (O)

```
void
thread_has_been_unblocked(token* thread_list)
--> void            // no service specific event context parameters
```

This event is raised when the blocking condition (e.g. due to a communication or synchronization call) of a thread in **thread_list** has been removed again.

9. **thread_has_been_stopped** .... A thread has been stopped by the monitoring system. (R)

```
void
thread_has_been_stopped(token* thread_list)
--> void            // no service specific event context parameters
```

This event is raised when a thread in **thread_list** is stopped by the monitoring system (using the **thread_stop** service).

10. **thread_has_been_continued** .... A thread has been continued by the monitoring system. (R)

```
void
thread_has_been_continued(token* thread_list)
--> void            // no service specific event context parameters
```

This event is raised when a thread in **thread_list** is continued by the monitoring system (using the **thread_continue** service).

11. **thread_has_been_scheduled** .... Thread has been scheduled. (O)

```
void
thread_has_been_scheduled(token *thread_list)
--> void            // no service specific event context parameters
```

On machines where thread scheduling is observable, this event is raised each time a thread in **thread_list** is scheduled by the operating system, i.e. it gets the CPU for one time-slice.

12. **thread_has_been_descheduled** .... Thread has been descheduled. (O)

```
void
thread_has_been_descheduled(token *thread_list)
--> void            // no service specific event context parameters
```

On machines where thread scheduling is observable, this event is raised each time a thread in **thread_list** is descheduled by the operating system, i.e. releases the CPU, either since its time-slice elapsed or the thread voluntarily blocked for some reason.

13. **thread_reached_addr** .... A thread reaches a given code address. (R)

```
void
thread_reached_addr(token* thread_list, integer address)
--> void            // no service specific event context parameters
```

The event is raised whenever a thread in **thread_list** is about to execute the machine instruction at the given **address**. This service will mainly be used to implement breakpoints for debugging purposes.

14. **thread_executed_insn** .... A thread has executed an instruction. (O)

```
void
thread_executed_insn(token *thread_list)
--> void            // no service specific event context parameters
```

This event is raised whenever a thread in **thread_list** has finished execution of a machine instruction. This service can be used to implement single stepping and execution tracing of threads.

15. **thread_executed_insn_call** .... A thread has executed an instruction (call is single instruction) (O)

```
void
thread_executed_insn_call(token *thread_list)
--> void            // no service specific event context parameters
```

This event is raised whenever a thread in **thread_list** has finished execution of a machine instruction. In contrast to **thread_executed_insn** this service regards a subroutine call as a single instruction, i.e. when a thread executes a call instruction, the event is raised not before the called subroutine returns. The main use of this service is single stepping and execution tracing of threads.

16. **thread_has_started_lib_call** .... A thread has invoked a call to the programming library. (R)
    thread_has_ended_lib_call .... A thread has returned from a call to the programming library. (R)

```
void
thread_has_started_lib_call(token* thread_list, string lib_call_name)
--> struct {
   any par1;      // the event context parameters are the input
   any par2;      // parameters of the library call
   ...
}

void
thread_has_ended_lib_call(token* thread_list, string lib_call_name)
--> struct {
   any par1;      // the event context parameters are the result
   any par2;      // parameters of the library call
   ...
}
```

These events are raised whenever a thread in **thread_list** is calling the specified routine of the parallel programming library (e.g. PVM). **thread_has_started_lib_call** is raised just before the routine is executed, while **thread_has_ended_lib_call** is raised just after the routine returns. In both cases, the event is not been raised if the routine has been called by another routine in the programming library, but only when it has been called directly by the application code.

The services are provided for all routines in the programming library; the value of the **lib_call_name** parameter specifies the routine's name. The service specific event context parameters of these services are the input or output parameters of the called library routine. They are named **par1**, **par2**, and so on. The number and the types of these parameters depend both on the programming library used and the selected library call. They are defined in the specification of the extension handling the specific programming library.

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

17. **thread_has_started_sys_call** .... A thread has invoked a system call. (O)
    end_sys_call .... A thread has returned from a system call. (O)

```
void
thread_has_started_sys_call(token* thread_list, string sys_call_name)
--> struct {
   any par1;      // the event context parameters are the input
   any par2;      // parameters of the system call
   ...
}
```

```
void
thread_has_ended_sys_call(token* thread_list, string sys_call_name)
--> struct {
   any par1;      // the event context parameters are the result
   any par2;      // parameters of the system call
   ...
}
```

These events are raised whenever a thread in **thread_list** is calling the specified system call of the node operating system. **thread_has_started_sys_call** is raised just before the system call is executed, while **thread_has_ended_sys_call** is raised just after the system call returns.

The services are provided for all system calls of the processing nodes' operating system(s); the value of the **sys_call_name** parameter specifies the name of the system call. The service specific event context parameters of these services are the input or output parameters of the system call. They are named **par1**, **par2**, and so on. The number and the types of these parameters depend both on the operating system and the selected system call. They are defined in the specification of the extension handling the specific operating system.

18. **thread_has_received_tagged_msg** .... A thread has received a tagged message. (O)

```
void
thread_has_received_tagged_msg(token* thread_list, integer* tag_list)
--> struct {
   token    msg;               // Token of received message
   token    sender;            // Token of thread that sent the message
   integer  size;              // Size of the message
   integer  monitor_tag;       // Tag added by the monitoring system
}
```

This service reports the receipt of a message by a thread in **thread_list**, iff the message has been tagged by the monitoring system and either the tag is contained in **tag_list** or **tag_list** is an empty list. See **message_tag** for more information on message tags.

8.1.4 Messages

In order to support the monitoring of message passing systems, OMIS includes services operating on messages and message queues. Both messages and message queues are handled as abstract data structures, i.e. their exact structure and implementation is not visible at the monitoring interface, since it depends on the particular target platform. Services allowing to construct messages or to view the contents of a message therefore must be contained in an extension for that platform.

The model of a message passing system used by OMIS is that messages are sent by a thread to a message queue from which they may be read by the receiver. OMIS does not make any assumption on the relation between that message queue and the receiver. Basically, there are three possibilities for this relation:

1. The message queue may be associated with a single receiver thread. In this case, we have direct message passing between threads. The message queue belonging to a thread can be obtained via the **thread_get_info** service.
2. The message queue may be associated with a receiver process. Here, the message is received by an arbitrary thread in that process. In this case, **proc_get_info** returns the token of the message queue associated with a process.
3. The message queue may have a mailbox semantics, i.e. any process may read the message from the queue. For such a platform, an extension must provide a service returning the message queue tokens for the existing mailboxes.

On singlethreaded systems, the first two cases coincide. The fact that in cases 1 and 2 the model uses only a single message queue per receiver (thread or process) does not restrict the applicability of OMIS for systems that have separate queues for e.g. differently tagged messages or messages originating from different senders. In these cases, the different queues can always be merged into a single (logical) queue, which again can be easily separated into the original ones.

Manipulation Services

The services in this group will allow to modify message queues and single messages.

1. **message_insert_into_queue** .... Insert a message into a message queue. (O)

   ```
   void
   message_insert_into_queue(token msg, token queue, integer pos)
   ```

   For debugging message passing errors, the service **message_insert_into_queue** allows to insert a message **msg** at a given position **pos** in a message queue **queue**. Both message and message queue are specified by a token. The token of a process' or thread's message queue can be determined by the **proc_get_info** and **thread_get_info** services; the message token is usually taken from the result of the **message_create** service.

   The parameter **pos** defines the position of the message in the message queue (as returned by **message_queue_get_info**) after which the new message will be inserted. If **pos** is 0, the message is inserted at the beginning of the queue, if **pos** is -1, the message is inserted at the end of the queue. The service does *not* duplicate the specified message **msg** before it is inserted into the given queue, so the caller must ensure that the message is not already contained in a message queue. Otherwise, unexpected behavour may result.

   Each implementation of this service must at least provide the ability to insert a message at the end of a message queue.

2. **message_queue_remove** .... Remove a message from a message queue. (O)

```
void
message_queue_remove(token queue, token msg)
```

This service removes the specified message **msg** from the given message queue **queue** (but does not destroy the message). If **msg** is not in **queue**, an error will be returned. The service may be used mainly for debugging message passing errors and for program testing.

3. **message_queue_clear** .... Clear a message queue. (O)

```
void
message_queue_clear(token* queue_list)
```

This service deletes all messages from the message queues given in **queue_list**.

4. **message_tag** .... Add a tag to a message (O)

```
void
message_tag(token* msg_list, integer tag)
```

To support debugging of message passing programs, OMIS defines services that use a special message tag. This tag is independent of any message tag defined by the programming model. In fact, 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.

This tag 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 monitoring system can stop the receiver, when it receives exactly 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 **message_tag** service puts the specified **tag** (which must be different from 0) into the messages specified by their tokens in **msg_list**. It will mainly be used as an action in combination with a **thread_has_started_lib_call** event for a 'send' library call that provides the message token as an event context parameter.

5. **message_create** .... Create a new message. (O)

```
token
message_create()
```

This service creates a new message and returns the message token. The message may then be initialized with **message_copy** or with a service provided by an extension for the specific programming library.

6. **message_copy** .... Copy a message. (O)

```
void
message_copy(token src, token dst)
```

This service copies the message specified by **src** into the (existing) message specified by **dst**.

7. **message_destroy** .... Destroy a message. (O)

```
void
message_destroy(token* msg_list)
```

This service destroys the messages given in **msg_list**. The caller must take care not to destroy messages that are still accessible by the monitored application.

Information Services

Currently, this group contains only one service:

1. **message_queue_get_info** .... Return information on a message queue. (O)

```
typedef struct {
   [token    msg;]          // Token of this message
                            // present if bit 0 is set in flags
   [token    sender;]       // Token of thread that sent this message
                            // present if bit 1 is set in flags
   [integer  size;]         // Size of the message
                            // present if bit 2 is set in flags
   [integer  monitor_tag;]  // Tag added by the monitoring system
                            // 0 --> message has not been tagged
                            //       by the monitoring system
                            // present if bit 3 is set in flags
} Message_info;
struct {
   integer queue_length;    // number of messages in queue
   Message_info* messages;
}
message_queue_get_info(token* queue_list, integer flags)
```

This service returns information on the message queues specified in **queue_list**. In analogy to **process_get_info** a bit set mechanism allows 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.

Event Services

Only one event service is especially provided for the monitoring of message passing programs. Other events, such as the beginning and the end of send or receive library calls, can be monitored using the **thread_has_started_lib_call** and **thread_has_ended_lib_call** services (see Section 8.1.3). When the event defined below is detected, the monitoring system ensures that no message is removed from the message queue until all action lists have been executed. However, messages may still be appended to the queue.

Note that only the service specific event context parameters are specified in this section. The common event context parameters contain further information on the detected event, especially the token of the node where the event occurred.

1. **message_queue_has_been_extended** .. A message has been appended to a message queue. (O)

```
void
message_queue_has_been_extended(token* queue_list)
--> struct {
   token    queue;            // Token of this message queue
   token    msg;              // Token of appended message
   token    sender;           // Token of thread that sent this message
   integer  size;             // Size of the message
   integer  monitor_tag;      // Tag added by the monitoring system
                              // 0 --> message has not been tagged
                              //       by the monitoring system
}
```

This event is raised when a message is inserted into a queue contained in **queue_list**. The token of the process and thread the message queue belongs to are contained in the standard event context parameters. This event is essential for tools based on event traces that need information on the size or contents of the message queue.

8.2 Monitor Objects

Currently, OMIS defines two types of objects in the monitoring system: user defined events and conditional 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 Conditional Service Requests

Manipulation Services

1. **csr_enable** .... Enable a previously defined conditional service request. (R)
   csr_disable .... Disable a previously defined conditional service request. (R)

   ```
   void
   csr_enable(token* csr_list)

   void
   csr_disable(token* csr_list)
   ```

   These services will enable or disable the previously defined conditional service requests specified by **csr_list**. **csr_list** is a list of conditional service request tokens, which are returned as the immediate result of a conditional service request. Since all conditional service requests are initially disabled, they must be 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 or conditional performance measurements are possible.

2. **csr_delete** .... Delete a previously defined conditional service request. (R)

   ```
   void csr_delete(token* csr_list)
   ```

   This service deletes the conditional service requests specified by the tokens in **csr_list**.

8.2.2 User Defined Events

Manipulation Services

1. **user_event_create** .... Create a user defineded event. (R)

   ```
   token
   user_event_create()
   ```

   An OMIS compliant monitoring system allows tools to define their own events via this service. Once the event has been defined, it can be raised using the **user_event_raise** service.

2. **user_event_raise** .... Raise a user defined event. (R)

```
void
user_event_raise(token user_event, any* params, integer resume)
```

This service will raise the user defined event specified by its token that has been returned by a previous call to **user_event_create**. **params** is a list of parameters that will be copied to the service specific event context parameters of the **user_event_has_been_raised** event service. The size of this list, i.e. the number of parameters is arbitrary.

If the service is contained in the action list of a conditional service request, the **resume** parameter determines what happens to the object where the event occured that lead to the execution of this action list. If **resume** is zero, the object will not be allowed to change its state until all action lists associated with the user event have been executed. In this case, the standard event context parameters of **user_event_has_been_raised** will specify that object as being the source of the event.

If **resume** is non-zero, the object may change its state before (and even while) the action lists associated with the user event are executed. In this case, or if the service is contained in an unconditional service request, the standard event context parameters of **user_event_has_been_raised** will specify an undefined object as being the source of the event.

3. **user_event_destroy** .... Destroy a user defined event. (R)

```
void
user_event_destroy(token user_event)
```

This service is used to destroy a user defined event given by its token when it is no longer needed. Any subsequent call to **user_event_raise** for this event will result in an error. The tool is responsible for deleting all conditional service requests referring to the destroyed user defined event.

Event Services

1. **user_event_has_been_raised** .... A user defined event has been raised. (R)

```
void
user_event_has_been_raised(token user_event)
--> struct {
       par1;        // parameters specified in the call to raise_event
       par2;
       ...
}
```

The service **user_event_has_been_raised** gives notice that the specified user event has been raised using **user_event_raise** service. The service specific event context parameters contain the values of the elements in the **params** list passed to **user_event_raise**. **par1** contains the value of the first element of that list, **par2** the value of the second one and so on. The number of service specific event context parameters depends on the length of the **params** list.

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** of actions with event **E1** or event **E2**, you can specify:

```
user_event_create()                    // returns token e_321
E1: user_event_raise(e_321, [], 1)
E2: user_event_raise(e_321, [], 1)
user_event_has_been_raised(e_321): A
```

In addition, user defined events allow to chain actions. For instance, you could provide some service **filter(var, val, user_event)** in a distributed tool extension that raises a user defined event, iff **var == val**. If we assume that the fourth parameter returned by the **start_lib_call(tid_list, "MPI_Send")** service is the destination process token, then in the following situation

```
user_event_create()                   // returns token e_23
thread_has_started_lib_call([p_2], "MPI_Send"): filter($par4, p_5, e_23)
user_event_has_been_raised(e_23): A
```

action list **A** will be executed, iff process **p_2** sends to process **p_5**.

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

8.2.3 Miscellaneous

Synchronous Services

1. **print** .... Return arguments. (R)

```
struct {
   integer numargs;  // number of elements in following list
   any*    args;     // copy of argument list
}
print(any* args)
```

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

2. **version** .... Return version information. (R)

```
struct {
   integer omis_major;    // Major version of OMIS specification
                          // the monitoring system complies to.
   integer omis_minor;    // Minor version of OMIS
   string  ocm_ident;     // String identifying the implementation
                          // of the OMIS compliant monitoring system
                          // (vendor specific)
   integer ocm_major;     // Major version of monitoring system
                          // (defined by vendor)
   integer ocm_minor;     // Minor version of monitoring system
                          // (defined by vendor)
}
version()
```

This service returns version information on the monitoring system. This includes the proper version of the OMIS specification (i.e. **omis_major** = 2, **omis_minor** = 0 for this version), and a vendor-defined name and version numbers for monitor implementation.

3. **extensions** .... Return a list of available extensions. (R)

   ```
   struct {
      integer numext;      // number of elements in following list
      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 returns 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.

4. **services** .... Return a list of available services. (R)

   ```
   struct {
      integer numfully;    // number of elements in following list
      string* fully_impl;  // names of fully implemented services
      integer numpart;     // number of elements in following list
      string* part_impl;   // names of partially implemented services
   }
   services(string extension)
   ```

   This service allows to obtain the names of all services which are fully or partially implemented by the extension defined by its prefix string. If **extension** is the empty string, the names of all services provided by the basic monitoring system are returned.

Chapter 9

Specification of a PVM Extension

9.1 Data Types

The PVM extension defines one new token class: the group token. The prefix 'pvm_g_' is used to characterize group tokens. The expansion operation on a group token leads to a list of all attached processes that are in the specified group. There are no localization operations: groups cannot be localized on a specific node; likewise, since a PVM task can be member of several groups, it is not possible to uniquely convert a process token into a group token.

In addition, there is another implicit object in PVM: the user's virtual machine, consisting of a set of nodes and several application processes (tasks) running on these nodes. Since in PVM there can be only one virtual machine per user, there is no need to introduce a token for this object, however, there are services for this object class.

9.2 System Objects

9.2.1 Virtual Machine

Information Services

1. **pvm_vm_get_nodelist** .... get all nodes of the virtual machine. (R)

```
typedef struct {
   token   node;     // Token of this node
   integer tid;      // PVM task id of PVM daemon on this node
} nodeinfo;
struct {
   integer   num;    // Number of elements in the following list
                     // i.e. number of nodes in the virtual machine
   nodeinfo* nodes;  // Information on each node
}
pvm_vm_get_nodelist()
```

   This service returns information on the nodes currently being in the user's virtual machine. The information for each node includes the node's token and the PVM task id of the PVM daemon located on this node.

2. **pvm_vm_get_proclist** .... get all processes in the virtual machine. (R)

```
typedef struct {
      token proc;      // The process token
      token thread;    // The token of the process' (only) thread
   } taskinfo;

   struct {
      integer   num;   // Number of elements in the following list
                       // i.e. number of PVM tasks in the virtual machine
      taskinfo* tasks; // List of process/thread tokens of all PVM tasks
   }
   pvm_vm_get_proclist()
```

This service returns the list of tokens of all processes (i.e. PVM tasks) currently existing in the user's virtual machine, together with the token of the (only) thread in each of these processes.

9.2.2 Processes

Information Services

1. **pvm_proc_get_buffers** ..... Get information on a process' message buffers. (O)

```
   typedef struct {
      integer bufid;    // The buffer ID as specified by PVM
      token   msg;      // The message token for this buffer
   } Buffer;
   struct {
      integer numbufs;  // Number of elements in following list
      Buffer* bufs;     // List of all existing message buffers in process
   }
   pvm_proc_get_buffers(token* proc_list)
```

This service returns information on a process' PVM message buffers. The first element in the returned list specifies the active receive buffer, the second one the active send buffer. The following entries contain message buffers saved by the PVM **pvm_setrbuf** and **pvm_setsbuf** library calls.

9.2.3 Threads

Event Services

1. **thread_has_started_lib_call** .... A thread has invoked a PVM call. (R)
   thread_has_ended_lib_call .... A thread has returned from a PVM call. (R)

   The PVM extension allows to use these services for all documented PVM calls. The values of the input and output parameters of these calls can be accessed via the event context parameters.

   In this version of the specification of the PVM extension, we will not give an exact definition of this service for each PVM call, but rather explain the general concepts. As a convention, **$par0** represents the function result, **$par1** its first parameter and so on. This implies that not all event context parameters have defined values for both services, e.g. **$par0** will be undefied for any call to **thread_has_started_lib_call**. If the function parameters have simple types, the corresponding event context parameters have the analogous type, e.g.

integer for parameters of type **int**, **short**, ... Function parameters with more complex types may not be accessible via an event context parameter.

However, for some calls there are event context parameters that do not correspond to an explicit parameter of the call. For the communication and packing/unpacking functions there is an event context parameter containing the token of the manipulated message, which is an implicit parameter of these calls. For send and receive functions another event context parameter contains the length of the message.

As an example, here is the specification of the services for the **pvm_send** function. Later versions of this document will contain such a specification for each PVM call.

```
start pvm_send:
    integer par1;     // tid of destination process
    integer par2;     // message tag
    token   par3;     // destination process
    token   par4;     // message being sent
    integer par5;     // message length

end pvm_send:
    integer par0;     // return value of function
```

9.2.4 Messages

The following services are provided to access messages in applications based on the PVM programming library. Note that the term 'message' here means an abstract message in the sense of OMIS, which is very similar, but not fully identical to a PVM 'message buffer'.

Manipulation Services

1. **pvm_message_pack** .... Pack a PVM message. (O)

```
void
pvm_message_pack(token msg, token sender, int tag,
                 string fmt, any* cont)
```

This service initializes or overwrites an existing message given by its token **msg** with the specified message data. **sender** is the token of the sender process; **tag** is the PVM message tag. **fmt** is a format string as specified in the documentation of **pvm_packf**, **cont** is a list of values whose types must conform with the specification given in **fmt**.

When the service exits successfully, the specified message will look exactly as if sent by process **sender** with the specified tag and contents.

Information Services

1. **pvm_message_unpack** .... Unpack a PVM message. (O)

```
struct {
   integer tag;     // PVM tag of this message
   token   sender;  // Token of sender process
   integer num;     // Number of elements in following list
   any*    cont;    // Message contents
}
pvm_message_unpack(token msg, string fmt)
```

This service returns the PVM specific information on a message. The message contents returned is acquired by unpacking the message using the given format string **fmt**, which must conform to the the documentation of **pvm_unpackf**. The result of unpacking is stored in the untyped list **cont**. When **fmt** is an empty string, **cont** will also be empty.

Note that since PVM does not store information on the data types packed into a message, a tool calling **pvm_message_unpack** must acquire this information from other sources (e.g. from the user, or by a detailed monitoring of the packing and exchange of messages)

9.2.5 Groups

Information Services

1. **pvm_group_get_info** .... Get information on a PVM group. (O)

```
struct {
   [string  name;]      // Group name
                        // present if bit 0 is set in flags
   [integer nummemb;]   // Number of group members
                        // present if bit 1 is set in flags
   [token*  members;]   // Token of processes that are member of the
                        // group
                        // present if bit 2 is set in flags
   [integer numwait;]   // Number of members waiting at the group's
                        // barrier
                        // present if bit 3 is set in flags
   [token*  waiting;]   // Token of processes waiting at the group's
                        // barrier
                        // present if bit 4 is set in flags
   [integer expected;]  // Number of expected calls to pvm_barrier (only
                        // valid if numwait > 0)
                        // present if bit 5 is set in flags
}
pvm_group_get_info(token* group_list, integer flags)
```

This service returns information on the PVM groups matching the token list **group_list**. The bit field **flags** determines which information will be contained in the result. Notice that the reply of the call **group_list([],0)** will contain the list of the tokens of all existing groups in the user's virtual machine.

Since in PVM there is a one-to-one correspondence between groups and barriers, the service **pvm_group_get_info** will also provide information on the barrier associated with the specified group(s), namely the number of processes waiting at the barrier, the list of waiting processes and the number of expected calls to **pvm_barrier**, as specified in the **count** parameter of that PVM call.

Part IV

Concepts for an Implementation

Chapter 10

The Design of an OMIS Compliant Monitoring System

A detailed design document of the OMIS compliant monitoring system OCM will be provided when the implementation is finished. The OCM will be provided as a reference implementation of OMIS and will be released unter the GNU license conditions.

Chapter 11

Time Schedule of Implementation

Start of implementation: January, 1997

Four researchers and three students of LRR-TUM are currently working on the implementation of OCM. Implementation will be finished in fall 1997.

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 adaptation of OMIS based tools.

Start of design phase: January, 1996

A group of six researchers and students at LRR-TUM worked out the design of the OMIS compliant monitoring system OCM based on PVM as programming paradigm and networks of workstations as target architecture.

Part V

Diverse

Chapter 12

Requests for Comments

In this chapter we summarize important open questions 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 a really object oriented manner instead of the current object based one, i.e. should we employ inheritance? This could have some advantages:

- The on-line monitoring interface specification could be defined in terms of even more 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**, and so on. A UNIX process class, derived from this abstract process class, would inherit these services and could define additional ones, e.g. **send_signal** or **set_priority**. A PVM task could be derived from a UNIX process and provide additional services such as **message_buffer**.

  This scheme would replace the current scheme of OMIS extensions by the more common scheme of creating derived classes.
- When object oriented techniques are also used for the monitors' implementation, its portability with respect to different programming libraries could probably be increased. 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 environments? Since the hardware model already accounts for SMPs, it might be not very difficult to do so. However, support for (virtual) shared memory programming models requires additional services that still have to be defined.

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

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

- conditions (filters) that control the execution of action lists.

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 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.

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 introduce a UNIX process switch here, the overhead would make the whole measurement useless. 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 **thread_has_started_lib_call** and **thread_has_ended_lib_call** services should be generic, in order to reduce the dependency between the monitoring system and the parallel programming library. 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 should mirror these interfaces. It is not yet known, how this can be achieved. A possible way for the hoster interface would be to intercept the calls to **pvm_reg_hoster**. In this way, the monitor knows to which tasks it must forward the information on new hosts or tasks. However, for the tasker interface, no such solution is possible, since the tasker usually wants to be the parent process of the newly created task, but the monitoring system also needs to be the parent.

As a result, tools requiring the use of the tasker interface can only be used at the same time, if they both use OMIS.

6 Since OMIS specifies (optional) services for process migration, a monitoring system implementing these services must also take care of properly handling requests when performing a migration. There are at least two situations that have to be handled:

1. There are active conditional service requests for the process that is to be migrated. The current specification of OMIS seems to imply that migration must not affect these services. This means that the monitoring system must somehow transfer the necessary monitoring activities and the associated management data to the process' destination node. It is not yet clear how this scheme can be implemented.
2. The execution of an action list is requested (either by an unconditional request or by the triggering of a conditional request), while a process named in that action list is migrated. In this situation, it is not clear what should happen. Basically, there are three possible solutions:
   (a) Block request processing until migration has finished. This scheme will completely hide migration from the tools. However, it may result in poor performance of the monitoring system and even may lead to deadlocks.
   (b) Try to process the request on both the source and the destination node. In these cases, a single service invoked for a single object may return two results (of which usually one will be an error message). If the tool is aware of this behavior, this may be a suitable solution.
   (c) Return an error. This is probably the least attractive possibility.

We will address these issues in the near future when we will explore the interaction of monitoring and process migration.

Acknowledgements

External Discussion Partners

During the issue of OMIS version 1.0 and OMIS version 2.0 we received much valuable feedback from many cooperation and discussion partners. There are plannings to integrate OMIS/OCM with the following projects:

- The SMiLE project (Shared Memory in a LAN-like Environment) by Hermann Hellwagner and Wolfgang Karl at LRR-TUM.
- The DOSMOS project (Distributed Objects Shared MemOry System) by Lionel Brunie, Laurent Lefevre, Olivier Reymann at Ecole Normale Supérieure of Lyon, France.
- The ←PBeam project by Stefan Petri (formerly Technische Universität Braunschweig, Germany, now LRR-TUM) and Bettina Schnor (now at Universität Lübeck, Germany).
- The Grade project by Peter Kacsuk at KFKI MSZKI,Research Institute of the Hungarian Academy of Sciences, Budapest, Hungary.
- The Mad project by Jens Volkert, Dieter Kranzlmller, and Sigfried Grabner at Department of Computer Science at University Linz, Austria.

We thank all researchers in these groups for there constant interest in our work and the input they gave us. Further critical referees of OMIS 1.0 have been:

Helmar Burkhart, Günter Dehner, Jack Dongarra, Wolfgang Gentzsch, Jeff Hollingsworth, Rusty Lusk, Barton Miller, Rod Oldehoeft, Douglas M. Pase, Jeffrey Vetter.

We thank you for your effort with reading our document and giving us feedback on the proposed project. We tried to make correction according to your suggestions.

We apologize for any name that might be mssing in this list. We promise to correct this as soon as we detect it.

The OMIS/OCM Team

We would like to thank our students Manfred Geischeder, Michael Uemminghaus, and Hans-Günter Zeller for their participation in the design phase of the OCM implementation.

Jörg Trinitis is currently in charge of managing this implementation. Many thanks to him for his activities.

Our students Alexander Fichtner, Matthias Hilbig, Carsten Isert, and Andreas Schmidt are currently very busy with implementing all the modules. We appreciate their support.

Many people at the chair were involved in discussions about details of OMIS, OCM, and The Tool-set that will be attached to the monitoring system. We thank all of them for their feedback they gave to us. Special thanks to Michael Oberhuber, who was heavily involved in discussing the specification in its first phase.

Glossary

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

Basic Service A service of an OMIS compliant monitoring system that is defined in Part 8 of this specification.

Conditional Service Request A service request of an OMIS compliant monitoring system that is composed from an event service (called the event definition) and an list consisting of information and/or manipulation services (called the action list). The action list will be executed whenever an event matching the event definition is detected by the monitoring system. Thus, an arbitrary number of service replies may result from a conditional service request.

Distributed Tool (DT) A distributed tool is spread over all nodes of our node set and usually has 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.

Event-Action-Paradigm The monitoring system's behavior is programed with a language that follows the event-action-paradigm of our monitoring concept. Every request to the monitoring system is composed of an event definition followed by the definition of one or several actions. The semantics is that the actions are invoked whenever the event takes place. Following the event-action-paradigm transfers the monitoring system into an autonomous component that observes and manipulates programs.

Event Context Parameters Event context parameters are reply values of asynchronous services such as e.g. the current time, the node where the event occured etc. They are not meant for being delivered to the tool directly. Instead they are used as input parameters for manipulation services and synchronous services.

Event Service A service that asks the monitoring system to detect a certain class of events in the monitored system. Event services are used in conditional service requests to trigger the execution of an action list whenever an event of the given class is detected.

Information Service A service that returns information on monitored objects or the monitoring system, without modifying their state.

Manipulation Service A service that manipulates objects in the application or the monitoring system. These services usually only return an error status.

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 system 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 program execution, the underlying operating system, and the hardware 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 of the hardware/software instances which get the program running, i.e. the parallel programming library (e.g. PVM), the operating system, and the hardware. Access to specialized runtime libraries like e.g. PFSLib for parallel file access is controlled via monitor extensions.

Node Component of a multi-computer system that provides a single system image. Nodes may consist of serveral processors, however, observation and manipulation of these individual processors is usually not possible, since from an application's they are indistinguishable from each other.

OCM An **O**MIS **C**ompliant **M**onitoring system for a specific target platform.

OMIS The **O**n-line **M**onitoring **I**nterface **S**pecification.

Service The functions offered by an OMIS compliant monitoring system, i.e. the commands that can be invoked at the tool/monitor-interface. Services are the building blocks from which service requests are constructed.

Service Reply Data structure sent back from the monitoring systemrepresenting an answer to a service request.

Service Request String sent to the monitoring system requesting the execution of a set of services. See Conditional/Unconditional Service Request.

Target Architecture The system on which the monitoring system, the programs, and all programming libraries that are necessary for the application programs are running. It is composed of the hardware and the operating system.

Target Platform The system on which the monitoring system and the programs are running. It is composed of the hardware, the operating system, and all programming libraries that are necessary for the application programs.

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.

Unconditional Service Request A service request of an OMIS compliant monitoring system that is composed only from information and/or manipulation services. The services will be executed immediately and exactly once by the monitoring system. Thus, an unconditional service request will result in exactly one reply being returned.

History

Version 2.0: July 15, 1997

Current version. Published as a technical report at TUM [LWSB97]. Considerable changes were performed.

Version 1.0: February 1, 1996

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. The session was moderated by Roland Wismüller.

Kick-off meeting: July, 1995

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

Changes

Version 2.0 (July 15, 1997)

- OMIS is no longer oriented towards PVM. Instead it tries to cover all common message passing programming models without preferring one or the other. In future it will also be extended to work with shared memory systems.
- Several cooperations were started since OMIS 1.0 was released. We had intensive discussions with other researchers who would like to employ an OMIS compliant monitoring system for their own tool environments.
- Design and implementation of an OMIS compliant monitoring system (OCM) were started in 1996. The text gives some details.
- The system model was clarified with respect to target architecture and target platforms. Execution objects and node objects are defined more clearly (see Chapter 4).
- The technical terms synchronous and asynchronous service had been replaced (see Chapter 5).
- The technical terms information service, manipulation service, event service, conditional and unconditional request have been introduced (see Chapter 5).
- The list of objects was enhanced by threads (see Chapter 5).
- Examples in Chapter 5 were re-written according to the new syntax.
- Some technical terms have been renamed to make their intended meaning more clear and to avoid confusion (see Glossary).
- An extensive specification of the semantics of service requests and replies has been added to the document (see Chapter 7). This also includes the specification of error handling.
- The procedural interface has been extended and revised. It is more flexible and should be rather complete now (see Section 7.1).
- Syntax and semantics of service requests have been modified in order to simplify the usage of OMIS and to make it more general (see Section 7.2 to 7.2.4):
  - The OMIS 2.0 interface offers location transparency, i.e. services no longer have to be prefixed with a list of node numbers. Instead, services now get as a parameter a list of objects they shall work on. The nodes a service will be executed on are uniquely defined by this object list.
  - Objects like nodes, processes etc. are no longer addressed by some concrete ID (e.g. a PVM task ID), but by an abstract token generated by the monitoring system. In this way, objects can be identified in both a globally unique and platform independent way.

  - OMIS 2.0 now defines a hierarchy of application objects, that allows a conversion between the different object types, e.g. expanding a node token into a list of tokens for all processes on that node (see Section 7.2.2).
  - Tokens are now also used to identify monitor objects, especially conditional service requests. Thus, services no longer have to be preceded with a request ID. Instead, a request token is returned upon successful definition of conditional request. Requests containing actions modifying their own request (e.g. deleting it in order to implement a temporary breakpoint) are still possible by using a special event context parameter (see Section 7.2.4). Dependency cycles between conditional requests may also be broken via user defined events.
  - To support tools extracting large amounts of data from an application, a new data type "binary string" has been added to avoid the overhead of converting to/from an ASCII representation.
  - The specification no longer requires an atomic multicast protocol to be used for the distribution of requests touching multiple nodes. Instead, less restrictive requirements on the ordering of service execution have been specified (see Section 7.2.3). To enforce exclusive execution of a request, action lists can now be locked.
  - The specification on ordering constraints on the individual actions in an action list has been revised. There is no longer a parallel action lists where even actions on the same node may be executed in parallel or in any order different from the specified one. Instead, execution of services on the same node is now guaranteed to be sequential; parallelism can occur between nodes. The separator ';' can be used anywhere within an action list to denote a barrier-like synchronization (see Section 7.2.3). The separator ',' has been removed.
  - Event context parameters now are referenced by name rather than by number to increase ease-of-use.
  - The colon separating the event definition from the action list in a request now is even required for unconditional requests. This simplifies parsing considerably.

- The format of service replies has been changed completely. Instead of a single linear string that has to be parsed by the tool, a more structured representation is used now. A reply consist of a sequence of sub-replies, one for each service contained in the request. Each of these sub-replies is again a sequence of results, one for each object the service operated on. Identical results may be merged into a single entry (see Section 7.3).

- The way how the monitoring system connects to an application program has been changed in order to remove the previous dependency on the notion of a virtual machine (as in PVM). In OMIS 2.0, a tool must explicitly attach to all the nodes and processes it wants to be monitored. This scheme also implies that, if multiple tools connect to the same monitoring system, each of these tools has its own specific view of the observed system. In the course of this change, we also had to change the handling of creation and deletion of processes and nodes (see Section 8.1.1, 8.1.2, and 8.1.3).

- There is a large couple of changes in the description of the specific monitoring services:
  - The naming of services is more systematic now. A prefix indicates the type of object the service is working on. In addition, names are chosen in a way that indicates the kind of service (i.e. information service, manipulation service, event service).
  - The set of process services of OMIS 1.0 has been carefully analyzed and split into separate services for processes and threads. Thus, OMIS 2.0 can also be used for multithreaded systems.

  - Some parameter lists have been extended (e.g. block length and stride for memory accesses, length of stack backtrace, etc.).
  - Specifications are now more accurate and complete. This mostly concerns the services **node_get_info**, **proc_get_info**, and **thread_get_info**.
  - Finally, the set of services has been split into a basic set that is independent of the parallel programming library used (see Chapter 8), and a PVM extension providing services to support PVM (see Chapter 9).

- The sections on implementation concepts have been removed from this document. When our implementation of the OMIS compliant monitoring is finished, a design document will be prepared as a separate report.

Version 1.0 (February 1, 1996)

First version of the OMIS document.

Bibliography

[AHKL96] Georg Acher, Hermann Hellwagner, Wolfgang Karl, and Markus Leberecht. A PCI-SCI Bridge for Building a PC-Cluster with Distributed Shared Memory. In *Proceedings The Sixth International Workshop on SCI-based High-Performance Low-Cost Computing*, pages 1–8, Santa Clara, CA, September 1996. SCIzzL.

[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 *Lecture Notes in Computer Science*, 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.

[BLR96] L. Brunie, L. Lefevre, and O. Reymann. Execution Analysis of DSM Applications: A Distributed and Scalable Approach. In *SPDT'96, Symposium on Parallel and Distributed Tools*, pages 51–60. ACM Sigmetrics, may 1996.

[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 Conference on Vector and Parallel Processing, Zurich, Switzerland*, volume 457 of *Lecture Notes in Computer Science*, pages 756-765, Berlin, September 1990. Springer.

[Bod94] A. Bode. Parallel Program Analysis and Visualization. In J. Dongarra and B. Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 246-253. SIAM, 1994.

[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/pub/fgcs95.ps.gz`.

[Gei96] M. Geischeder. Development of the Program/Monitor-Interface for an OMIS Compliant Monitoring System. Master's thesis, Technische Universität München, München, October 1996.

[Gro95a] X/Open Group. *System Management: Universal Measurement Architecture Guide.* X/Open Company Linited, Reading, UK, 1995.

[Gro95b] X/Open Group. *Systems management: UMA Measurement Layer Interface (MLI).* X/Open Company Linited, Reading, UK, 1995.

[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.

[HKL97a] Hermann Hellwagner, Wolfgang Karl, and Markus Leberecht. Enabling a PC Cluster for High-Performance Computing. *SPEEDUP Journal*, 11(1), June 1997.

[HKL97b] Hermann Hellwagner, Wolfgang Karl, and Markus Leberecht. Fast Communication Mechanisms–Coupling Hardware Distributed Shared Memory and User-Level Messaging. In *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, pages 1294-1301, Las Vegas, Nevada, June 30–July 3 1997.

[Hoo96] R. Hood. The *p2d2* Project: Building a Portable Distributed Debugger. In *Proc. of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 127-136, Philadelphia, Pennsylvania, USA, May 1996. ACM Press.

[KCD+97] P. Kacsuk, J.C. Cunha, G. Dozsa, J. Lourenco, T. Antao, and T. Fadgyas. GRADE: A Graphical Development and Debugging Environment for Parallel Programs. *Parallel Computing Journal*, 22(13):1747-1770, February 1997.

[KDF96] P. Kacsuk, G. Dozsa, and T. Fadgyas. Designing Parallel Programs by the Graphical Language GRAPNEL. *Microprocessing and Microprogramming*, (41):625-643, 1996.

[KGV96] D. Kranzlmueller, S. Grabner, and J. Volkert. The Tools of the Monitoring And Debugging Environment. In *2nd European School of Computer Science, Parallel Programming Environments for High Performance Computing*, Institut IMAG-INRIA (Projet APACHE), 1996.

[Lam97] S. Lamberts. *Parallele verteilte Dateisysteme in Rechnernetzen*, volume 7 of *Research Report Series/Lehrstuhl fr Rechnertechnik und Rechnerorganisation (LRR-TUM), Technische Universitt Mnchen.* Shaker, Aachen, 1997.

[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/pub/europvm95.ps.gz`.

[LWOB97] T. Ludwig, R. Wismüller, M. Oberhuber, and A. Bode. An Open Interface for the On-Line Monitoring of Parallel and Distributed Programs. *Intl. Journal of Supercomputer Applications and High Performance Computing*, 11(2), to appear 1997.

[LWSB96] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. OMIS — On-line Monitoring Interface Specification (Version 1.0). Technical Report TUM-I9609, SFB-Bericht Nr. 342/05/96 A, Technische Universität München, Munich, Germany, February 1996. `http://wwwbode.informatik.tu-muenchen.de/~omis/OMIS/Version-1.0/version-1.0.ps.gz`.

[LWSB97] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. OMIS — On-line Monitoring Interface Specification (Version 2.0). Technical Report TUM-I9733, SFB-Bericht Nr. 342/22/97 A, Technische Universität München, Munich, Germany, July 1997. `http://wwwbode.informatik.tu-muenchen.de/~omis/OMIS/Version-2.0/version-2.0.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/pub/zeus95.ps.gz`.

[PSL96] Stefan Petri, Bettina Schnor, and Horst Langendörfer. ⟨PBeam – Fehlertoleranz für verteilte Anwendungen mittels Migration und Checkpointing. In Clemens H. Cap, editor, *Workstations und ihre Anwendungen, Proceedings der Fachtagung SI-WORK'96*, pages 91–102, Universität Zürich, Institut für Informatik, May 1996. vdf Hochschulverlag AG an der ETH Zürich.

[PSLS96] Stefan Petri, Bettina Schnor, Horst Langendörfer, and Jens Steinborn. Consistent Global Checkpoints for Distributed Applications on Clusters of Unix Workstations. In Herrmann G. Matthies and Josef Schüle, editors, *Paralleles und Verteiltes Rechnen – Beiträge zum 4. Workshop über Wissenschaftliches Rechnen*, pages 77–86, TU Braunschweig, October 1996. TU Braunschweig, Shaker Verlag.

[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.

[Ste96a] G. Stellner. CoCheck: Checkpointing and Process Migration for MPI. In *Proceedings of the International Parallel Processing Symposium*, pages 526–531, Honolulu, HI, April 1996. IEEE Computer Society Press, 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos,CA 90720-1264.

[Ste96b] G. Stellner. *Methoden zur Sicherungspunkterzeugung in parallelen und verteilen Systemen*, volume 2 of *Research Report Series/Lehrstuhl fr Rechnertechnik und Rechnerorganisation (LRR-TUM), Technische Universitt Mnchen*. Shaker, Aachen, 1996.

[Uem96] M. Uemminghaus. Communication and Request Processing in OMIS Compliant-Monitoring Systems. Master's thesis, Technische Universität München, München, October 1996.

[WOKH96] R. Wismüller, M. Oberhuber, J. Krammer, and O. Hansen. Interactive debugging and performance analysis of massively parallel applications. *Parallel Computing*, 22(3):415-442, March 1996.
`http://wwwbode.informatik.tu-muenchen.de/~wismuell/pub/pc95.ps.gz`.

[Zel96] H.-G. Zeller. Information and Event/Action Management in OMIS Compliant Monitoring Systems. Master's thesis, Technische Universität München, München, October 1996.