CrossGrid User Guide

Task 3.3.1: OCM-G

Task 3.3 – Grid Monitoring

Document Filename:	**cg-wp3.3-ocmg-devguide**
Workpackage:	**Task 3.3 – Grid Monitoring**
Partner(s):	**CYF, (TUM)**
Lead Partner:	**CYF**
Config ID:	**cg-wp3.3-ocmg-devguide-v0.1**
Document classification:	**PUBLIC**

Abstract: This is the developer's guide for the Grid application monitoring system OCM-G, developed within the CrossGrid project.

Delivery Slip

	Name	Partner	Date	Signature
From	Bartosz Baliś	CYFRONET AGH	Aug 2004	
Verified By				
Approved By				

Document Log

Version	Date	Summary of changes	Author
1-0	Jan 13th, 2005	First version	Roland Wismüller, Bartosz Baliś

Contents

Copyright Notice

This research is partly funded by the European Commission IST-2001-32243 Project CrossGrid.

1 About the software

The OCM-G (Grid-enabled OMIS-Compliant Monitor) is an application monitoring system which supports both cluster and Grid environments based on Globus 2.4, in particular the CrossGrid (and DataGrid) testbeds. The purpose of the OCM-G is to provide on-line information about a running parallel / distributed application to application-development-support tools, specifically performance analysis tools, like the G-PM tool. The information is obtained via a standardized interface OMIS.

This document is intended for tool developers who would like build tools for application development support compliant with the OCM-G and the OMIS specification.

For information concerning installation and running of the OCM-G, refer to **OCM-G Installation Guide**. Details concerning the using the OCM-G by application developers are covered in **OCM-G User Guide**.

Fig. 1.1 shows the run-time components of OCM-G, their distribution across the different kind of machines in the Grid, and their communication topology.

The OCM-G is a distributed system composed of three types of components.

1. Service Managers (SM) – one for each site of the target system. Started automatically on site's computing element (CE) machine.
2. Local Monitors (LM) – one for each host of the target system. Started automatically on worker nodes (WN) where there are application processes running.
3. Main Service Manager (MainSM) – one for each grid user in the system. Currently started by the user manually on the machine of his choice, usually the User Interface (UI) machine.

Fig. 1.1 shows the distribution of these components in a sample grid environment. Note that Local Monitors handle application processes running on Worker Nodes. The user can launch a tool, such as the G-PM performance analyzer for grid applications, which connects to the OCM-G via the MainSM.

Tools connect to the OCM-G and communicate with it via a standard interface OMIS (On-line Monitoring Interface Specification), which is described in this document.

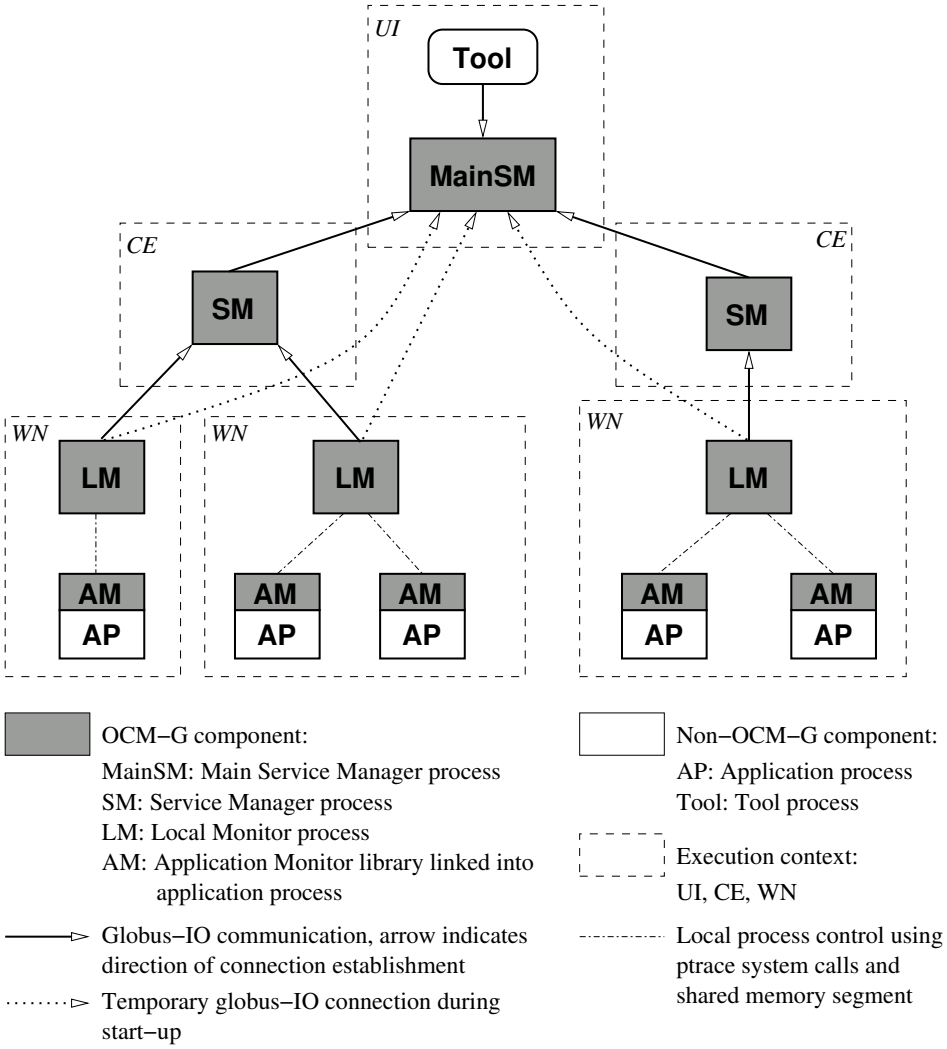


Figure 1.1: Components of OCM-G at run-time

2 Introduction

The OCM-G is based on the On-line Monitoring Interface Specification (OMIS) [1], a standardized specification for interface of communication between tools and on-line monitoring systems. Tool developers who would like to base their tools' functionality on the OCM-G need to know the following topics, which are covered by this document:

- The OMIS API to communicate with the monitoring system.

  This is covered in Chapters 3 and 4.

- The syntax and semantics of OMIS services to send monitoring requests to the OCM-G and handle replies incoming from the OCM-G.

  These services are specified in Chapters 5 to 9.

- The Specification of the G-PM Extension Interface in case the developer wants to extend OCM-G capabilities with new monitoring services.

  This is documented in Chapter 10.

Note that Chapters 3 to 5 are an updated excerpt from the original OMIS 2.0 specification [1]. In contrast to [1], this document contains only the specification of services actually implemented in OCM-G.

3 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 4.1). 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, 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 3.2 provides a basic outline of the mechanisms and the syntax used in this interface. Section 3.3 presents two examples giving an impression of the interface's expressiveness. A detailed description of the interface procedures is contained in Section 3.4. In section 3.5 we discuss the the scope of tools, i.e the set of objects that they control. Section 3.6 finally presents some additional remarks on the tool/monitor-interface.

3.1 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".

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

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 Chapters 5 to 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.

- 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, six different types are supported by the basic services:

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

  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.

  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).

3.2 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 4. The discussion here will not cover all relevant aspects. Instead it will try to introduce the concepts that will be used.

3.2.1 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* | *identifier* '<' *index* '>'
index ::= *identifier* | *ev_ctx_param*
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 4.2.2. 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. OMIS also supports tokens, which identify arrays. The index, which again is a token, is enclosed in '<' and '>'. 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.

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

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.

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 4.2.2.1.

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.

3.2.2 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 4.2.3.

[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.

3.2.3 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 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 4.3.

3.3 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.

3.3.1 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 4.2.2.1.

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.

3.3.2 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.

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.

[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.

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.

3.4 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 4.1.

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 4.1.

3.5 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 5.1.3 and 5.1.2.

3.6 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 5.2.1.1, 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.

4 General Description

4.1 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_fds**: Auxiliary function: returns the file descriptors 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 5 to 9 specify in detail the requests accepted by **omis_request** and the resulting replies.

4.1.1 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.

4.1.2 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.

4.1.3 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 4.3.1.

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 4.3 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 >>                  }
  ...                                      ...
}                                        }
```

4.1.4 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**.

4.1.5 omis_fds

```
int *omis_fds(int *num)
```

omis_fds returns an array of file descriptors that can be used to block a tool process until a message from the monitoring system arrives, e.g. by passing the file descriptors to a **select** system call. When a message arrives on one of the file descriptors, `omis_handler` has to be called to handle that message.

If the function cannot execute correctly, it returns NULL, and the result returned in `*num` is undefined. Otherwise, it returns a dynamically allocated array of file descriptors which should be freed after usage. In this case, `*num` returns the number of entries in the array.

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.

4.1.6 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_fds**) 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_fds** or **omis_handler**.

The following two examples show the proper use of **omis_fds** 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_fds;
   int num_fds;
   int i;
   char input_buf[80];

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

   /* Get OMIS file descriptors */
   mon_fds = omis_fds(&num_fds);

   do {
      /* Build file descriptor set containing the OMIS file
         descriptor and the stdin file descriptor */
      FD_ZERO(&fds);
      for (i=0; i<num_fds; i++)
           FD_SET(mon_fds[i],&fds);
      FD_SET(0,&fds);

      /* Block until there is some input */
      select(FD_SETSIZE, &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"));

    free(mon_fds);
    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 *mon_fds;
    int num_fds;
    int i;
    XtAppContext app;

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

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

    /* Add OMIS file descriptors as an additional input source
       to the X window main loop. */
    mon_fds = omis_fds(&num_fds);
    for (i=0; i<num_fds; i++) {
XtAppAddInput(app, mon_fds[i],
  (XtPointer)XtInputReadMask,
  (XtInputCallbackProc)cb, NULL);
}
    free(mon_fds);

    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 **XtAppMainLoop** never returns, some X window call-back function must invoke **omis_finalize** before the tool exits.

4.2 Service Requests

4.2.1 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* | *identifier* '<' *index* '>'
index ::= *identifier* | *ev_ctx_param*
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 4.2.4 and a set of event specific parameters for each event service which is defined in Chapters 5 to 9.

The semantics of the different data types used in the request language is specified in Section 4.2.2. The semantics of the input *parameter_list* of a service depends on the concrete service and is specified in Chapters 5 to 9.

4.2.2 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 4.2.1.

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.

4.2.2.1 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 nine basic classes of objects:

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

Objects belonging to the first six 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. 4.1. 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.

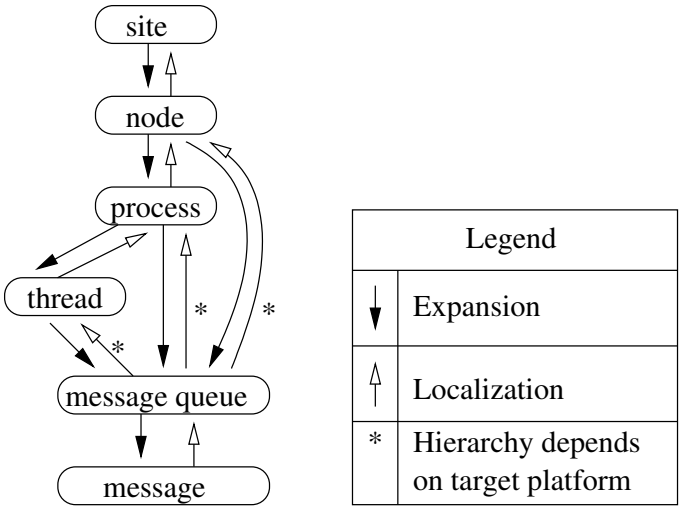


Figure 4.1: Object hierarchy in OMIS

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. 4.1 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 4.2.3.3) 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:

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

[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.

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_'.

4.2.3 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.

4.2.3.1 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 4.2.2.1). In this situation, it is undefined whether the service will be executed once or twice for that object.

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.

[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

4.2.3.2 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 4.2.4 and in the specifications of the individual event services.
   (c) The request's action list is executed as described in the previous subsection.

4.2.3.3 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).

4.2.4 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).

4.3 Service Replies

4.3.1 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 4.2, the semantics of the **result** strings depends on the service and is specified in Chapters 5 to 9.

4.3.2 String Syntax

The syntax of the strings pointed to by the **obj_list** and **result** fields of an **Omis_object_result** strucure (see Section 4.3.1) 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.

4.3.3 Reply Examples

Fig. 4.2 to Fig. 4.4 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. 4.2. 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. 4.3 shows how a reply may look like when errors occur.

Now, consider a conditional request:

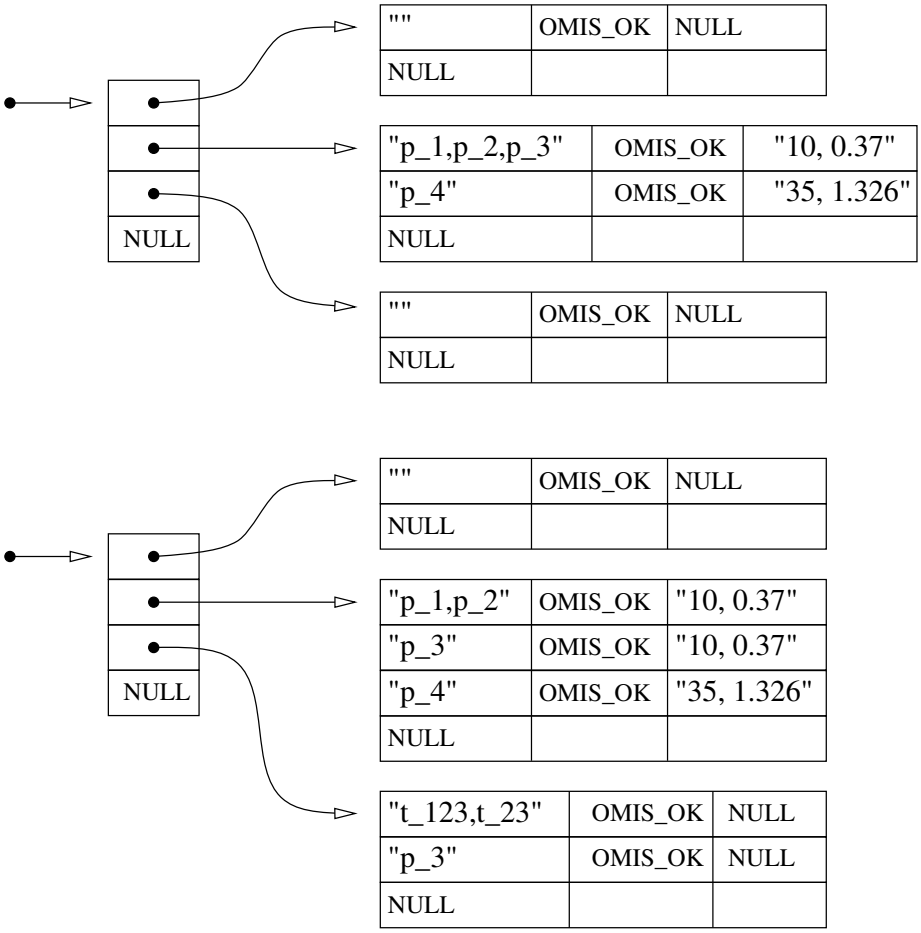


Figure 4.2: Two possible replies for an unconditional request without errors

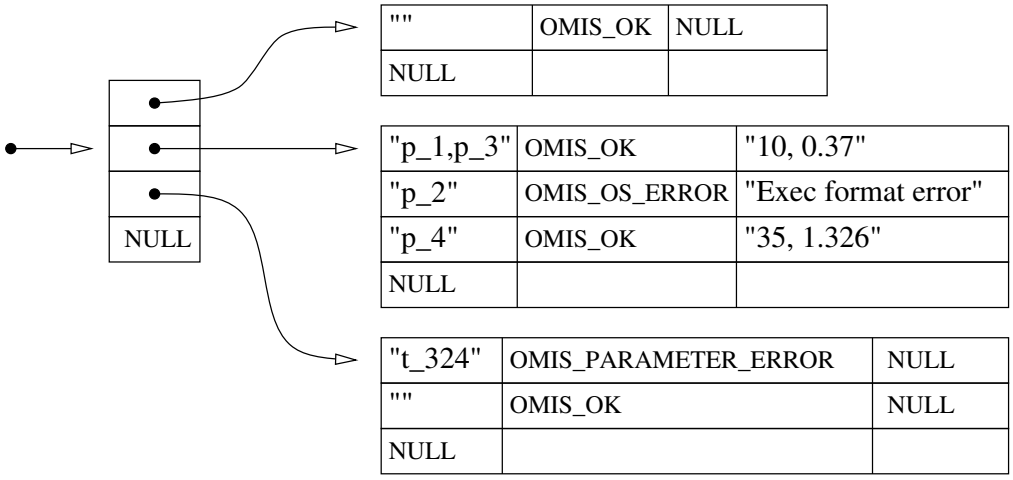


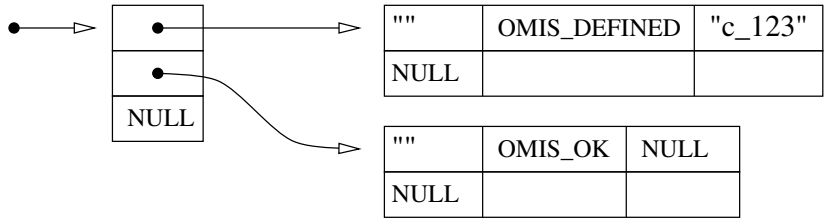
Figure 4.3: Reply for an unconditional request with errors

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

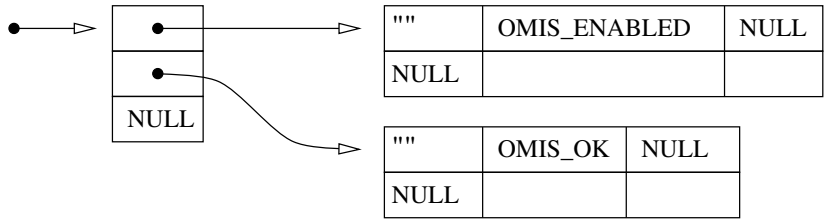
When this request is passed to the monitoring system, three different kinds of replies may be received as shown in Fig. 4.4. The reply generated when the event is detected indicates the object (i.e. process in this example) where the event occurred in the **obj_list** component of the reply's first **Omis_service_result** structure.

When the tool now attaches to a new process, say **p_32**, a reply as shown in Fig. 4.5 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.

a) reply for definition:



b) reply for enable (analogous for disable/delete):



c) reply when event has been detected:

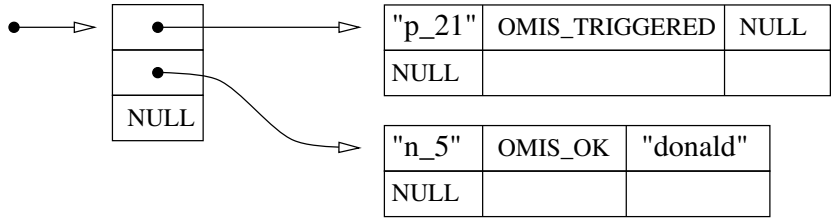


Figure 4.4: Replies for a conditional request

4.4 Description Method for Services

Throughout the service descriptions in Chapters 5 to 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*,

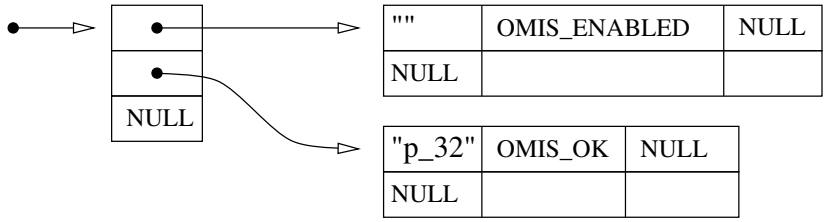


Figure 4.5: Reply for a conditional request when new object is monitored

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 ']').

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)
```

A valid reply could look as shown in Fig. 4.6.

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);
```

[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.

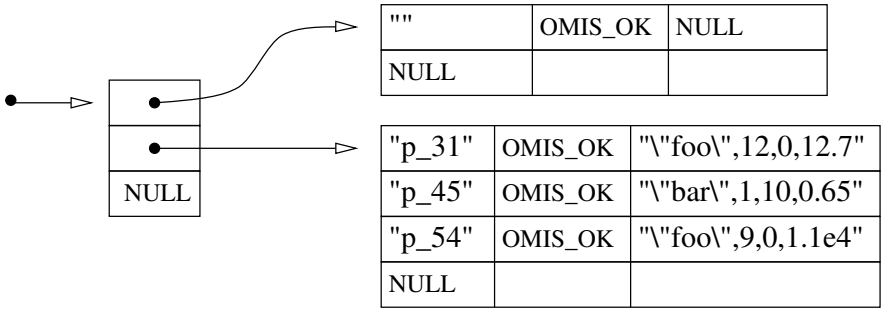


Figure 4.6: Reply example

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. 4.7.

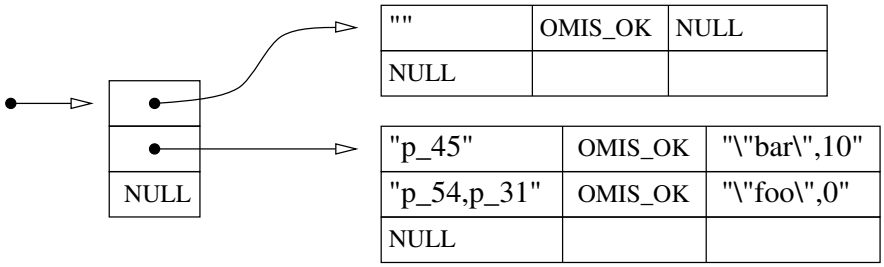


Figure 4.7: Reply example

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:

```
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.

5 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.

5.1 System Objects

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

- sites (Section 5.1.1),
- nodes (Section 5.1.2),
- processes (Section 5.1.3),
- threads (Section 5.1.4).

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.

5.1.1 Sites

Sites are introduced in the current version of OMIS to support georgraphically distributed systems which are composed of computing clusters located at multiple sites, like e.g. in a Grid environment. A site represents a collection of nodes, usually within a single administrative domain. E.g. in a typical Grid system, a site would be a cluster of machines representing a single computational resource in the Grid.

Support for site objects is optional, i.e. an OMIS compliant monitoring system does not need to provide them.

5.1.1.1 Manipulation Services

The manipulation services for site objects are for attaching or detaching sites to/from the scope of objects to be monitored.

1. **site_attach** ..... attach to a site. (O)

```
void
site_attach(token* site_list)
```

   Calling this service results in the monitoring system attaching itself to the sites specified in **site_list**. The site token(s) required as an argument may be obtained from the output of information services from some OMIS extensions supporting a specific target platform.

2. **site_detach** ..... detach from a site. (O)

```
void
site_detach(token* site_list)
```

   Calling this service results in the monitoring system detaching itself from the site(s) specified in **site_list** and in turn also from all nodes and processes located on these sites. 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 **site_attach** for a detached site will result in an error.

5.1.1.2 Information Services

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

1. **site_get_info** ..... Return information on a site. (O)

```
typedef struct {
  [string   name;]            // Name of this site. Usually, this is
                              // a host name representing the site.
                              // Present if Bit 0 is set in flags.
} Site_info;

Site_info
site_get_info(token* site_list, integer flags)
```

   As with the other information services, the bit-vector **flags** defines which kind of information has to be retrieved. By calling **site_get_info([],0)**, the tokens for all sites currently observed by the monitoring system (i.e. all attached sites) can be retrieved.

5.1.2 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.

5.1.2.1 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.

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.

5.1.2.2 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 {
                              // 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)
}
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.

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.

5.1.3 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 4.2.2 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.

5.1.3.1 Manipulation Services

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

1. **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.

2. **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.

3. **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.

4. **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**.

5.1.3.2 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
} 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
   [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
} 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_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.

5.1.3.3 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).

5.1.4 Threads

Threads are the only objects in the monitored system that actually execute code. See Section 5.1.3 for a discussion of the relation between threads and processes.

5.1.4.1 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_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.

2. **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.

3. **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.

4. **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.

5. **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.

6. **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.

5.1.4.2 Information Services

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

1. **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.

5.1.4.3 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_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.

2. **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.

3. **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.

4. **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).

5. **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).

6. **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.

7. **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.

8. **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.

9. **thread_executes_probe** ..... A thread executes an application-specific instrumentation (probe). (O)

```
void
thread_executes_probe(token* thread_list, string probe_name)
--> struct {
   any par1;      // the event context parameters are specific
   any par2;      // to the instrumentation point
   ...
}
```

This event is raised whenever a thread in **thread_list** is executing the application-specific instrumentation point (probe) specified by **probe_name**.

This document does not specify *how* this instrumentation is done. It only requires that each instrumentation point has a unique name. One possible approach is to insert special function calls into the application code, which then raise this event. The event context parameters returned by the event may then just be the parameters passed to the special function call.

5.2 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 5.2.2.

5.2.1 Conditional Service Requests

5.2.1.1 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**.

5.2.2 Miscellaneous

5.2.2.1 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. **mon_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. **mon_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. **mon_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.

6 Specification of the Generic Application Extension

This extension provides a generic means to refer to an arbitrary subset of processes as one application entity. It introduces a new object class for applications and services to get information on an application. The constructor of application objects identifies the application by a string. This document does not specify how the connection between this string and the application's processes is established technically. One way to do this is to pass the application name to each process when it starts up.

6.1 Data Types

The extension introduces a new token class: application tokens. The prefix 'app_' is used for these tokens. In order not to break the hierarchy of system objects (each object class should have only one parent class), application tokens are not subject to expansion or localization.

6.2 System Objects

6.2.1 Manipulation Services

1. **app_attach2** ..... attach to an application. (O)

```
token
app_attach2(String appname)
```

Calling this service results in the monitoring system attaching itself to the application specified by its name **appname**. Note that this service just is a constructor for an application token, it does not imply that the monitoring system attaches to any process or thread of the application, nor to any site or node used by it. The service returns the token of the application which can be later used to obtain the list of the application's processes using the **app_get_proclist** service.

6.2.2 Information Services

Currently, the only information service related to applications is **app_get_proclist** which returns the list of all processes of the application.

1. **app_get_proclist** ..... Return the process list of an application. (P)

```
typedef struct {
   token site;      // The site token
   token node;      // The node token
   token proc;      // The process token
} procinfo;

struct {
   integer   num_procs; // Number of elements in the following list
                        // i.e. number of processes in the application
   procinfo* procs;     // List of site/node/process tokens
}
app_get_proclist(Token app)
```

Calling this service results in the monitoring system returning information on the processes of the application specified by the token **app**. For each process of the application three tokens are returned: site token, node token, and process token.

7 Specification of the Performance Analysis Extension

7.1 Data Types

This extension defines two new token classes useful for performance analysis tools: counters and integrators.

7.1.1 Counters

Counters are essentially plain `integer` variables that can just be incremented, read, and reset. They are intended to be used mainly for measuring values like e.g. a message count (by incrementing the counter by one for each message event) or a message volume (by incrementing the counter by the message length).

Usually, to obtain a meaningful measurement of a value changing with time, we not only need the measurement value, but also the time stamp indicating when the measurement was performed. Experience with on-line performance analysis tools showed that in addition to this time stamp, a second one is useful, which provides the time of the beginning of the current measurement. In this way, a counter value is accompanied by the time interval during which a measurement was carried out. Thus, when a counter is read, the following elements are returned:

1. `value` — the current value of the counter,
2. `time` — the current time, i.e. the end time of the measurement interval,
3. `start_time` — the start time of the measurement interval.

See the description of the read services below.

7.1.2 Integrators

Integrators be used for storing `floating` values. They are intended to support the measurement of the duration of some activity, e.g. the time spent in library calls for sending or receiving messages. As the name implies, what they actually measure is an *integral* of a function which is constant over some interval. An integrator conceptually consists of three variables:

1. `current_value` — the current value of the function,
2. `last_time` — time of the last update of the integrator,
3. `integral_value` — the function's integral value at the time of the last update.

All three variables are initially set to 0.

There is a method to update the integrator, which takes a **time** and an **increment** as its parameters. It performs the following computation:

1. `integral_value += current_value * (time - last_time)`
2. `last_time = time`

3. `current_value += increment`

The duration of some activity can thus be measured in the following way:

- at the beginning of the activity: update the integrator with an **increment** of 1,
- at the end of the activity: update the integrator with an **increment** of -1.

This seemingly complicated procedure makes it possible to correctly measure the duration for concurrent, overlapping activities, too.

When an integrator is read at a specific **time**, the following elements are returned:

1. `integral_value + current_value * (time - last_time)` — the current value of the function's integral,
2. `current_value` — the current value of the function,
3. `time` — the current time, i.e. the end time of the measurement interval,
4. `start_time` — the start time of the measurement interval.

See the description of the read services below.

7.1.3 Global and local objects

There are several variants of counters or integrators:

- local counters/integrators,
- global counters/integrators,
- arrays of local counters/integrators.

The **local** objects are accessible only on the node where they have been created.

In contrast, the **global** objects can be accessed from any node in the target system. The specification strongly recommends that these objects are implemented in a scalable, distributed way (e.g. by creating a local counter/integrator on each node and returning a properly aggregated global value when the counter/integrator is read).

In addition, **arrays** of local objects are provided. These arrays can be indexed with node, process, or thread tokens. Array elements are allocated on demand: when the array is indexed with some token for the first time, a local counter or integrator is created on the token's local node and is added to the array. Arrays of counters / integrators support the comfortable performance measurement of several processes, using a minimal number of requests.

7.1.4 Token prefixes

The performance analysis extension uses the following prefixes for the different kinds of objects:

	local	global	array
counter	`pa_cl`	`pa_c`	`pa_ac`
integrator	`pa_il`	`pa_i`	`pa_ai`

7.2 Services

7.2.1 Local counters

7.2.1.1 Manipulation Services

1. **pa_counter_local_create** ..... create a new local counter. (R)

```
token
pa_counter_local_create(token* objlist)
```

This service creates a new local counter for each object specified in **objlist**. The local counter created is only accessible on the node where it was created. **objlist** typically contains a list of node or process tokens. Token expansion is performed only down to the node level, i.e., nodes are not expanded to processes, so only one local counter is created for each node token passed.

The counter value of all counters created are initialised with 0.

2. **pa_counter_local_delete** ..... delete a local counter. (R)

```
void
pa_counter_local_delete(token* counter_list)
```

This service is used to delete the local counters specified in **counter_list**, formerly created with **pa_counter_local_create**.

3. **pa_counter_local_attach** ..... attach to a local counter. (O)

```
void
pa_counter_local_attach(token* counter_list)
```

The **pa_counter_local_attach** service attaches the local counters specified in **counter_list**, so that they can be used in the current OMIS session.

Using this service is normally not needed, since the local counters are attached upon creation by **pa_counter_local_create**. Its only use is when passing tokens between two tools.

4. **pa_counter_local_detach** ..... detach from a local counter. (O)

```
void
pa_counter_local_detach(token* counter_list)
```

This service detaches the tokens specified in **counter_list**, which have been formerly attached by **pa_counter_local_attach**.

The only use of this service is when passing tokens between two tools.

5. **pa_counter_local_increment** ..... increment a local counter. (R)

```
void
pa_counter_local_increment(token counter, integer increment)
```

This service increments the value of the local counter passed in via **counter** by the value specified in **increment**.

7.2.1.2 Information Services

1. **pa_counter_local_read** ..... return the value of a local counter. (R)

```
struct {
  floating value;       // return value, see below
  floating time;        // time when counter was read
  floating start_time;  // start time of measurement, see below
}
pa_counter_local_read(token* counter_list, integer flags)
```

This service returns the current values of the local counters that are specified in **counter_list**. If Bit 0 is set in **flags**, the counters are reset to 0 after thay have been read and the **start_time** returned for a counter is the time when the previous read operation has been performed on this counter. Otherwise, **start_time** is the time when the counter has been read first. If Bit 1 is set in **flags**, the return value is the counter's value divided by (**time** - **start_time**), otherwise, it is just the counter's value.

7.2.2 Local integrators

The following section describes the services that operate on local intergrators.

7.2.2.1 Manipulation Services

1. **pa_integrator_local_create** ..... create a new local integrator. (R)
   pa_integrator_local_delete ..... delete a local integrator. (R)
   pa_integrator_local_attach ..... attach to a local integrator. (O)
   pa_integrator_local_detach ..... detach from a local integrator. (O)

```
token
pa_integrator_local_create(token* objlist)

void
pa_integrator_local_delete(token* integrator_list)

void
pa_integrator_local_attach(token* integrator_list)

void
pa_integrator_local_detach(token* integrator_list)
```

These services work analogously to their local counter counterparts.

2. **pa_integrator_local_increment** ..... increment a local integrator. (R)

```
void
pa_integrator_local_increment(token integrator,
                              floating increment,
                              floating time)
```

This service updates the local integrator passed in via **integrator**. With the help of the **time** parameter, as well as that of internal variables of the integrator, a new integral value of the integrator

is calculated. **increment** is then added to the internal value of the integrated function, and **time** replaces the internal integrator time.

Typically, **increment** is set to 1 at the beginning of an event and to −1 at the end, while **time** is set to event context parameter **$time**.

7.2.2.2 Information Services

1. **pa_integrator_local_read** ..... return the state of a local integrator. (R)

```
struct {
  floating integral;
  floating value;
}
pa_integrator_local_read(token* integrator_list, floating time,
                         integer clear)
```

This service returns the current state of the local integrators specified in **integrator_list**. It returns the current **integral** value, as well as the current **value** of the integrated function.

time is used to calculate the integral value if **pa_integrator_local_read** was called in the middle of an event (i.e. when the value of the integrated function is nonzero). When **time** is 0, which is usually the case, the current time is used.

If **clear** is nonzero, the integral values of integrators are reset to 0 after reading.

7.2.3 Global counters

The following section describes the services that operate on global counters.

7.2.3.1 Manipulation Services

1. **pa_counter_global_create** ..... create a new global counter. (R)

```
token
pa_counter_global_create()
```

This service creates a new global counter.

The created counter is 0–initialised.

2. **pa_counter_global_delete** ..... delete a global counter. (R)
   pa_counter_global_increment ..... increment a global counter. (R)

```
void
pa_counter_global_delete(token* counter_list)

void
pa_counter_global_increment(token counter, integer increment)
```

These services work analogously to their local counter counterparts.

7.2.3.2 Information Services

1. **pa_counter_global_read** ..... return the value of a global counter. (R)

```
integer
pa_counter_global_read(token* counter_list, integer clear)
```

This service works analogously to **pa_counter_local_read**.

7.2.4 Global integrators

The following section describes the services that operate on global integrators.

7.2.4.1 Manipulation Services

1. **pa_integrator_global_create** ..... create a new global integrator. (R)
   pa_integrator_global_delete ..... delete a global integrator. (R)
   pa_integrator_global_increment ..... increment a global integrator. (R)

```
token
pa_integrator_global_create()

void
pa_integrator_global_delete(token* integrator_list)

void
pa_integrator_global_increment(token integrator,
                               floating increment,
                               floating time)
```

These services work analogously to their global counter and local integrator counterparts.

7.2.4.2 Information Services

1. **pa_integrator_global_read** ..... return the state of a global integrator. (R)

```
struct {
  floating integral;
  floating value;
}
pa_integrator_global_read(token* integrator_list,
                          floating time, integer clear)
```

This service works analogously to **pa_integrator_local_read**.

7.2.5 Arrays of counters

The following section describes the services that operate on arrays of counters. Actually, in this case only a creation service is needed, since the services for local counters (see Section 7.2.1) are used to operate on arrays.

7.2.5.1 Manipulation Services

1. **pa_ac_create** ..... create a new array of local counters. (R)

```
token
pa_ac_create()
```

This service creates a new array of local counters.

7.2.6 Arrays of integrators

The following section describes the services that operate on arrays of integrators. Actually, in this case only a creation service is needed, since the services for local integrators (see Section 7.2.2) are used to operate on arrays.

7.2.6.1 Manipulation Services

1. **pa_ai_create** ..... create a new array of local integrators. (R)

```
token
pa_ai_create()
```

This service creates a new array of local integrators.

8 Specification of the Symbol Table Extension

8.1 Introduction

This extension provides some information on a process's symbol table. Currently, it just can deliver information on text symbols (functions).

8.2 Services

8.2.1 Information Services

1. **symtab_proc_get_functions** ..... Return a list of a process's functions. (R)

```
struct {
   string    file_name;     // "GLOBAL" for global functions,
                            // name of the object file for static ones
   string    function_name; // the function's name
   integer   start_address; // the function's start address
   integer   end_address;   // the function's end address
} function_info;
function_info *
symtab_proc_get_functions(token* proc_list)
```

   This service returns a list of function names, together with their start and end addresses for each process in **proc_list**.

2. **symtab_proc_get_function_addr** ..... Return the address of a process's function. (R)

```
struct {
   integer   start_address; // the function's start address
   integer   end_address;   // the function's end address
}
symtab_proc_get_function_addr(token* proc_list, string name)
```

   This service returns a the start and end address of the function with the specified **name** for each process in **proc_list**.

9 Specification of the G-PM Extension

9.1 Introduction

Generic performance analysis services are provided within the Performance Analysis Extension which is part of the OMIS specification [1]. This section presents the additional services which were designed to satisfy some specific features of the G-PM performance tool[1].

The features provided by these services allow to limit the scope of measurements in the G-PM to a user-specified subset of the *functions* in his application. Currently the base OMIS specification does not support the notion of the application functions as objects.

Another feature that cannot be specified using OMIS services is the notion of *partner objects*. Partner objects are secondary objects involved in some operations such as communication calls. In communication, always at least two parties are involved: a primary object and a secondary object. From the point of view of send operations, the primary object is the sender, while the secondary (partner) object is the receiver. For receive operations this is the other way round.

Original OMIS services for detecting start and end of function calls can be parametrized, as all OMIS services, with primary objects, for example we can limit the detection `MPI_Send` start event to particular processes. With partner objects we can specify events such as: *start of MPI_Send in process p1 but only if the message destination is process with rank 2.*

In general, when monitoring the communication calls, the services described in below allow to limit the scope of event detection to just some destination processes in case of the messages sent or to some source processes in case of the messages received.

9.2 Services

9.2.0.1 Event services

1. **patop_lib_call_started** ..... A thread has invoked a communication library call. (R)

   patop_lib_call_ended ..... A thread has returned from a communication library call. (R)

```
void
patop_lib_call_started(token* thread_list,
                       string lib_call_name,
                       integer* tid_list, integer* func_list)
--> struct {
   integer len;
   integer potential;
}

void
patop_lib_call_ended(token* thread_list, string lib_call_name,
                     integer* tid_list, integer* func_list)
```

[1] These services were actually designed for the predecessor of the G-PM tool, PATOP, hence the prefix of the services' names

```
--> struct {
   integer len;
}
```

These services are filters for the basic OMIS services **thread_has_started_lib_call** and **thread_has_ended_lib_call**. They accept the same parameters as the standard services, plus two new ones.

- **tid_list** is a list of *partner objects*, which are actually process identifiers. The events are triggered by some calls to communication routines only if objects mentioned in **tid_list** are involved in the communication as partner objects, i.e. are destination processes for send operations or source processes for receive ones. An empty list matches all processes. Note that numbers in **tid_list** are meant as task identifiers from the target parallel environment, such as MPI ranks, not OMIS tokens.
- **func_list** is a list of *function names*. The event is triggered only when it occurred within a function from the list. An empty list matches all functions. The list consists of pairs of addresses: the beginning and the end of the code of the function.

For convenience, the services provide a new Event Context Parameter **$len** that contains the message length. Otherwise this value could be extracted from parameters of communication function calls, but this depends on the particular functions. If the message length is unknown, **$len** is set to 0.

At the start of the message receive call, the sending process might not be known. For this reason, **patop_lib_call_started** provides an additional Event Context Parameter: **$potential**. It is set to 0 when the event is *known* to be of interest, and to 1 if it only *might* concern the monitoring tool. Whether the event is actually of concern is not known until the end of the communication call. Therefore, these two cases can be distinguished by **patop_lib_call_ended**: if it triggers, then the event was actually of interest, otherwise not. See also **patop_integrator_global_increment** and **patop_integrator_local_increment**.

9.2.0.2 Manipulation services

1. **patop_integrator_global_increment** ..... increment a global integrator. (R)
   patop_integrator_local_increment ..... increment a local integrator. (R)

```
void
patop_integrator_global_increment(token integrator,
                                  floating increment,
                                  floating time,
                                  integer potential)

void
patop_integrator_local_increment(token integrator,
                                 floating increment,
                                 floating time,
                                 integer potential)
```

These services act as wrappers for **pa_integrator_global_increment** and **pa_integrator_local_increment**, and are specially designed to work together with **patop_lib_call_started** and **patop_lib_call_ended**.

They accept one additional parameter: **potential**. This parameter should be set to 1 at the beginning of an event when it is not certain if the event should be recorded. In all the other cases it must set to 0.

If a potential beginning of an event was found to be of no interest, then these services must not be called again at the end of the event at all. If the event was found to be of interest, then these services should be called again at the end, just as normally.

A number of restrictions are in place:

(a) For any given process and any given integrator, only one potential beginning at a time is supported.

(b) **increment** must be less than 0 at the event's end.

10 The OCM-G Extension Interface

10.1 The C-Interface for Services

This section specifies, how new OMIS services can be added to OCM-G. The basic idea how extending OCM-G works is as follows:

1. For each extension, a registry file has to be provided that specifies the new services and their relevant attributes.
2. For each service in the extension, one or more C functions have to be written. The functions for all services of the extension are collected into a single extension library. A translator creates prototype definitions for these functions from the registry file. In addition, it creates some code to be included in the extension library.
3. This library is then linked to the OCM-G using a special tool.

In the following subsections we specify the syntax of the registry file, the interfaces for the C-functions implementing new services, and how these functions can efficiently invoke already existing services.

10.1.1 Mapping of OMIS Data Types

Since OMIS services have to be implemented by C-functions, there must be a mapping between the OMIS data types and C data types. This mapping is specified in Table 10.1.

OMIS data type	C data type
integer	`long`
floating	`double`
string	`char *`
token	`Ocm_obj_token`
binary	`Ocm_bin_string`
any	`Ocm_any_val`

Table 10.1: Mapping between OMIS data types and C data types

`Ocm_obj_token` is an opaque data type as specified in Section 4.2.2.1. The other types are defined as follows:

```
typedef enum {
    Ocm_any,
    Ocm_integer,
    Ocm_floating,
    Ocm_string,
    Ocm_binary,
    Ocm_token,
    Ocm_list
} Omis_param_type;

typedef struct ocm_list {
```

```
    Omis_param_type type;       /* The type of the elements */
    int length;                 /* Number of elements */
    union {
        long             *i;    /* type == Ocm_integer */
        double           *f;    /* type == Ocm_floating */
        char            **s;    /* type == Ocm_string */
        Ocm_bin_string   *b;    /* type == Ocm_binary */
        Ocm_obj_token    *t;    /* type == Ocm_token */
        struct ocm_list  *l;    /* type == Ocm_list */
        struct ocm_any   *a;    /* type == Ocm_any */
    } value;
} Ocm_list_val;

typedef struct ocm_any {
    Omis_param_type type;       /* The type of this value */
    union {
        long              i;    /* type == Ocm_integer */
        double            f;    /* type == Ocm_floating */
        char             *s;    /* type == Ocm_string */
        Ocm_bin_string    b;    /* type == Ocm_binary */
        Ocm_obj_token     t;    /* type == Ocm_token */
        Ocm_list_val      l;    /* type == Ocm_list */
    } value;
} Ocm_any_val;
```

Lists are passed in a different way, depending on their nesting level. A parameter of a service, which has a list type, is passed to the C function implementing that service as two parameters: An `int` parameter specifying the length of the list, and a second parameter being an array of the C type corresponding to the list's element type. As an example, consider the `thread_send_signal` service, which has the following parameter list:

```
token* threads, integer sig
```

This is mapped to the following parameters

```
int num_threads, Ocm_obj_token threads[], long sig
```

which are passed to the C function implementing the service. List that are nested inside lists, or are passed to a parameter of type *any*, are mapped to an `Ocm_list_val` structure. Thus, a list of lists of integers would be mapped like this:

```
..., int num_lists, Ocm_list_val lists[], ...
```

Each element in `lists` has `type == Ocm_integer` and `value.i` pointing to an array of integer values. This different handling of lists is done in order to have a simple interface for the most common case (i.e. non-nested lists), without giving up the ability to handle the complex cases, too.

The OCM-Gperforms parameter type checking before passing values to the C functions implementing the services, so there is no need for a service function to check the `type` field in an `Ocm_list_val` structure. In case of the `Ocm_any_val` structure, however, the `type` field is essential, since the type is not statically known.

10.1.2 The Execution Context

In OCM-G, a service can be executed in the context of different kinds of processes:

- The Main Service Manager, i.e. the central process of the monitoring system,
- the Service Manager running on each site of the target system,
- the Local Monitor running on each host of the target system,
- the Application Module, which is included in the monitored application processes.

Where a service is executed depends on (1) the 'context' attribute in the registry file (see Section 10.1.4, and (2) the localization of the objects the service is invoked on.

10.1.3 Invoking Services From Within Other Services

The services implemented in an extension will often be based on already existing services (either a basic service or a service defined by another extension). So there should be mechanisms for calling other services from within a service. OCM-G defines two such mechanisms. One handles the most simple, but also most common case, where you just want to call an information or manipulation service. The other mechanism handles the rare cases where you want to submit a complete service request from within a service. These two mechanisms are explained in the next subsections.

There is also a mechanism that allows events to be based on other events. This is explained in Section 10.1.4.4.

10.1.3.1 Local Information/Manipulation Services

An information or manipulation service that can be executed locally, can be invoked simply by calling a C function associated with that service. The exact condition for the feasibility of local execution is that the called service must have the same execution context as the calling one, and must be invoked only on objects located on the local node. Note that the latter condition is always true when the service is invoked on objects passed to the calling service.

The C prototype for the function associated with an action (i.e. an information or manipulation service) is as follows:

```
int ocm_action_<service_name>(Ocm_expansion_context context,
                              Omis_service_result *result,
                              <service_parameters>);
```

Here `<service_name>` refers to the name by which the service is known at the OMIS tool/monitor-interface, `<service_parameters>` must be replaced by the proper transcription of the service's input parameters (see Section 10.1.1). Object tokens passed to the called service need not match the token class the service operates on; they will be converted as described in Section 4.2.2.

The `context` parameter specifies an internal execution context of the OCM-G. It should not be changed or interpreted by the action. It is provided only because it needs to be passed to all services called by the current one.

The `ocm_action_<service_name>` function returns two results: The function's return value indicates the presence of errors during its execution. If the return value is ≥ 0, no errors occurred; if it is < 0, there has been an error for at least one of the objects the service operated on.

More detailed results and/or error descriptions, if any, are returned via the `result` parameter. When a non-`NULL` value is returned, this is a pointer to a dynamically allocated array of `Omis_object_result`

structures, as defined in Section 4.3. When a `NULL` value is returned, there is either no real result of the invoked service, which is usually the case with manipulation services, or an out-of-memory error occurred, preventing the function from allocating an `Omis_object_result` structure. These cases can be distinguished by examining the function's return value.

Section 10.1.4.3 provides a small example showing how to call actions using this interface.

10.1.3.2 Arbitrary Requests

To submit arbitrary service requests to the monitoring system, a function implementing a service can call a special version of `omis_request`, named `omis_request_context`. It works exactly like `omis_request`, with the only exception that it accepts an additional argument to pass the `Ocm_expansion_context` of the calling service:

```
Omis_reply
omis_request_context(Ocm_expansion_context context,
                     const char * request,
                     void (* callback)(Omis_reply reply, void * param),
                     void * param,
                     Omis_flags flags)
```

10.1.4 Adding New Services

10.1.4.1 The Registry File

Each extension of the OCM-G is described by a registry file specifying the services and tokens provided by that extension. In this subsection, however, we are only specifying the part of the registry file that is concerned with services; the part defining new token types is introduced in Subsection 10.2.2.1. The syntax of the registry file, which is relevant for services, is as follows:

registry ::= *header entries*
header ::= *ext_title opt_require opt_init*
ext_title ::= 'extension' *extension_name* ';'
extension_name ::= *identifier*
opt_require ::= 'requires' *extension_name_list* ';' | ϵ
extension_name_list ::= *extension_name* | *extension_name* ',' *extension_name_list*
opt_init ::= 'init_function' '=' *identifier* ';'
entries ::= *token_entries hook_entries service_entries*
service_entries ::= *service_entry service_entries* | ϵ
service_entry ::= 'event' *service_name* '(' *parameters* ')' ':' '(' *ecps* ')' *opt_attributes* ';'
| 'action' *service_name* '(' *parameters* ')' *opt_attributes* ';'
service_name ::= *identifier*
parameters ::= *parameter_list* | ϵ
parameter_list ::= *parameter* | *parameter* ',' *parameter_list*
parameter ::= *type_spec parameter_name*
parameter_name ::= *identifier* | ϵ
type_spec ::= *type_name list_spec*
type_name ::= *identifier*
list_spec ::= '*' | ϵ
ecps ::= *ecp_list* | *ecp_list* ',' '...' | '...' | ϵ
ecp_list ::= *ecp* | *ecp* ',' *ecp_list*
ecp ::= *qualifier type_spec ecp_name*
qualifier ::= 'local' | ϵ

ecp_name ::= *identifier*
opt_attributes ::= '{' *attribute_list* '}' | ϵ
attribute_list ::= *attribute attribute_list* | ϵ
attribute ::= *attribute_name* '=' *attribute_value* ';'
attribute_name ::= *identifier*
attribute_value ::= *identifier* | *integer*

Comments are allowed at anywhere in between two syntactic elements. The syntax for comments in as in C++, i.e. there are line comments introduced by '//' and block comments delimited by '/*' and '*/'.

As can be seen from the syntax, a service entry in the registry file consist of two parts: the first (mandatory) part defines the service's input parameters. In case of an event service it also specifies the (non-standard) event context parameters after the colon (':'). The second (optional) part is used to define attributes providing information on the service's implementation. The semantics of the keywords and the supported attributes are specified in tables 10.2 and 10.3. A more detailed description of their usage is given in the next sections.

Keyword	Description
'extension'	The following identifier defines the name of the extension, i.e. the prefix used for all services.
'requires'	The list of identifiers that follows this keyword defines all extensions this extension depends on.
'init_function'	The value assigned to this symbol is the name of an initialization function that is called once after starting up the monitor or this extension. The init functions of different extensions will be called in an order that respects the dependencies specified with the 'requires' keyword. I.e. if extension A requires extension B, the init function of B will be invoked prior to that of A.
'action'	The following prototype defines an information or manipulation service.
'event'	The following prototype defines an event service.
'...'	The ellipsis denotes that the number and types of event context parameters is not known statically.
'local'	This qualifier is only allowed in front of a parameter that is a token or a list of tokens. In this case, it specifies that the token(s) reference an object that is located on the node where the event occurrs.

Table 10.2: Semantics of keywords in the registry file

The OCM-G contains a utility named `makeregistry` which is invoked as follows:

```
makeregistry <name_of_registry_file> <name_of_output_directory>
```

It reads the registry file and translates it into a couple of files that are stored in the named output directory. In particular, the following files will be generated:

- A header file for each defined service, that should be included by the C source file implementing the service. This file will contain the C prototype of the function implementing the service (see Sections 10.1.4.3 and 10.1.4.4) and also includes all other necessary header files.
- An implementation dependent number of other C source and header files which need to be added to the extension library.
- A makefile, that compiles these files and updates the extension libraries to be linked to the monitoring system and the application processes, respectively.

Attribute	Values	Description
'context'	'monitor' 'mainsm' 'sm' 'application'	Defines the context where the action must be executed, or where the event is detected, respectively. Several alternative values can be specified, separated by white space.
'support'	'full' 'partial' 'none'	Specifies whether the service is fully or partially supported, or not supported at all.
'localize'	'node' 'site' 'process' 'none'	Specifies whether the service must be localized to a specific node, site, or process. 'none' means that the service can be executed anywhere, provided that the restrictions given by 'context' are met.
'localize_arg'	*integer* (default: first token or token list parameter)	Specifies the position of the input parameter used for automatic token conversion (see section 10.1.4.2) and for determining the node where the service is to be executed. The specified parameter must either be a token or a token list. The service will be executed on the node, site, or process (depending on the value of 'localize') where the object referenced by the token resides.
'scope'	'extern' 'intern'	If 'intern', the service cannot be invoked via the OMIS tool/monitor-interface.
'auto_convert_to'	*identifier* ['..' *identifier*] (default: no automatic conversion)	When this attribute is set, it is interpreted as the prefix specifying the token class used as the target of automatic token conversion. See section 10.1.4.2.
'allow_unattached'	'false' 'true'	Only relevant when automatic token conversion is in effect. When set to 'true', the service can be invoked for objects not attached by the tool. Otherwise, the monitoring system checks that objects are attached before calling the service.
'instrumentation'	'on_enable' 'on_define'	Specifies when the instrumentation necessary to detect an event should be performed.
'service_function'	*identifier* (default: *service_name*)	Allows to specify a different name for the function implementing the service.
'filter_function'	*identifier* (default: *service_name*)	Allows to specify a different name for the filter function of an event. May be 'NULL' when no such function is needed.
'instrument_function'	*identifier* (default: *service_name*)	Allows to specify a different name for the instrumentation function of an event. May be 'NULL' when no such function is needed.
'delete_function'	*identifier* (default: *service_name*)	Allows to specify a different name for the delete function of an event. May be 'NULL' when no such function is needed.
'distribution_function'	*identifier* (default: 'NULL')	Allows to specify a special request distribution function for this service.
'assembly_function'	*identifier* (default: 'NULL')	Allows to specify a special reply assembly function for this service.

Table 10.3: Attributes in the registry file

10.1.4.2 Automatic Token Conversion

For increased scalability and ease of use, OMIS defines that a token or a list of tokens passed to a service will be converted to the token class the service works on. As this conversion is independent on the actual service, the code actually performing this conversion can be generated by the `makeregistry` tool. However, there may be cases where the implementation of a service needs to see the original token list. Thus, automatic token conversion is optional and can be controled via the registry file. There are three attributes that control automatic token conversion:

- 'auto_convert_to'
  If this attribute is not defined, `makeregistry` will not generate code for automatic token conversion. Thus, the function implementing the service will receive the original token or token list and must care for the proper conversion itself. If the attribute is set, it defines the class of tokens expected by the service. The class is specified by its token prefix, e.g. 'p_' for process tokens. It is also possible to specify a range of token classes, e.g. 't_ .. p_'. In this case, the tokens are converted to the nearest class within the specified range.

- 'localize_arg'
  This attribute specifies the position of the parameter to which automatic token conversion should be applied. At the same time, this parameter will be used to determine the location where the service should be executed.

- 'allow_unattached'
  When automatic token conversion is done and this attribute is set to 'true' (which is the default), the generated code will also check whether the resulting tokens are attached by the tool that requested the service. If a token is not attached, the service will not be invoked for this token. Instead, an error reply will automatically be generated for this token.

10.1.4.3 Implementing Manipulation and Information Services

Manipulation and information services are in most cases implemented by providing a single C function as outlined below. Exceptions occur for services that need specific control over the way how the service is distributed to the local monitors and/or the way how results from local monitors are combined to a global result. See Section 10.1.4.5 for some more details.

No Automatic Token Conversion When automatic token conversion is switched off, the C prototype of the function implementing the service is exactly the same as outlined in Section 10.1.3.1:

```
int ocm_action_<service_function>(Ocm_expansion_context context,
                                  Omis_service_result *result,
                                  <service_parameters>);
```

Here `<service_function>` is to be replaced by the value of the service's 'service_function' attribute in the registry file. The default for this value is the service name. However, a different name may be explicitly specified in the registry file. In this case, the function of Section 10.1.3.1 will be automatically generated by `makeregistry`. This is useful, if an already existing implementation of a service shall be reused and made available with a new service name.

`<service_parameters>` denotes the sequence of the service's input parameters, mapped to C parameters as described in Section 10.1.1. Object tokens passed to the service are passed to the implementing function as-is, i.e. without any conversion or checks. The function is responsible for doing the proper token conversions and for ensuring that the objects refered to are attached by the calling tool, if necessary. The routines defined in Section 10.2 can be used to achieve this.

The `context` parameter specifies the internal execution context of the OCM-G, which among others contains an identifier of the tool that requested execution of this service. The `context` will usually be passed on to other services or to token conversion and access routines.

The return value of this function must be ≥ 0, if processing was successful, and < 0, if an error occurred during its execution for at least one of the objects the service operated on.

If the service has a real result (other than a simple `OMIS_OK` status) or a processing error occurred, the implementing function must assign a dynamically allocated array of `Omis_object_result` structures to `*result`, which is initialized according to the rules outlined in Section 4.3. Otherwise, it must assign a `NULL` pointer. If the function can not allocate the result structure, it should return a negative value and set `*result` to `NULL`, which indicates an out-of-memory condition.

Automatic Token Conversion When automatic token conversion is enabled for a service, the function implementing that service needs to deal with only one object at a time, which simplifies the implementation of services operating on token lists. The C prototype of the function to be supplied in this case is:

```
Omis_status ocm_sngl_action_<service_function>(Ocm_expansion_context context,
                                               char **result,
                                               <service_parameters>)
```

The remarks about `<service_function>`, `context`, and `<service_parameters>` in the previous paragraph also apply in this case, with the following additions:

- The `makeregistry` tool will always create the function defined in Section 10.1.3.1, as this function must do the token conversion and must loop over all objects.
- If the service parameter specified by the 'localize_arg' attribute in the registry file is a list of tokens, only a single token will be passed to the function in `<service_parameters>`.

The function's return value must be the status for the processed object; detailed results or error descriptions must be returned as dynamically allocated strings assigned to `*result`. If the returned status is `OMIS_OK`, `*result` need not be assigned, if there is no real result other than the status. If, however, the return value indicates an error, `*result` must be set to a dynamically allocated error description, except in case of an out-of-memory condition (i.e. `OMIS_NO_MEMORY` is returned). The return value and the string assigned to `*result` are used to build the `Omis_object_result` structure for the object the function is called on.

The following short example shows how to specify a hypothetical process service returning a process's CPU time. It also shows how to invoke an existing service via its C interface.

Assume the following entry for the service in the registry file:

```
extension demo;
action demo_proc_get_cpu_time(token *proc_list)
{
    auto_convert_to = p_;
}
```

Then the implementation of this service could look like:

```
#include <demo_proc_get_cpu_time.h>
#include <proc_get_info.h>
```

```
Omis_status
ocm_sngl_action_demo_proc_get_cpu_time(Ocm_expansion_context context,
                                       char **result,
                                       Ocm_obj_token proc)
{
    Omis_service_result sr;
    Omis_status status;

    /*
    ** prog_get_info will give us the CPU time. There is no need to
    ** check the return value, since the status is also encoded in sr.
    */
    ocm_action_proc_get_info(context, &sr, 1, &proc, (1 << 11));

    if (sr == NULL) {
        *result = strdup("proc_get_info couldn't allocate result");
        return OMIS_NO_MEMORY;
    }

    /*
    ** Just copy the information and free the result of proc_get_info
    */
    status = sr[0].status;
    *result = strdup(sr[0].result);
    ocm_service_result_delete(sr);

    /*
    ** If we couldn't strdup, we are out of memory.
    */
    return (*result != NULL) ? status : OMIS_NO_MEMORY;
}
```

This code assumes that the directory where `makeregistry` placed its file for the 'demo' extension is in the include search path, as well as the directory containing the include files of the OMIS core. Note that this example is for the purpose of illustration only; it would not make much sense to really implement it.

More meaningful and complete examples of action implementation are provided in the OCMG source code in the file `src/Monitor/OCM/SERVICES/EXAMPLEEXT/example_actions.c`.

10.1.4.4 Implementing Event Services

The OCM-G Event Model Before we can go into the details of how to implement new event services, we have to explain the basics of the OCM-G event model. To start with, we will define two terms that must not be confused:

event (occurrence):
An event (or event occurrence) is a state transistion in the observed system that happens at some instance in time. By definition, this means that there can not be two identical event occurrences.

event class:
An event class is a predicate specifying a set of event occurrences. An event belonging to an event class is said to match that class.

So an event service, together with its actual parameters, defines an event class. For instance, the request `thread_has_started_lib_call([p_1],"pvm_send")` defines one event class. Now a basic idea

of OCM-G is that these event classes form a hierarchy, i.e. it is possible to derive smaller, more specific event classes from larger ones by using filters. E.g. we could have a hypothetical event service `pvm_task_sends_to_task(token *task_list, token receiver)` detecting when a task in `task_list` sends to task `receiver`. Then the event class `pvm_task_sends_to_task([p_1], p_2)` can be derived from `thread_has_started_lib_call([p_1],"pvm_send")` by a filter that compares the event context parameter containing the receiver task's token against `p_2`.

Thus, we end up with a tree, whose root is a completely unspecific event class matched by all events. Each event class in this tree may be associated with a set of filters, each of which is used to derive a more specific event class. In addition, some of the event classes may have action lists associated with them. When an event is detected by the OCM-G, it is matched against all the event classes in that tree by doing a tree traversal starting at the tree's root, which matches all events. At each traversed node, the filters associated with that node are evaluated. If a filter evaluates to true, the event also matches the derived event class, which is therfore traversed, too. During this tree traversal, the action lists associated with event classes matched by the event are scheduled for execution.

The `Ocm_ip` Data Type In OCM-G, event classes are represented by an abstract data type `Ocm_ip`. A value of this data type is called an IP. In the following, we specify the methods are available for an IP in OCM-G.

```
Ocm_ip ocm_evmgmt_ip_new(Ocm_instrmnt parent_filter, void *usrdata);
```

This is the constructor for an IP that represents an event class derived from another one by a filter. `parent_filter` is the handle returned by `ocm_evmgmt_ip_filter_add`, when this filter was added to the parent event class. `usrdata` is a generic pointer that is stored in the `Ocm_ip` data structure and may point to user-defined data associated with this IP. It can be retrieved again using `ocm_evmgmt_ip_usrdata`.

Upon error, this function will return NULL.

```
Ocm_ip ocm_evmgmt_ip_root_new(int mem_segment, void *usrdata);
```

This constructor returns an IP that is not derived from another one, but is the root of a new tree. This means that the IP will never be executed during the normal event processing in the OCM-G, but may be explicitly triggered by a call to `ocm_evmgmt_ip_exec`. This call is used only when an extension implements its own means of detecting certain events.

Upon error, this function will return NULL.

```
int ocm_evmgmt_ip_delete(Ocm_ip ip);
```

This is the destructor of `Ocm_ip`. It returns a value ≥ 0 on success, < 0 on failure.

```
Ocm_instrmnt ocm_evmgmt_ip_filter_add(Ocm_ip ip,
                                      const char *filter_name,
                                      void *param);
```

This function adds a filter to an IP that will be called whenever an event matches the event class represented by that IP. `filter_name` is the name of the service to which the filter belongs; `param` is a generic parameter passed to that filter. See Section 10.1.4.4 for a description of filters. The function returns a handle that is used to reference this filter instance, or NULL if an error occurred.

```
int ocm_evmgmt_ip_instr_remove(Ocm_instrmnt instr);
```

This function is used to remove a filter again. This function must not be called after the first child IP has been added to the filter, using `ocm_evmgmt_ip_new`. Once a child IP has been added to the filter, the filter will be removed automatically when it no longer has any child IPs. The function returns a value ≥ 0 on success, < 0 on failure.

```
void * ocm_evmgmt_ip_usrdata(Ocm_ip ip);
```

Returns the value of the `usrdata` argument of the call to `ocm_evmgmt_ip_new` or `ocm_evmgmt_ip_root_new` that created the given IP.

```
Ocm_bool ocm_evmgmt_ip_is_enabled(Ocm_ip ip);
```

Returns True, iff the given IP is enabled. Enabling and disabling IPs is done by the `csr_enable` and `csr_disable` services. This call is needed in the case where an event service does not make use of automatic token conversion. In this case, the service's implementation must decide whether or not to (un)instrument a process when the results of token conversion changed. This usually depends on whether or not the IP is enabled.

```
typedef Ocm_any_val Ocm_ecp_val;
int ocm_evmgmt_ip_exec(int num_ips, const Ocm_ip ips[],
                       Ocm_evmgmt_context context,
                       int num_ecps, const Ocm_ecp_val ecps[]);
```

This function executes a set of IPs given by the array `ips` of length `num_ips`, i.e. executes all filters associated with these IPs and schedules all associated actions lists for execution.

The event context parameters passed to the filters and actions are given by the array `ecps` of length `num_ecps`. They must be stored in the following way: `ecps[1]` ... `ecps[5]` contain the values of the general event context parameters, i.e. **node**, **proc**, **thread**, **time**, and **csr**. `ecps[0]` is reserved for internal use and need not be initialized. The elements `ecps[6]` ... contain the service specific event context parameters of the event service that created the IPs to be triggered. The order in which they have to be stored corresponds to the order of their definition in the registry file.

The `context` argument provides a means for OCM-G to pass context information during traversal of the IP tree. When `ocm_evmgmt_ip_exec` is called from within a filter to execute some of the filter's child IPs, the value of the filter's `context` parameter must be passed. If the function is used to trigger a root IP, `NULL` should be passed. The function returns a value ≥ 0 on success, < 0 on failure.

Getting an IP for an Event Class For each event service, the `makeregistry` tool will create a function

```
Ocm_ip ocm_event_<service_name>(Ocm_expansion_context context,
                                Omis_event_hints hints,
                                Omis_service_result *result,
                                <service_parameters>);
```

Here `<service_name>` refers to the name by which the service is known at the OMIS tool/monitor-interface, `<service_parameters>` must be replaced by the proper transcription of the service's input parameters (see Section 10.1.1).

This function can be called inside an event define function (see below) to get the IP of a specific event class, from which a new event class can be derived via a filter.

The `context` parameter specifies an internal execution context of the OCM-G. The value passed to this argument should be the same that is received by the event define function. The `hints` parameter should be handled in the same way (although it is currently not used in OCM-G).

The Functions Implementing an Event Service In the most general case, four functions must be provided to implement an event service:

1. The *event define function*
   This function constructs an IP representing the event class defined by the event service and its actual parameters. It is also responsible for generating the reply sent back to the tool immediately after it requested this service.

2. The *instrumentation function*
   Some events can only be detected after certain manipulations have been done to the target system. These manipulations are usually called *instrumentation*. An example is to insert trap instructions into a target process's code in order to detect that execution of this process reaches a given point in the code. The instrumentation function is responsible for inserting and removing this kind of instrumentation. When it is called depends on the setting of the event's 'instrumentation' attribute in the registry file. If 'instrumentation' is set to 'on_define', inserting the instrumentation is requested right after the event define function finished, while removing the instrumentation is requested only when the IP is deleted. If 'instrumentation' is set to 'on_enable', instrumentation is requested to be inserted whenever the IP is enabled, and removed again whenever the IP is disabled or deleted. Thus, the setting of 'instrumentation' is determined by a trade-off between the cost of inserting/removing the instrumentation and the cost of unnecessarily executing it.

3. The *filter function*
   As explained above, filters are used to derive more concrete event classes from more general ones. The filter function is called whenever an event matching the more general event class is detected. The filter then has to determine whether the event matches one or more of the more concrete event classes. This decision is usually based on the values of event context parameters, but can also be based on other information on the target system that is acquired by the filter itself. The filter then calls `ocm_evmgmt_ip_exec` for those of its child IPs that representing event classes matched by the processed event.

4. The *event delete function*
   This is a callback that is invoked just before an IP is deleted from the IP tree. Its task is to perform additional cleanup, such as freeing data pointed to by the IP's `usrdata` pointer.

These functions must not be directly called from extensions, they are invoked only by the monitoring system's event management infrastructure.

The Event Define Function

```
Ocm_ip ocm_event_define_<service_function>(Ocm_expansion_context context,
                                           Omis_event_hints hints,
                                           Omis_service_result *result,
                                           <service_parameters>);
```

Here <service_function> is to be replaced by the value of the service's 'service_function' attribute in the registry file. The default for this value is the service name. However, a different name may be explicitly specified in the registry file.

<service_parameters> denotes the sequence of the service's input parameters, mapped to C parameters as described in Section 10.1.1. When automatic token conversion is disabled in the registry file, the parameters passed to the function will be an exact copy of the service's parameters. Otherwise, the token list (or single token) argument defining the objects the service should work on is converted to the requested token class before it is passed to this function.

The `context` parameter specifies the internal execution context of the OCM-G, which among others contains an identifier of the tool that requested execution of this service. The `context` will usually be passed on to other services or to token conversion and access routines.

The parameter `hints` is reserved for future use. It is intended to be used to optimize event detection and/or filtering. At the moment, it is not set to any meaningful value.

The function must return a valid IP, if processing was successful for at least one object, or `NULL`, processing failed completely. There are several alternatives to get the IP to be returned:

- The function may call another event service function to get a parent IP representing an event class that includes the current class. It then may add a filter to this IP and generate a new IP with this filter as its parent filter. In this case, the filter only determines whether or not that IP should be executed.
- It is also possible to get the parent IP and install the filter only once (when the event define function is called for the first time), but still generate a new IP with every call. In this case, it must maintain some data structure that allows the filter to determine *which* of these IPs must be executed. Although more complex to implement, this strategy can increase efficiency, since the number of filter executions is reduced.
- An already existing IP can be reused, provided that it represents exactly the same event class.
- The function may return a new root IP. In this case, the service's implementation itself must ensure that the IP is executed whenever an event matching the requested event class occurs.

Of course, also mixtures of these alternatives can be implemented. Note that this function should not perform any instrumentation of the target system, as this is done in the instrumentation function.

When a processing error occurred, the function must assign a dynamically allocated array of `Omis_object_result` structures to `*result` containing the error message(s) according to the rules outlined in Section 4.3. If the function can not allocate the result structure, it should return `NULL` and set `*result` to `NULL`, which indicates an out-of-memory condition.

The Instrumentation Function (No Automatic Token Conversion)

```
int ocm_instrument_<instrument_function>(Ocm_expansion_context context,
                                         Omis_event_hints hints,
                                         Ocm_ip ip,
                                         Ocm_bool insert,
                                         Omis_service_result *result,
                                         <service_parameters>);
```

<instrument_function> is the value of the *instrument_function* attribute in the registry file. `context`, `hints`, `result`, and <service_parameters> have the same meaning as with the event define function. `ip` specifies the IP for which instrumentation has to be inserted (if `insert` == True) or removed (if `insert` == False).

The function will always be called with an exact copy of the event service's parameter list. It is the responsibility of the programmer to ensure that the token (or token list) specifying the objects to be monitored by the event service is properly converted to figure out which objects actually have to be (un)instrumented. Since the semantics of OMIS dictates that token lists are evaluated dynamically, the programmer must also account for the fact that the result of this conversion may change with time (e.g. due to objects being attached or detached by the tool). Thus, callbacks handling these changes should be installed using the dynamic token conversion function described in Section 10.2.1.3.

The function must return a value ≥ 0 if completely successful, otherwise, it must return a value < 0 and assign a dynamically allocated array of `Omis_object_result` structures to `*result` containing the error message(s) according to the rules outlined in Section 4.3. If the function can not allocate the result structure, it should set `*result` to `NULL`, which indicates an out-of-memory condition.

The Instrumentation Function (Automatic Token Conversion)

```
Omis_status ocm_instrument_<instrument_function>(Ocm_expansion_context context,
                                                 Omis_event_hints hints,
                                                 Ocm_ip ip,
                                                 Ocm_bool insert,
                                                 char **result,
                                                 <service_parameters>);
```

`<instrument_function>` is the value of the *instrument_function* attribute in the registry file. `context` and `hints` have the same meaning as with the event define function. `ip` specifies the IP for which instrumentation has to be inserted (if `insert` == True) or removed (if `insert` == False).

`<service_parameters>` denotes the sequence of the service's input parameters, mapped to C parameters as described in Section 10.1.1, with the parameter specified by the 'localize_arg' attribute in the registry file handled in a special way. At its parameter position, always a single token will be passed (even if the service itself accepts a token list). The type of this token will always reflect the type defined in the 'auto_convert_to' attribute in the registry file. The monitoring system's event management will automatically take care of objects that must be (un)instrumented due to changes in the conversion result of the object list passed to the service, or due to enabling or disabling of the IP. The function is called once for each single object that needs to (un)instrumented.

The function's return value must be the status for the processed object. If this status indicates an error, `*result` must be set to a dynamically allocated error description. Setting `*result` to to `NULL` indicates an out-of-memory condition.

The Filter Function

```
typedef Ocm_any_val Ocm_ecp_val;
int ocm_filter_<filter_function>(Ocm_evmgmt_context context,
                                 int num_ips,
                                 const Ocm_ip ips[],
                                 int num_ecps,
                                 const Ocm_ecp_val ecps[],
                                 void *param);
```

After a filter has been added to an IP using `ocm_evmgmt_ip_filter_add`, the filter function is called whenever an event matching this IP's event class is detected and at least one of the filter's child IPs is enabled. The task of the filter function is to analyze the event context parameters it receives in the `ecps` array of length `num_ecps` and maybe also other state of the target system. Based on this information,

the filter function must decide which of its child IPs, which are passed to it in the `ips` array (of length `num_ips` should be processed. To do so, the filter might use information stored in these IPs' `usrdata` components. The filter may also copy and extend/modify the event context parameters to be passed to its child IPs. Once the IPs to be executed are determined, the filter must call `ocm_evmgmt_ip_exec` to execute them.

The event context parameters are stored in the following way: `ecps[1]` ... `ecps[5]` contain the values of the general event context parameters, i.e. **node**, **proc**, **thread**, **time**, and **csr**. `ecps[0]` is reserved for internal use. The elements `ecps[6]` ... contain the service specific event context parameters of the event service used to create the filter's parent IP. The order in which they are stored corresponds to the order of their definition in the registry file.

The `param` argument is set identical to the value passed to `ocm_evmgmt_ip_filter_add` when adding the filter. This parameter is useful, if the same filter function is added to several IPs.

`context` is a pointer to an implementation dependent data structure that is used by the monitoring system's event management to pass around context information when traversing the IP tree. This parameter should be passed to `ocm_evmgmt_ip_exec` without modification.

The return value of the filter must be ≥ 0, unless a processing error has been occurred.

The Event Delete Function

```
void ocm_event_delete_proc_control(Ocm_ip ip,
                                   Ocm_bool deleting_filter);
```

This function is called just before an IP generated by the event define function is deleted by the monitoring system, since it is no longer needed. If `deleting_filter` is True, this indicates that also the IP's parent filter has been removed, since it no longer has any child IPs and thus is not needed any longer.

An event delete function is usually necessary for two purposes: freeing any data structures referenced by the IP's `usrdata` component and updating any static data structures used by the event service's implementation. The function must not free the IP or the filter itself.

An example A complete example of an event service implementation is provided in the OCM-G source code in the file `src/Monitor/OCM/SERVICES/EXAMPLEEXT/example_event.c`.

10.1.4.5 Implementing Custom Distribution and Reply Assembly

As explained in the previous paragraphs, the OCM-G automatically distributes services to the proper local monitors, based on the attributes specified in the registry file. In some cases, it may be necessary to define a non-standard way of distributing a request for a specific service. Likewise, it may be necessary to combine the results from different local monitors in a special way. Thus, OCM-G provides a means to define custom request distribution and reply assembly functions for a service.

Custom distribution functions are hardly ever needed. However, custom reply assembly functions are more common, especially in combination with global objects. Here, the service reading the value of the global object must provide a reply assembly function, which combines the results of the local nodes into a global result.

Distribution Function A distribution function has the following prototype:

```
Omis_status
<name_of_distribution_function>(Ocm_expansion_context context,
                                const Ocm_service_desc *service,
                                Ocm_service_distribution *distrib,
                                char **result)
```

Here **service** ist a description of the service request, including all its parameters, while **distrib** is a result parameter which specifies, how the service should be distributed to different nodes.

The **result** argument and the function's return value are used to return an error description in case of processing errors.

In order to use a custom distribution function, the following attribute must be added to the service's entry in the registry file:

```
distribution_function = <name_of_distribution_function>;
```

Reply Assembly Function A reply assembly function has the following prototype:

```
Omis_status
<name_of_assembly_function>(int num_replies,
                            const Omis_service_result *replies,
                            Omis_service_result *result,
                            char **errmsg)
```

The parameter **replies** is an array containing all the service results from local monitors (**num_replies** is the length of the array). The function must analyze these replies and combine them into a single service result for the whole service request and store it into **result**. This service result must be dynamically allocated.

The **result** argument and the function's return value are used to return an error description in case of processing errors.

In order to use a custom reply assembly function, the following attribute must be added to the service's entry in the registry file:

```
assembly_function = <name_of_assembly_function>;
```

10.2 The C-Interface for Tokens

OMIS uses opaque identifiers, called tokens, to identify objects in the target system or the monitoring system. In OCM-G, there is a data type **Ocm_obj_token**, which takes over this role. I.e., each object is accessed via this data type. The internal structure of this data type should neither be known nor be exploited by programmers writing extensions to the OCM-G. The only way to access tokens should be the functions documented in Section 10.2.1. Section 10.2.2 explains how an extension can add tokens for new object classes.

10.2.1 General Token Interface

10.2.1.1 Constructors and Destructors

The following functions create and destroy tokens. Note that these functions are not available in the **application** context.

```
Ocm_obj_token ocm_obj_new(int mem_segment, const char *string,
                          unsigned int size);
```

Creates a new (proxy) object[1] of given **size** in the memory segment specified by **mem_segment**. Possible values for **mem_segment** are **ocm_mem_global** for the shared memory segment, which can be accessed by both the local monitor and the monitor modules in the local application processes, and **OCM_MEM_PRIVATE** for the private memory of the local monitor.

The parameter **string** specifies the string representation, which the resulting token should have. Note that this representation must obey the rules for token strings defined in the OMIS document. In addition, the string must be globally unique, i.e., there must not be two tokens with the same token string inside one instance of the OCM-G.

The function returns a token for the newly created (proxy) object. The data area (of size **size**) allocated for the object is initialized with zeroes. The function **ocm_obj_token_deref()** can be used to get a pointer to this data area.

On error, the function returns NULL.

```
int ocm_obj_delete_when_unreferenced(Ocm_obj_token token);
```

Deletes the object referenced by the given **token**. A deleted object is no longer accounted for in the **ocm_obj_token_find_...()** functions. The actual deletion of the object's storage is delayed until there are no more tokens referencing the object in order to avoid dangling references. Note that the token itself is not deleted by this function. Use **ocm_obj_token_delete()** for this purpose.

The return value is < 0, if and only if an error occurred.

```
Ocm_obj_token ocm_obj_token_new(int mem_segment, const char *string);
```

Returns the token of the object referenced by the token string (as used in the OMIS interface) given in **string**. If a local (proxy) object matching the given **string** exists, a token referencing this object will be returned. It can be used to get a pointer to the object's data via the function **ocm_obj_token_deref()**. If no local (proxy) object exists, a new object token without any referenced object will be created. This token will not (and will never) reference any object on the local node. However, it can be inserted into parameter lists sent to other nodes. The token will be allocated in the memory given by **mem_segment**. Note that a request to allocating the token in shared memory when the referenced object is allocated in local memory is an error, since this would prevent the token from being dereferenced in the application context.

On error, the function returns NULL.

```
Ocm_obj_token ocm_obj_token_dup(int mem_segment, const Ocm_obj_token token);
```

Duplicates the given **token**. Note that only the token is duplicated, not the referenced (proxy) object.

On error, the function returns NULL.

```
int ocm_obj_token_delete(Ocm_obj_token token);
```

Deletes the specified **token**. Note that only the token is deleted, not the referenced (proxy) object. The object itself is only deleted if **ocm_obj_delete_when_unreferenced()** has been previously called and this is the last token referencing the object.

The return value is < 0, if and only if an error occurred.

[1] We write (proxy) object, since the function either can create a new object inside the monitoring system (e.g. a counter), or a proxy object representing an object of the traget system (e.g. a node or application process).

```
Ocm_obj_token *ocm_obj_token_array_dup(unsigned int len,
                                       const Ocm_obj_token token_ary[]);
```

Duplicates an array of tokens. The function returns a dynamically allocated array of size **len**. Each element of the array is a copy of the corresponding element of **token_ary**, created via **ocm_obj_token_dup()**.
On error, the function returns NULL.

```
int ocm_obj_token_array_delete(unsigned int len, Ocm_obj_token token_ary[]);
```

Deletes each token in the array **token_ary** of length **len** by calling **ocm_obj_token_delete()**.
The return value is < 0, if and only if an error occurred.

10.2.1.2 Access Functions

The following functions are available in any execution context.

```
const char *ocm_obj_token_string(Ocm_obj_token token);
```

Returns the string representation of the given **token**, i.e. the representation used in OMIS requests and replies. The string returned must not be altered or freed. This function does not dereference the token, i.e. it does not require that the token actually references a (proxy) object. The function never returns NULL.

```
Ocm_bool ocm_obj_token_is_equal(Ocm_obj_token t1, Ocm_obj_token t2);
```

Compares the two tokens **t1** and **t2** without referencing them. Returns **true**, if and only if the two tokens are equal, i.e. have the same string representation.

```
Ocm_bool ocm_obj_token_has_type(const Ocm_obj_token token, const char *type);
```

Returns **true**, if and only if the **token** belongs to the object class specified by **type**. The **type** is specified by the token prefix defined in the OMIS document (e.g., **p_** for processes).

```
Ocm_bool ocm_obj_token_is_undefined(const Ocm_obj_token token);
```

Returns **true**, if and only if the **token** is an undefined token or NULL. This function is actually implemented as a macro.

```
void *ocm_obj_token_deref(Ocm_obj_token token);
```

Dereferences the **token**, i.e., returns a pointer to the data area of the (proxy) object referenced by **token**.
On error, the function returns NULL.

10.2.1.3 Token Conversion Functions

The following two functions convert tokens from one type into the other. Please refer to the OMIS document for more information on token conversion (localization and expansion).

Note that these functions are not available in the **application** context.

```
Ocm_obj_token *ocm_obj_token_convert(unsigned int len,
                                     const Ocm_obj_token token_ary[],
                                     Ocm_expansion_context context,
                                     Ocm_bool check_attached,
                                     const char *to_type_low,
                                     const char *to_type_high,
                                     int *num,
                                     Omis_service_result *result);
```

Converts the array of tokens **token_ary** of length **len** into an array of tokens, which all have a type between **to_type_low** and **to_type_high**. The types are specified by the token prefix defined in the OMIS document (e.g., **p_** for processes). The value passed to **context** should be the same which the service calling **ocm_obj_token_convert** received as its **context** parameter. The function will return a dynamically allocated array of tokens; the length of this array is returned in ***num**. The returned array should be deleted with **ocm_obj_token_array_delete()**.

If **check_attached** is true, the function checks for each token whether it is attached, before trying to convert it. If a token is not attached, an error reply is generated and stored in **result**. If **check_attached** is false, the tokens are converted independently of whether they are attached or not.

On severe errors, the function returns NULL. If an error occurs only during the conversion of some tokens, an error reply is returned in **result**. This reply should be appended to the service's reply.

```
Ocm_obj_token *
ocm_obj_token_dynamic_convert(unsigned int len,
                              const Ocm_obj_token token_ary[],
                              int tool_id,
                              Ocm_bool check_attached,
                              const char *to_type_low,
                              const char *to_type_high,
                              Ocm_obj_token_watch_callback callback,
                              void *param,
                              int *num,
                              Omis_service_result *result,
                              Ocm_obj_token_watch_id *watch_id);
```

This function works like **ocm_obj_token_convert()**, with the following addition: Once the function is called, it remembers the requested conversion and calls the specified **callback** whenever the conversion changes. The prototype of the callback function is

```
typedef int (*Ocm_obj_token_watch_callback)(const Ocm_obj_token token,
                                            Ocm_bool added,
                                            void *param);
```

When the callback is called, the arguments **token** and **added** indicate that a specific token should be added to the conversion result (**added** = true) or removed from the result (**added** = false). The **param** argument is initialized with the value passed to the **param** argument of **ocm_obj_token_dynamic_convert()**.

The **watch_id** returned by **ocm_obj_token_dynamic_convert()** can be used to remove the callback again (see below).

Note that instead of a **context** argument, this function just receives a tool ID. The value passed to **tool_id** should be **context.tool_id**.

```
int ocm_obj_token_watch_callback_remove(Ocm_obj_token_watch_id id);
```

This function allows to remove the conversion callback installed via **ocm_obj_token_dynamic_convert()**. The value passed to **id** must be the value returned by **ocm_obj_token_dynamic_convert()** via the **watch_id** parameter. The return value is < 0, if and only if an error occurred.

10.2.1.4 Attaching and Detaching

As explained int the OMIS document, a tool can attach to and detach from objects in order to define its scope of interest. The following two functions provide this functionality. Note that these functions are not available in the **application** context.

```
int ocm_obj_token_attach(Ocm_obj_token token, int tool_id);
```

Attaches the tool identified by **tool_id** to the object identified by **token**. The **tool_id** can be extracted from a **Ocm_expansion_context context** variable via the field name **tool_id**, i.e. **context.tool_id**. The return value is < 0, if and only if an error occurred.

```
int ocm_obj_token_attach(Ocm_obj_token token, int tool_id);
```

Detaches the tool identified by **tool_id** from the object identified by **token**. The **tool_id** can be extracted from a **Ocm_expansion_context context** variable via the field name **tool_id**, i.e. **context.tool_id**. The return value is < 0, if and only if an error occurred.

10.2.1.5 Searching and Finding

The following functions can be used to search for (proxy) objects with certain properties. Note that these functions are not available in the **application** context.

```
Ocm_obj_token ocm_obj_token_find_one(const char *token_type,
                                     Ocm_bool (*predicate)(Ocm_obj_token tok,
                                                           const void *param),
                                     const void *param);
```

Searches all objects on the local node, which have a type specified by **token_type** for one that fulfills the given predicate and return its token. The type is specified by the token prefix defined in the OMIS document (e.g., **p_** for processes). The **predicate** is specified as a function which receives a token and the value of the **param** argument passed to **ocm_obj_token_find_one()**. A NULL **predicate** always evaluates to True. On error, or if no matching token is found, the function returns NULL.

```
Ocm_obj_token *ocm_obj_token_find_all(const char *token_type,
                                      Ocm_bool (*predicate)(Ocm_obj_token tok,
                                                            const void *param),
                                      const void *param,
                                      int *num)
```

Like **ocm_obj_token_find_one()**, with the difference that all matching tokens are returned in a dynamically allocated array. The size of this array is returned via **num**. On error, the function returns NULL. If no matching tokens are found, the function returns non-NULL and **num** is set to 0. The returned array should be deleted with **ocm_obj_token_array_delete()**.

10.2.2 Adding New Tokens

An OCM-G extension can add new token classes for monitor objects to the OCM-G. The following sections explain how this is achieved.

10.2.2.1 The Registry File

Like new services, new token classes must be declared in the registry file. The syntax for these entries is as follows:

token_entry ::= 'token' *token_name opt_attributes* ';'
token_name ::= *identifier*

See Section 10.1.4.1 for the syntax of *opt_attributes*. Table 10.4 documents the available attributes and thein meaning.

10.2.2.2 Implementing Tokens

In order to implement a new token type, up to four functions need to be implemented.

The Localization Function This function must be provided for all local tokens which should be subject to the OMIS token conversion. The function has the following prototype:

```
Ocm_obj_token ocm_obj_token_<token-type>localize(Ocm_obj_token token,
                                                 const char *to_type);
```

Here, `<token-type>` denotes the token type, specified by the token prefix defined in the OMIS document (e.g., **p_** for processes). If a localization function is provided, this prefix must also be assigned to the 'localize_function' attribute in the registry file.

The parameter **token** receives the token of the object which should be localized. In the description below, we denote this object with O. The parameter **to_type** specifies the type of the token which should be returned. The token type is specified by the token prefix defined in the OMIS document. The OCM-Gwill call this function only with the following values for **to_type**:

- 's_': In this case, the function must return the token of the site where O resides.
- 'n_': In this case, the function must return the token of the node where O resides.
- Prefix corresponding to the immediate parent of the token in the OMIS object hierarchy: In this case, the function must return the token of the immediate parent object of O.

Attribute	Values	Description
'class'	'local' 'global' 'array of' *identifier*	Objects can either be local, i.e., available only on a single node, or global, i.e., available on all nodes. In addition, arrays of objects can be defined. In this case, *identifier* is the token type of the array element, specified by its prefix. Arrays are always global.
'parent'	*identifier* (default: 'u_')	Specifies the token type of the object where objects of this type are contained in, i.e., the type of the immediated parent in the token hierarchy (see OMIS document). This attribut is only meaningful for objects with 'class = local'.
'attachable'	'false' 'true'	Specifies whether the objects should be attachable. If this attribute is set to 'false', the tokens can not and need not be attached by a tool.
'auto_attach'	'false' 'true'	Specifies whether the objects should be attached automatically when they are created. This attribute is ignored when 'attachable' is 'false'.
'localize_function'	*identifier* (default: 'NULL')	Specifies the localization function for these objects. See Sect. 10.2.2.2.
'attach_function'	*identifier* (default: 'NULL')	Specifies the name of a function which is called whenever an object is attached or detached. See Sect. 10.2.2.2.
'delete_function'	*identifier* (default: 'NULL')	Specifies the name of a function which is called whenever an object is deleted. See Sect. 10.2.2.2.
'constructor'	*identifier* (default: 'NULL')	Specifies the name of a function which constructs the local parts of a global object and/or the elements of an array. See Sect. 10.2.2.2.

Table 10.4: Token attributes in the registry file

The Attach Function This function is optional: the mechanism of attaching and detaching objects is provided by the OCM-G core. The only purpose of this function is to provide a notification when an object was attached or detached, in case the object implementation wants to perform some additional actions.

The function has the following prototype:

```
int ocm_obj_token_<token-type>attach(Ocm_obj_token token,
                                     int tool_id,
                                     Ocm_bool attach);
```

Here, `<token-type>` denotes the token type, specified by the token prefix defined in the OMIS document (e.g., **p_** for processes). If an attach function is provided, this prefix must also be assigned to the 'attach_function' attribute in the registry file.

The argument **token** receives the token of the object being attached (**attach = true**) or detached (**attach = false**). The argument **tool_id** indicates the ID of the tool that attached or detached the object.

The function should return a value ≥ 0 if and only if an error occurred.

The Delete Function This function is optional: the mechanism of deleting objects is provided by the OCM-G core. The only purpose of this function is to provide a notification before the object's data is deleted. Typically, the function will perform some additional cleanup and/or will free some memory, to which pointers have been stored in the object's data area (see **ocm_obj_new()**). Note that this function is called only when an *object* is about to be deleted, not when a **token** is deleted.

The function has the following prototype:

```
int ocm_obj_token_<token-type>delete(Ocm_obj_token token);
```

Here, `<token-type>` denotes the token type, specified by the token prefix defined in the OMIS document (e.g., **p_** for processes). If an attach function is provided, this prefix must also be assigned to the 'delete_function' attribute in the registry file.

The argument **token** receives the token of the object, which is about to be deleted.

The function should return a value ≥ 0 if and only if an error occurred.

The Constructor A constructor function must be implemented for tokens representing global objects or arrays of objects. For tokens of these types, the implementation consists of two parts:

- One part deals with the global aspects. This part is automatically generated from the information of the registry file. In particular, a function

  ```
  Ocm_obj_token ocm_obj_<token-type>new(Omis_status *status, char **result);
  ```

  is generated. This function can and should be used to create a new global object or a new array. Here, `<token-type>` denotes the token type (token prefix) specified in the registry file.

- The other part deals with local aspects, i.e., the local parts of a global object, or the elements of an array. This part of the implementation must be provided by the programmer, i.e., the token implementation must explicitly provide a constructor as outlined below.

The constructor must create:

- the local part of a global object, in case of tokens which have **class = global** specified in the registry file. The type of the local object is the same as
- a (local) element of an array, in case of tokens which have **class = array of *type*** specified in the registry file. Here ***type*** is the object of the token which must be created by the constructor.

The constructor function has the following protoype:

```
Ocm_obj_token ocm_obj_local_<token-type>new(const char *tok_str,
                                            Omis_status *status,
                                            char **result);
```

Here, `<token-type>` denotes the token type (token prefix) specified in the registry file. This prefix must also be assigned to the 'constructor' attribute in the registry file.

The constructor must create an object of the appropriate type, i.e., the `<token-type>` in case of a global object, or the element type in case of an array object. The object should be created by calling the **ocm_obj_new(ocm_mem_global, tok_str, size)**, where **size** is the size of the data area which should be allocated for the object.

If no error occurs, the function must return the token returned by **ocm_obj_new()**. In addition, ***status** should be set to **OMIS_OK**. The **result** parameter should not be set in this case.

In case of an error, the function must return NULL, ***status** should be set to the appropriate error code, and ***result** should be assigned with a dynamically allocated string providing an error description.

10.3 Complete Example Code for an Extension

A complete example of an OCM-G extension can be found in the OCM-G source code in the directory

`src/Monitor/OCM/SERVICES/EXAMPLEEXT`

Bibliography

[1] OMIS – On-line Monitoring Interface Specification. Version 2.0. Lehrstuhl für Rechnertechnik und Rechnerorganisation Institut für Informatik (LRR-TUM), Technische Universität München. `http://wwwbode.informatik.tu-muenchen.de/~omis`

11 GNU GENERAL PUBLIC LICENSE Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions

of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the program's name and a brief idea of what it does.> Copyright (C) 19yy <name of author>
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) 19yy name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> <signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.