Monitoring Technologies for Parallel On-Line Tools*

Arndt Bode, Wolfgang Karl, Thomas Ludwig, Roland Wismüller
Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Institut für Informatik, Technische Universität München
D-80290 München, Germany
{bode|karlw|ludwig|wismuell}@in.tum.de

Abstract *For more than 10 years the research group at LRR-TUM investigates concepts for on-line monitoring techniques and designs and implements monitoring systems for various hardware and software architectures. From the early systems in the beginning of the 90's to the sophisticated interface-based approach for interoperable tools many milestones were reached that represented each a considerable progress in the field of tool construction. The paper gives an overview about the most important aspects of our work and pinpoints the increase of knowledge in this area.*

Keywords: Run-time tools, interoperability, on-line monitoring, distributed systems

1 Introduction

Parallel processing is a key technology of the 21th century both for commercial/industrial applications and for science. Current needs of computational power can only be satisfied by using parallel and distributed architectures like multiprocessor and multicomputer systems. However, the development and maintenance of software for parallel computers requires significantly higher efforts than for sequential ones. A major part of this extra time and effort is spent on testing, debugging, and optimizing the code [23].

Performing these tasks efficiently requires elaborate tool support. Among the tools used, *on-line tools* provide the most benefits. On-line tools – in con-

*Partly funded by the German Science Foundation, Contract SFB 342, TP A1

trast to off-line tools – are applied to an application *while it is executing*, enabling the tools to adjust to the execution. In addition, such tools are capable not only of observing target application behavior, but also of manipulating the running application and guiding its execution trajectory. Numerous different classes of on-line tools exist. Among the most common are debuggers, performance analyzers, visualizers, load balancers, and steering tools.

Any on-line tool requires a component that handles low level accesses to the target system consisting of the hardware, operating system, programming environment, libraries, and finally the application. This component is termed an *on-line monitoring system* (or shortened, a *monitor*), although it is not usually designed as a separate system, but tightly integrated into the tool. A monitoring system has to support the manipulation of the target system as well as its observation. This requirement plus the demand to minimize the probe effect by keeping unwanted intrusion minimal makes on-line monitors difficult and expensive to implement. In addition, monitors cannot easily be constructed in a portable way, since they have to tightly interact with the target system. The monitoring system therefore turns out to be the key part of any on-line tool. It becomes even more important when we think towards tool environments.

Due to the paramount importance of monitoring techniques for the development of on-line tools, and thus, in turn, for the use of parallel computer architectures, the research group at LRR-TUM investigates this area for more than 10 years. This paper gives a review of the most important results and pinpoints the increase of knowledge in the area of parallel and distributed on-line monitoring. Section 2 provides an overview on early monitoring concepts and their deficiencies as seen from today's perspective. Our current approach to on-line monitoring is outlined in Section 3. In the next two sections, we show that this approach is powerful enough to support interoperable tool environments, and flexible enough to allow an easy adaption to largely different target architectures. A comparison to related work is provided in Section 6. Finally, we conclude with a summary of the lessons learned and an outlook to future research topics.

2 Early Monitoring Concepts

In the early 1990's, virtually no run-time tools existed that made use of a clearly separated monitoring system. Instead, they followed a monolithic approach, where a proprietary monitoring component is an integral part of the tool. Independent monitoring systems first have been used with embedded systems, as they often required a hardware monitor due to real time constraints and limited target resources.

2.1 The MMK Monitoring System

When the first parallel computers appeared, their distributed hardware architecture also required the use of distributed monitoring systems, which are separated from the tools. A goal of the TOPSYS project [2, 3] was to design and develop a monitoring system – both in hardware and in software – for the iPSC/2 and iPSC/860 computers running the MMK multi-threading kernel. This monitoring system [4] provided a well defined interface offering services specific to the tools contained in the TOPSYS environment. The existence of this interface allowed to provide different implementations of the monitoring system, like a software and a hardware realization, without a need to adapt the tools.

In order to support an implementation in hardware, a minimal and simple-to-implement interface has been designed. It offered services to monitor and control objects of the parallel program, like *return information on a mailbox*, *single-step a task*, *trace predefined event information of a task*, *measure the integrated wait queue length of a semaphore*, etc. In order to keep the interface simple, a service request could only relate to a single object. The only exception was a service to start all tasks. In addition, there have been services returning the list of all currently existing tasks, mailboxes and semaphores. However, the monitoring interface was completely local, i.e. at any given time, a tool had to directly communicate with each of the monitors on the individual nodes. In the above mentioned services, *all tasks* therefore really means *all tasks on the local node*. Thus, the monitoring system essentially was a simple collection of independent local monitors.

The tools communicated with these monitors using explicit message passing. After sending a request message to the proper monitor, the tool waited for the monitor's result. In later versions of TOPSYS, an asynchronous receipt of results was implemented, as the former synchronous scheme resulted in unacceptable performance of the tools due to high blocking times.

The local monitors have been designed using the event/action-paradigm. This means that the monitor internally distinguishes between two main activities: Most of the time, the monitor just observes the target program and waits for relevant events to occur. Once an event has been detected, the necessary actions are determined and executed. However, at the level of the monitoring interface, actions could not be freely associated with events. Only a small number of events allowed a restricted choice of actions. On the other hand, these events could be combined to complex predicates, using **and**, **or**, and $\rightarrow$ operators. By using so-called *cross-triggers*, i.e. signals sent from one monitor to an other, even global predicates could be defined. However, the tool was responsible to program these cross-triggers properly.

The software monitor was implemented as two additional threads in the address space of the parallel program. Thus, state information could be acquired by just reading the internal data structures of the MMK run-time system. Information on MMK-related events, like receipt of a message, was acquired via hooks in the MMK, i.e. places in the MMK, where monitoring code was inserted. While at first, this code was fixed at compile-time of the MMK, later versions used a call-back mechanism allowing to define the code to be executed at run-time. Although this monitor implementation eased the data acquisition and reduced the monitoring overhead, it had the disadvantage that the monitor was not protected from being write accessed by the target program and thus could be crashed, e.g. while debugging a faulty application.

The design of the MMK monitoring system primarily was a result of the distributed architecture of the target platform, combined with the wish to implement the monitors in both software and hardware. Nevertheless, the design also revealed additional advantages, like increased platform independence and portability of the tools. But it soon became clear that the design still had a couple of deficiencies. First, it was rather inflexible, since only a limited choice of actions was available for each observable event. Therefore, the monitor's code sometimes had to be changed in order to bind a specific action to an event, even if both the action and the event already have been supported by the monitor. Scalability proved to be insufficient, too, since the interface only allowed to request a service for a single object at a time, resulting in a high communication overhead. Finally, although the software implementation proved to be efficient, it showed a weak adaptability and portability, since there was no clearly defined interface to the target system. In addition, the missing protection often rendered the system useless for debugging purposes.

2.2 The PARIX Monitoring System

The deficiencies of the MMK monitoring system have been addressed in the successor version for Parsytec parallel computers using the PARIX run-time system [34]. Since an implementation in hardware was no longer a goal, a more flexible and powerful interface could be designed. It used a simple syntax allowing to request the execution of several services, either independently or in the form of a pipeline, where the output of one service is the input of the next. This syntax was independent of the specific services, thus, new services could be added without having to change the monitor's core modules, like the request parser. This feature was also enabled by the introduction of a uniform internal interface to the service implementations.

For the first time, this request syntax also allowed a fully orthogonal definition of events and the actions to be executed when the event occurs. The pipelining mechanism was used to forward information on the event occurrence to the actions.

The interface still was message based, but the tools only communicated with a root monitor, which routed the request to the proper local monitors. Using this architecture, broadcast and multicast could be efficiently implemented, as communication inside the parallel system was much faster than communication with the front-end computer where the tools were executed. In order to further increase the scalability, most services could be applied to a set of objects using a single request.

Although the PARIX monitor did not provide location transparency, i.e. it required the tool to specify the proper node(s) for each request, global actions have been supported. In this case, the monitor that detected the event sent a request containing the actions to the root monitor, which forwarded it to the appropriate monitors. Thus, no explicit programming of cross-triggers was necessary, but a larger perturbation was introduced due to the need of sending and parsing a complete request.

During the first design of the monitor, a clear interface to the operating system kernel was defined, later a similar interface to the parallel run-time system was added. Since the monitor accessed the target application only via these interfaces, it could be implemented in a separate execution context and thus was protected from faulty applications.

2.3 Trends and Shortcomings

The historical review above shows several interesting trends that resulted from practical requirements of both tool designers and users.

First, the *flexibility* of the monitoring interface steadily increased. While the first versions only provided specialized services for a limited number of tools, later ones allowed the tool to program the required relationships between events and actions at run-time. However, there has to be an appreciation of values between flexibility on the one hand and low monitoring overhead on the other hand as these goals usually contradict each other.

Extensibility turned out to be another critical issue. During the further development of the tools, often new services had to be added to the monitoring system. This proved to be difficult and error prone in the first systems and was greatly simplified by splitting the monitoring system into a service independent core and the implementations of the services, which are invoked via a uniform interface.

We can also recognize a trend towards a more *global view* on the target system. The tool no longer needs to communicate with the local monitors, there is a transparent routing including broadcast and remote action execution.

Last, but not least, the *portability and adaptability* steadily increased by defining and using interfaces to the operating system and the run-time system.

However, practice and experience also showed some still existing, severe shortcomings.

Although the *monitoring interface* of the PARIX monitor in principle allowed to freely associate events with actions, the scheme could not successfully be used, since the mechanism for passing event information to the actions was inappropriate. The pipelining mechanism provides the output of one service as the input of another service. If the output is an aggregate, consisting of several pieces of information, the chance that it matches the type and format the other service expects as its input is rather low. Thus, to make the scheme usable at all, the only result provided by an event was the identifier of the object that raised it. This severely restricted the scheme's usability.

The monitoring interfaces each were targeted towards a specific parallel programming library. Especially, they were defined in terms of messages offered by this library. Thus, the interface and in turn also all the tools needed to be adapted when the target platform changed.

A final flaw of the early interfaces was their cryptic syntax, which was optimized towards easy parsing. This often made their use rather error prone.

Although the PARIX monitor offered an increased *extensibility*, adding new services still was not an easy task. Since only the interface for calling the services was defined, but no interface to the core monitoring functions existed, the implementations of the services became dependent on internals of the monitoring system. In addition, the calling interface required the services to explicitly extract their parameters from a special data structure and to check for the correct number and types of parameters. This rendered its use very complex and error prone. Finally, each extension required to recompile and relink the monitoring system.

Early monitoring interfaces did not provide *location transparency.* This means that the tools had to specify the target node for each request. One consequence is that the tool is forced to store and maintain status information on the target system, like the list of all existing objects and their location. While this is a nuisance for the early single-user, space-sharing parallel computers, where the information is more or less static, it becomes a severe problem on multi-user platforms and especially on clusters and meta-computing systems where objects can be migrated. Another consequence is limited scalability. Although the PARIX monitor offered broadcast and multicast facilities, it was still impossible to define a single request for executing a service on several but not all tasks located on different nodes.

Finally, the monitoring systems only offered *exclusive interfaces.* Although they supported several different tools, they could be used only by one tool at a time. Thus, the concurrent use of e.g. a debugger and a program flow visualizer was only possible if both tools have been combined into a single monolithic program.

3 A New View to Monitoring: Open Interfaces, Programmability, Extensibility

The old monitoring systems and lessons learned from them brought us to a point where it was inevitable to come to a new methodology of monitoring system construction. Former systems were built with a concrete solution in mind. Now we recognized that in future a more generic approach is necessary in order to meet the manifold requirements. Our observation showed that many research groups were working in this field but did not really produce comprehensive sets of tools or even powerful stand-alone tools. Furthermore, most tools were off-line tools where the construction produces considerably smaller problems. One reason for this situation lies in the complete absence of a methodology of tool construction which must be part of any theory of operating systems. It became a goal of our research to improve this situation.

Three milestones characterize our way to this goal. First, we need a concept to avoid above mentioned deficiencies of older systems. Second, a monitoring system had to be implemented following these new concepts. Finally, already existing tools, which were still of interest to the users, had to be adapted to the new monitors. The three steps will be discussed in the next sections.

3.1 OMIS — The On-Line Monitoring Interface Specification

The OMIS project was started in 1995 as a cooperation between Vaidy Sunderam from Emory University, Atlanta, Georgia, and the tools group at LRR-

TUM. The objective was to define a versatile interface between the tools and the monitoring system. The basic idea was to separate the common functionality of a monitor from the specialized functionality of a tool. In order to serve the tools' purposes, monitoring systems have to be programmable.

By separating tools and monitors we can reach three important goals: First, we can support tool interoperability. This means that different tools can be used concurrently with the same target program. This aspect will be discussed in detail in the next chapter. Second, universal sets of tools will be available for the user. Consider the situation that we do already have tools for a given target architecture. If we change to any other target architecture all tools will immediately be available after the adaptation of the monitoring system alone. No tool adaptation will be necessary. By using this concept we will third come to an easier tool construction which will in the end be an enabling technology for more and better tools for the user. Figure 1 shows the basic idea of our concept.

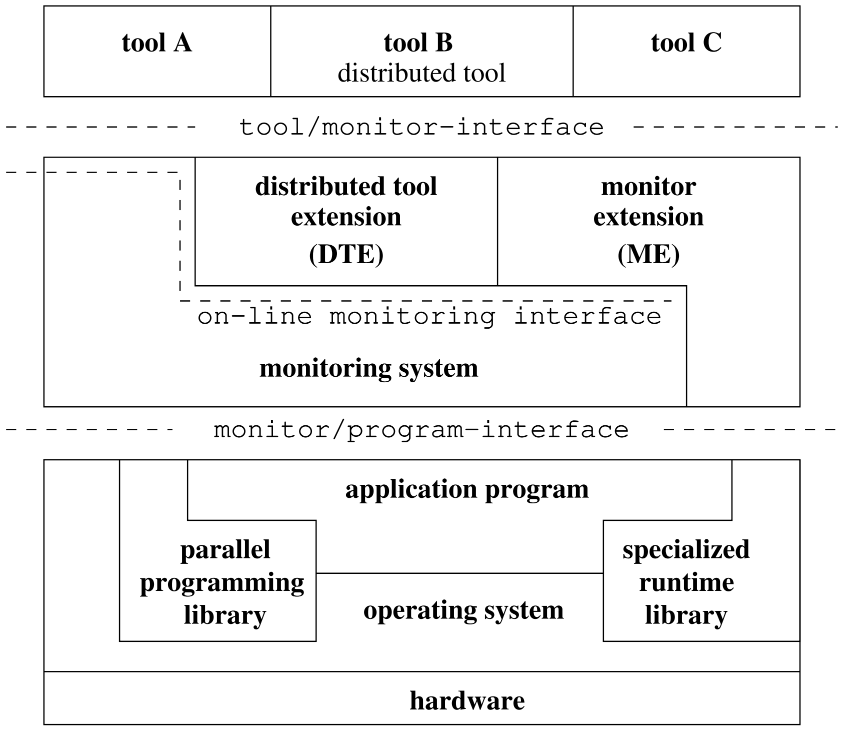

Figure 1: System model of OMIS

The upper layer shows an example of several tools where tool B should have a distributed architecture, i.e. it has no central control unit (e.g. a load balancer). All tools sit on top of the tool/monitor-interface and can thus be used concurrently. However, OMIS does not specify this interface completely, but allows for extensions to be made. OMIS only defines the syntax of the interface and a core set of general platform independent services that can be used by the tools, but also serve as a basis for the realization of extensions. In Fig. 1 this part of the interface is labeled on-line monitoring interface. Extending this interface can become necessary when a tool requires some very special functionality or when platform dependent services need to be introduced. As these services are not generic in the sense of our approach, they are not included in the OMIS core. The monitoring system has an interface to the composite block of hardware, operating system, libraries and user programs. This interface might vary from platform to platform. However, we try to use well defined standards to ease portability of the monitoring system.

What are the general requirements to be met by the tool/monitor-interface? First, its functionality must be sophisticated enough to support many different types of tools: interactive, automatic, centralized, and distributed tools for various purposes listed in earlier sections. Second, it must exhibit a mechanism for extensions. OMIS does not cover the functionality of all current and future tools, either because the functions they offer are very special or because they are unknown today. However, the interface provides a means to extend both the list of available services and the set of accessible objects. Third, the structure of the interface must be appropriate to ensure an efficient implementation. Intrusion on the system to be observed must be kept low. Here, we will discuss the first issue only; for further details please see the specification document [20].

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

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

All object classes are subject to a hierarchy where some of the concrete relations depend on the chosen programming model. With PVM, there is currently only one thread per process and message queues belong to process objects. There are other models where message queues are realized as mailboxes between processes and thus belong to nodes and not to a single process.

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

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

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

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

Two features are included in the design of the request language that are crucial for a powerful and abstract interface. First, there is a sophisticated parameter passing defined between events and actions. Result values from events triggered can now easily be passed to actions and can be used as actual parameters for their execution. Second, the basic interface does not refer to a specific programming library (e. g. PVM or MPI). Instead, the reference to function names of those libraries is made in the form of parameters used with events. Thus the interface specification is generic and library independent.

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

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

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

The results returned to the tool have a complex structure according to the number of actions contained in a request and the number of objects each service referred to. What makes it even more complex is the fact that various error conditions may exist that must be handled in an efficient and unambiguous way. For details on this issue please see the specification document [20].

3.2 OCM — An OMIS Compliant Monitoring System

After having finished version 2.0 of OMIS we started to design and implement a compliant monitoring system. The high level features of OMIS like e. g. location transparency, independence from libraries used, parameter passing between events and actions etc., resulted in a very high implementation complexity for the real monitoring system. We will just give a short overview over the basic design concept of this system which is also depicted in Figure 2.

OCM basically consists of a monitor process on each node of the target system. Each process is responsible for processing incoming requests, retrieving local information, and detecting certain local events mostly related to debugging (e. g. an application process receives a signal).

In addition, there are monitoring components contained in instrumented libraries linked with the application processes. The task of these components is to detect events within the context of the triggering application process and to execute simple actions directly in that process. In this way, we avoid time-consuming context switches which would result in excessive perturbation. Typical examples for events being detected by these components are the

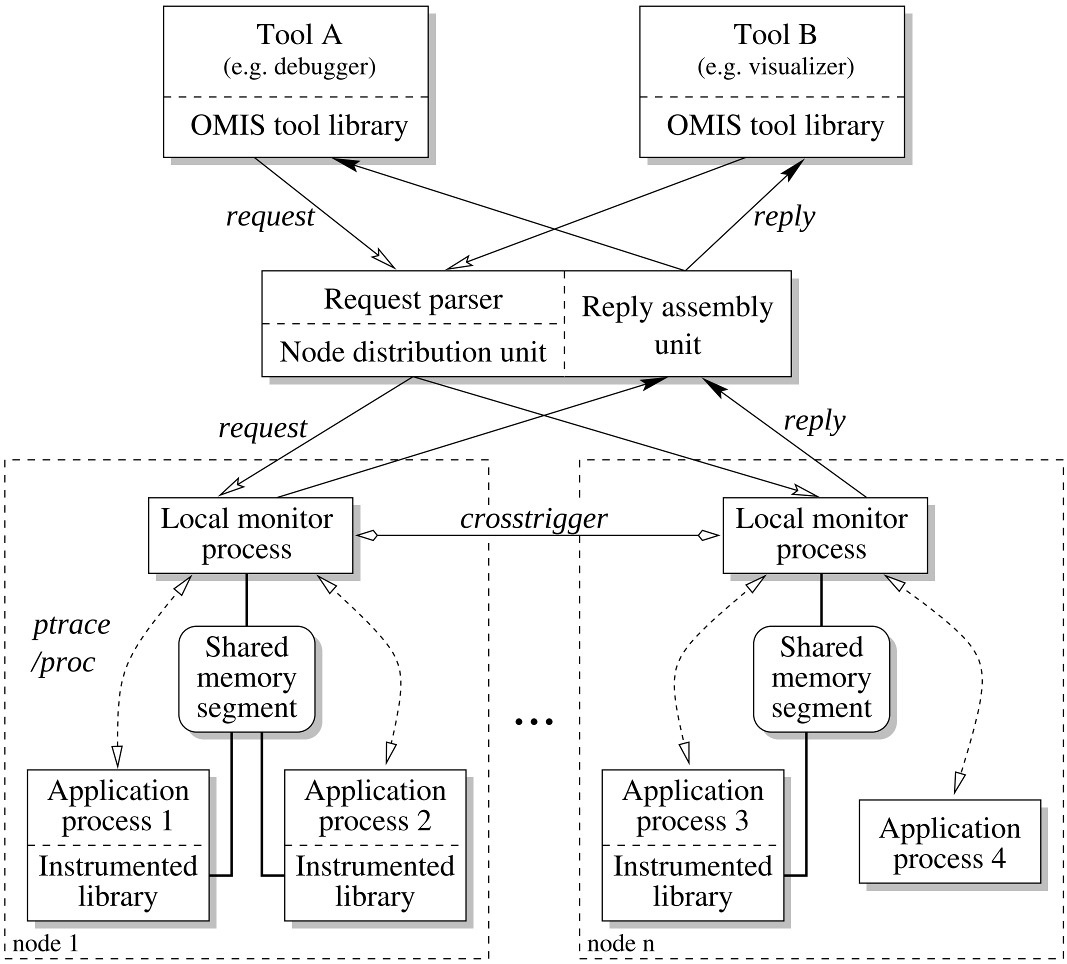

Figure 2: Coarse structure of the OCM

beginning and the end of library calls. In order to tell these monitor components what they shall do, we use a shared memory segment that stores information on which events to monitor and which actions to execute. Note that actions in the application process should not return results (except in the case of errors), since this would again result in context switches. However, they may update data stored in the shared memory segment (e.g. a trace buffer), which is read by actions executed in the monitor process upon request from a tool.

As OMIS allows a tool to issue requests that relate to multiple nodes, but each monitor process can only operate on its local node, we need an additional component called Node Distribution Unit (NDU). It analyzes each request and splits it into pieces that can be processed locally. The NDU is also responsible for assembling partial results and returning them to the tool. Currently the NDU is implemented as a central process, which communicates with the tools and the local monitor processes via message passing

3.3 The Tool-Set

Starting from OMIS and OCM we now are able to put together a first set of interoperable tools. The ideas and concepts for this were already published in 1997 under the project name The Tool-Set [33]. However, it was not before 1999 that we could present a first prototype of parts of the tool set. The reason for this does in no way lie in the complexity of tool construction - this was simplified by OMIS and OCM. It lies in the complexity of the monitoring system itself. Even though this is time consuming, the OMIS approach is very powerful, as the subsequent construction of tools is much easier than it was before. The next chapter will give details on a tool set prototype.

4 Tool Interoperability

As stated in the previous section, one of the design goals of OMIS and OCM was support for *tool interoperability*, i.e. the concurrent use of multiple run-time tools. Based on previous definitions by Fisher [8] and Wegner [30], the term *interoperability* can be formally defined as follows:

> Interoperability is the ability of independently developed, autonomous software components to cooperate at a syntactical and semantical level.

In the context of run-time tools this means:

1. Several tools can be used concurrently to observe and manipulate the same target application.
2. The tools are able to cooperate, i.e. there is an exchange of information, where the participating tools associate the exchanged data with a common meaning.
3. Even so, the tools do not need to have explicit knowledge on each other.
4. The tools do not depend on each other and can still be used as stand-alone tools.

The last two properties distinguish environments of interoperating tools from tightly integrated tool environments. Among the advantages of the former approach are better modularity and the ability to re-use existing, independently developed tools.

In order to determine the requirements that must be met to enable the design and implementation of interoperating run-time tools, and to assess the usability of the OCM for such a scenario, a case study has been conducted. The case study examined a tool environment supporting the debugging of long-running parallel programs. The conventional cyclic debugging approach regularly requires to restart the target program from its beginning, as this is the only way to examine program states previous to the point where the program is currently halted. Thus, debugging of long-running programs can be a time consuming job. Productivity could be greatly increased by a debugging environment that allows to quickly return to previous program states without need for a complete re-run. As outlined in [32], such an environment can be built by combining a parallel debugger with a checkpointing system, a program flow visualizer, and a tool for deterministic re-execution. The basic idea of operation is that during program execution, checkpoints are generated at regular intervals. At any time, the user can select a target state in the history recorded by the program flow visualizer and request the target program to be reset to this state. This is accomplished by first resetting it to the last checkpoint prior to this state and deterministically re-executing it from this point on until it reaches the target state. This is depicted in Fig. 3.

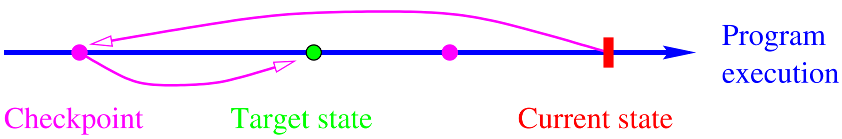

Figure 3: Resetting a target program to an arbitrary state

In the case study, we used four tools that have been developed at LRR-TUM: the debugger DETOP [21], the checkpointing system CoCheck [28], the visualizer VISTOP [31] and the deterministic execution controller codex [9]. A short discussion of the relevant implementation issues of these tools, as well as an examination of their mutual interactions is presented in [32]. The case study revealed four primary requirements that must be met in order to be able to implement environments of interoperating tools:

1. There must be a common, open interface for on-line monitoring, that is usable by all of the tools.

   This requirement is necessary to avoid structural conflicts between tools that result from incompatible monitoring techniques or the use of exclusive (operating system) interfaces for monitoring [28]. It is already met by the OCM, as it is the main motivation behind OMIS.

2. A mechanism must be provided that allows to observe events *in tools*, to request the execution of actions *in tools*, and to define a relationship between events and action executions.

   This mechanism is needed for implementing the behavioral integration between tools, which means that additional behavioral rules are defined that supplement the tools' individual behaviors [27]. For example, in our scenario, the visualizer can show information on past program states. When it does so, the debugger should mark its displayed state information as invalid, as this information relates to the current program state [32]. In general, the behavioral rules always have the form "when tool A does x, tool B must do y". Sullivan [27] investigates different methods to implement such rules. He concludes that the best suited method are *mediators*. A mediator is a first class object that encapsulates the behavioral relationships between A and B. It monitors event x in tool A and invokes y in tool B when the event occurs. If necessary, it may also perform adaptations between the interfaces of A and B, e.g. parameter translations. In comparison to other approaches, like *hardwiring*, where tool A explicitly invokes action y in tool B whenever it does x, mediators have the advantage of completely avoiding dependencies between the tools. The mediator of course needs knowledge on the involved tools, but the tools themselves need not be aware of each other.

3a. The monitoring system must enable tools to observe each modification of a monitored object performed by the monitoring system.

3b. The monitoring system must offer tools a way to intercept and possibly modify accesses to state components of monitored objects.

   These requirements address problems that occur when a tool A modifies the state of the target program, while another tool B is monitoring this state. In this case, two problematic situations can occur:

   (a) Tool B may display a cached copy of the state. In this case, B needs to be notified about the change, so it can update its display.

   (b) The modification should not be visible for other tools. In this case, tool A needs to intercept accesses to the modified state component and to manipulate their results accordingly.

The following discussion will focus on the second requirement. We will see that the advanced concepts of OMIS and OCM allow to easily provide a mechanism supporting behavioral integration of run-time tools. All we have to do is to include the tools into the set of processes monitored by the OCM. Although this might sound strange, the advantage is that we do not need additional mechanisms and – more important – we need much less changes to the tools' source code than other mechanisms. Indeed, we will see that in the ideal case, we don't need to modify the tools at all.

Therefore, we implemented a simple form of inheritance in the OCM, allowing us to have an object type "tool" that is derived from "process". As soon as a tool connects to the OCM, the OCM starts monitoring this tool, so all the OCM's process services can be applied to it. In order to implement a special action inside a tool, which can be called by the OCM, we have to create a monitor extension (see Fig. 1). The extension consists of a file with the action's interface definition, together with the code implementing the action. The interface definition is compiled into a shared library that is dynamically loaded into the OCM's monitor processes, while the action's code is compiled into another shared library dynamically loaded into the tool. This results in the action's code being able to invoke any public function and to access any public data structure within the tool. Thus, in all cases where we just have to add code to a tool in order to implement the desired action, there is no need to modify the tool's source code or executable.

Now, the OCM knows about the action's interface and its code has been plugged into the tool. But how can the OCM force the tool to execute the action? We use a kind of a Trojan horse approach. The standard OMIS interface requires the tool to listen at a file descriptor and to call a handler function within the OMIS library whenever there is data to read at this file descriptor[1]. We extended this handler function to also check whether the OCM requests an action to be called inside the tool and to call it if necessary.

For detecting events inside a tool, the OCM uses the same mechanism that allows it to detect the entry into and the exit from library functions in normal target processes (see Section 3.2). The OCM originally used *binary wrapping* [5] to wrap additional code around the library functions that performs the event detection and the execution of the associated actions. This method has the drawback that the tools have to be relinked. In a cooperation with the University of Wisconsin-Madison, USA[2], we therefore have extended the OCM to exploit the dynamic instrumentation technique provided by the

[1]This is necessary, as the replies to the tool's requests have to be received asynchronously.

[2]Partly funded by the German Science Foundation, Contract WI1346/1-1.

dyninstAPI [11]. This API allows to dynamically patch the code of the target processes, enabling the OCM to insert the code for notifying function entries or exits without a need for a special linkage or any other preparation of the processes' executables.

Given these mechanisms for event detection and action execution, together with the monitoring interface that allows to program event/action relations, the OCM now supports the mediator model for behavioral integration developed in [27]. The behavior can be explicitly defined by a "mediator tool" that sends the proper requests to the monitoring system. If necessary, OCM extensions can be used to handle interface adaptations. In contrast to other approaches, the OCM can provide the necessary coordination without a need to modify the tools' code, provided that the relevant events are associated with the beginning or end of a function and the necessary actions can be implemented by merely adding code to the tool. From our current experience with implementing the scenario presented above, we expect this to be the case for most well structured tools, but there is no empirical proof yet. However, even if the tool needs to be adapted, the necessary adaptations do neither create a dependency to other tools nor to a specific coordination infrastructure (as e.g. the insertion of calls to the ToolTalk library [13] would), since all the infrastructure specific code is plugged into the tool at run-time.

It is interesting to notice that this solution relies on all of the basic design principles of the OCM: the object-oriented interface, the event/action paradigm, location transparency, and dynamic extensibility. A working prototype of the proposed tool environment has already been implemented. It demonstrates the flexibility and effectiveness of the OMIS/OCM approach for monitoring parallel and distributed systems.

5 Adaptation to new Architectural Paradigms

The OMIS core and its implementation are completely independent from the programming environment. They are always the same, independent from whether a message passing or a shared memory environment has to be monitored. Since the core only has generic interfaces to node local resources, it can be directly used without change. Programming environment dependent functionality is provided by means of extensions.

These aspects can be demonstrated through the adaptation of the OMIS approach to new architectural paradigms. The SMiLE cluster architecture [1] and software infrastructure [16] developed at LRR-TUM is a state-of-the-art cluster platform built of commodity off-the-shelf compute nodes and a high-speed interconnection network fabric, namely the Scalable Coherent Interface (SCI, IEEE Std. 1596-1992) [12]. SCI facilitates communication via distributed shared memory as individual load and store operations are turned into remote memory accesses. Thus, the SMiLE cluster architecture represents a parallel system with NUMA characteristics (Non-Uniform Memory Access). Fast communication at user level can be provided in a VI-Architecture style through exploiting SCI's hardware supported DSM. This approach is the base for the SMiLE software infrastructure enabling both message passing and shared memory programming.

The main prerequisite for shared memory programming on clusters is a fully transparent, global memory providing applications with the abstraction of a cluster-wide global process and a single virtual address space. Within the SMiLE project, such a layer, the SCI Virtual Memory or SCI-VM [25] has been developed. It not only provides the required abstractions, but also forms the basis for any kind of shared memory programming model on top of SCI-based PC clusters [25, 26].

In order to most efficiently utilize the given HW-DSM features of the SCI interconnection fabric, the SCI-VM layer is built as hybrid hardware/software system. While the management and configuration tasks are carried out in software in a DSM-like fashion, the actual communication is handled in hardware by SCI's HW-DSM mechanism. The memory is distributed at page granularity and these distributed pages are then combined into a global virtual address space. This is achieved by mapping local pages conventionally through the processor's memory management unit (MMU) and remote pages through SCI's HW-DSM mappings. There is no need for any page replication or special multiple writer protocols. This eliminates many of the typical SW-DSM bottlenecks like false sharing and complex differential page update protocols.

The performance of such a DSM system depends on the efficient use of local memory accesses. Due to the NUMA characteristics of the SCI cluster, any access to remote memory has a latency overhead of about one to two orders of magnitudes compared to accesses to local memory. For a good overall application performance, it is therefore mandatory for the application to exhibit good locality properties. Unlike in message passing systems, however, where any communication is done explicitly at a priori known locations, shared memory system are based on implicit communication triggered by processor reads and writes. It is therefore not predictable and can only be monitored with the help of specialized hardware, like the SMiLE monitor described in detail in [10, 14], capable of snooping transactions on the interconnection network. With the help of the information acquired with such systems and after their transformation to a higher level of abstraction equal to the one of the original application source, it is then possible to incrementally optimize the application's behavior statically (through different data and tasks assignments to nodes) and dynamically (through runtime mechanisms like data and thread migration). In both cases, however, the impact on the application source code itself is minimal, as only the tuning of a few parameters is required. This maintains the comfortable shared memory programming model for the user, but potentially results in significantly higher performance. An example how to optimize data locality for SCI-based PC clusters with the SMiLE monitoring approach in principle is presented in [15].

The monitoring system utilizes a layered approach. Each of these layers contributes information to the monitoring system resulting in a flexible hierarchical structure which enables the transformation of acquired low–level information into a user readable form at application level. The basic monitoring architecture and the corresponding extensions of our SCI-DSM monitor system are depicted in Figure 4.

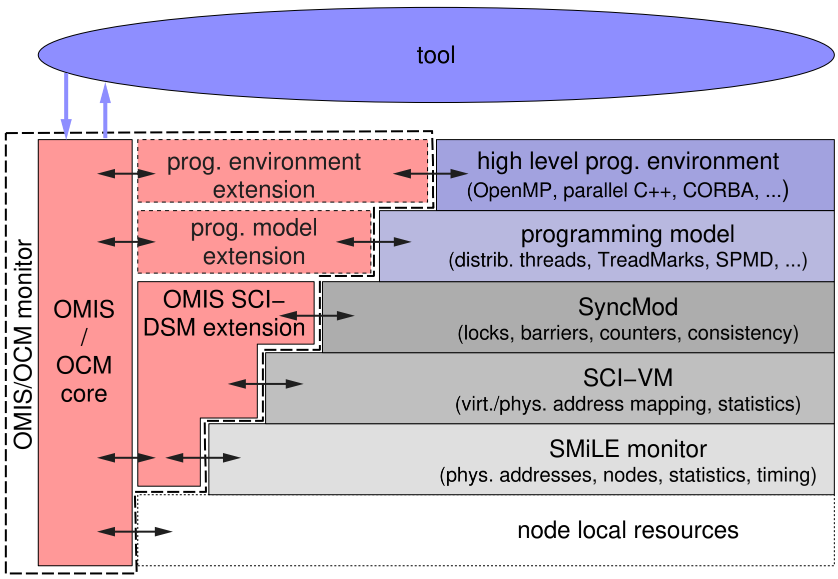

Figure 4: Multi-layer SMiLE DSM monitoring with OMIS/OCM

The lowest layer in the system is represented by the SMiLE hardware monitor and its low-level driver software as it has been described above. The information delivered is very detailed, low-level, and purely based on physical address information.

The main information missing for an evaluation of the data acquired from the SMiLE hardware monitor is the translation of physical addresses seen by the monitor into virtual addresses visible from the application processes. In a system with a global virtual memory abstraction, as discussed in the work, this translation is controlled by the DSM layer, in this case the SCI-VM. It extends the virtual memory management of the operating system to a cross-node memory resource control and is responsible for setting up the correct virtual address mappings to both local and remote memory. It therefore is able to directly provide the required information to the monitor system.

Besides the pure mapping information, the DSM layer can provide further information that can aid in performance evaluations of shared memory applications. It can be extracted from statistics collected during the dynamic events of the SCI-VM and includes data about mapping invalidations, mapping misses, and access history. This forms the basis for an in depth investigation of the application's memory access patterns and therefore provides an important contribution to the monitoring system.

Synchronization primitives generally have a critical performance impact on shared memory applications. In order to enable the user of the monitoring system to observe their impact, a module (the *SyncMod*) containing generic synchronization constructs like locks, barriers, and counters delivers various statistical information to the monitoring system.

Various higher level programming models like globally distributed threads or hardware assisted TreadMarks-like shared memory can be implemented on top of them. In the case of very high level distributed programming environments like Split-C, parallel C++, or distributed CORBA objects, another layer can be added above the programming model.

In a complex layered model like the above one monitoring data have to be gathered at various levels and combined to give a single global image. Similarly, configuration information for the SMiLE hardware monitor has to be translated, only in opposite order. The decision about what parts of the application are most interesting to monitor in detail is often made at the abstraction level of the application. The corresponding information then has to be translated onto low level information like hardware addresses layer by layer. This is not possible with simple traditional monitoring approaches. However, the OMIS compliant monitoring system OCM provided an excellent basis that could easily be adapted to monitor our SMiLE environment.

The OCM core covers all generic node local resources. One monitor extension, the OMIS SCI-DSM extension, is responsible for monitoring and controlling the lower three layers of the system. It uses interfaces provided by the OCM core plus interfaces from the SMiLE monitor drivers, the SCI-VM, and the SyncMod layers. Statistical and performance data are obtained from the SMiLE hardware monitor. As a platform specific extension, the SCI-DSM extension includes knowledge about the special semantics and utilizes memory mapping information from the SCI-VM layer to map this data from physical to logical cluster addresses. In addition, it gathers statistical data about address translations. Similarly, data obtained from the SyncMod layer is related to the low level performance and statistical data from the hardware monitor. Thereby, semantically higher level and more useful performance data is computed.

The resulting higher level information can then be obtained from the OCM via standard monitor services that are implemented by the SCI-DSM extension. The interface is identical to the one outlined in Section 3.1.

To configure the SMiLE hardware monitor, information can also be downloaded from a tool via the standard OMIS monitoring interface similarly as it is e. g. done with source line information obtained from the symbol table in the DETOP debugger extension. This information can be specified at the higher level and is automatically transformed to its low level specification and sent to the SMiLE hardware monitor by the OMIS SCI-DSM extension. To monitor the higher level layers, like distributed threads or parallel C++, additional monitor extensions can be then brought into the OCM. These utilize the already present services from the underlying OMIS SCI-DSM extension by calling them via functionality provided by the OCM core. These extensions can provide abstractions like cluster global processes, distributed threads, or C++ objects.

6 Related Work

For such a comprehensive research field like monitoring systems, on-line tools, and tool integration there is of course much related work to be considered. We will mention here only a few aspects which are related to our state-of-the-art research question: How can monitoring systems enable tool interoperability? Thus we refer to related work in the field of interoperability and of monitoring systems.

The term interoperability is used in many circumstances, most of which don't relate to our area of focus. Several standards exist that address interoperability in the wider sense, but are not usable for on-line monitoring. *CORBA* (Common Object Request Broker Architecture) [22] is a middleware architecture addressing interoperability among distributed object oriented software components. *ToolTalk* [13] is a mechanism that enables different applications to communicate and thereby interoperate. However, CORBA and ToolTalk only address application level aspects and are not sufficient for the purpose of monitoring. They could be used by interoperable tools to interact at the user level, after the problems at the monitoring level have been solved. *PCTE* (Portable Common Tool Environment) [7] is a framework to support integrated software engineering tool environments. It defines how different tools may share common objects during development. Its support to access *running* applications is neglectable and it is not usable for interoperable on-line monitoring. Therefore, while technologies aimed at interoperability exist, they address scenarios different from the monitoring situation we are addressing. Nevertheless we can adopt ideas that refer to information consistency in general as this is a crucial aspect for interoperability.

Besides OMIS/OCM there are two other environments and approaches that address aspects of on-line monitoring and interoperability: These are DAMS and DPCL.

DAMS (Distributed Applications Monitoring System) [6] is a distributed monitoring system where a server runs on each node to be controlled and clients (tools) connect to a central service manager. The individual components of DAMS exhibit well defined interfaces for communication. Thus, a multi-vendor environment could be supported. DAMS is configurable with respect to monitoring services offered. Although it allows multiple clients to attach to the service manager and also partly supports requirement 3a) outlined in Section 4, it does not integrate interoperability concepts that satisfy our remaining requirements of consistent tool usage.

The DPCL approach (Dynamic Probe Class Library) [24] is an effort to provide an API for simplifying tool construction. DPCL daemons act as node local monitoring systems. They are controlled by the DPCL library which gets linked to the user's tools. DPCL is by itself not yet a full distributed monitoring system. The merge of the node local views has to be provided by the tools themselves. Consequently, as it does not cover the higher levels of abstraction of our infrastructure layer it does also not support tool interoperability. DPCL concentrates on performance analysis but its functionality allows also any type of manipulation services.

For further details on related work please refer to publications about OMIS/OCM and interoperability, e. g. [17, 29]

7 Lessons Learned

More than 10 years of developing monitoring systems for parallel and distributed computers led to a couple of experiences in designing and implementing these systems.

Since monitoring is inherently event-oriented, the *event/action-paradigm* seemed to be a well suited basis for monitoring interfaces from the very beginning. However, in practice the concept only becomes useful when the interface offers a *carefully designed request language.* Especially, flexible mechanisms for passing parameters from events to actions, as well as for grouping of actions must be provided. Only under these conditions the goals of the event/action-paradigm, like easy programmability, reuse of events and actions, and versatility of the interface can be achieved. Finally, it turned out that the *readability of the request syntax* for humans significantly influences the interface's usability.

A second vital issue is *scalability.* For reasons of efficiency and ease of use, a monitoring interface should allow a request to affect a whole set of objects instead of only a single one at a time. Just providing services that can operate on object lists is not enough. First, efficient ways of denoting sets like 'all monitored processes' are needed, especially since these sets are dynamic in nature. OMIS addresses this issue by the concept of token conversions. Second, as has been learned from the PARIX monitor and OMIS 1.0 [19, 18], object lists don't make sense without *location transparency*, as the lists should be allowed to contain objects located on different nodes.

By their nature, monitoring systems are strongly interconnected with their target system. This makes the design of a portable and adaptable system a challenge. While operating system dependencies can be avoided by using or defining standardized low-level interfaces, adaptability w.r.t. different programming libraries and models requires further concepts. OMIS utilizes an *object based model* of the target system to clearly separate universal services from services depending on the programming model. From the 85 services defined by OMIS 2.0, only 6 fall into the latter category. To achieve the same modularity in the implementation, too, it is divided into a core management module and *extensions*, which provide the actual services. Extensions are described using an interface definition language and can be plugged-in at run-time. Thus, not only services adapting the OCM to other target platforms, like the SMiLE cluster architecture can easily be added, but also special services required by some tools. Last but not least, this plug-in architecture supports interoperability by allowing to implement the mediator concept, without forcing the OCM to have an a-priori knowledge on all the tools. Actions and events in each tool can just be defined as services in a special extension, that is dynamically loaded on demand. The experience with implementing support for interoperating tools, however, showed that OMIS should better be based on a fully *object oriented model* of the target system, as we need to introduce new object classes that are derived from existing ones.

An important lesson learned about the implementation of monitoring system and tools was to make them *as asynchronous as possible*. This means that tools must not block while waiting for replies; a monitor must never block waiting for an event. Blocking not only results in very bad performance, but also prevents a timely reaction to other events and can even lead to deadlocks. Since until recently, multi-threading still suffered from portability problems as well as non thread-safe libraries and operating system interfaces, in the OCM and the accompanying tools we use a data-flow based callback mechanism. It can avoid most, but not all blocking situations. As multi-thread support is getting more mature, new developments should be multi-threaded from the beginning.

8 Future Research

Research done in the last years concentrated on parallel programming environments mainly with message passing but also with distributed shared memory programming. Future research efforts are aimed towards object oriented environments.

We will adapt monitoring technologies as well as tools to CORBA based environments. Monitoring is investigated in the MIMO project (Monitoring and Managing Distributed Middleware Environments) [35]. Here, a multi-layer monitoring is specified that is mapped onto the hierarchical layers within CORBA environments. As a result, monitoring can be performed at various levels of abstraction. As a first automatic tool in this environment we will adapt load balancing technologies to CORBA.

References

[1] G. Acher, W. Karl, and M. Leberecht. *SCI: Scalable Coherent Interface Architecture and Software for High-Performance Compute Clusters*, volume 1734 of *Lecture Notes in Computer Science*, chapter The TUM PCI/SCI Adapter, pages 89–101. Springer, Berlin, 1999. `http://www.springer.de/cgi-bin/search_book.pl?isbn=3-540-66696-6`.

[2] T. Bemmerl and A. Bode. An Integrated Environment for Programming Distributed Memory Multiprocessors. In A. Bode, editor, *Distributed Memory Computing – 2nd European Conference, EDMCC2*, volume 487 of *Lecture Notes in Computer Science*, pages 130–142, Munich, Germany, Apr. 1992. Springer.

[3] T. Bemmerl, A. Bode, P. Braun, O. Hansen, T. Treml, and R. Wismüller. The Design and Implementation of TOPSYS. SFB-Bericht 342/16/91 A, Technische Universität München, July 1991. `http://wwwbode.in.tum.de/~wismuell/pub/design.ps.gz`.

[4] T. Bemmerl, R. Lindhof, and T. Treml. The Distributed Monitoring System of TOPSYS. In H. Burkhart, editor, *CONPAR 90 – VAPP IV, Joint Intl. Conf. on Vector and Parallel Processing*, volume 457 of *Lecture Notes in Computer Science*, pages 756–765, Zurich, Switzerland, Sept. 1990. Springer.

[5] J. Cargille and B. P. Miller. Binary Wrapping: A Technique for Instrumenting Object Code. *ACM SIGPLAN Notices*, 27(6):17–18, June 1992.

[6] J. Cunha and V. Duarte. Monitoring PVM Programs Using the DAMS Approach. In V. Alexandrov and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Messag Passing Interface, Proc. 5th European PVM/MPI Users' Group Meeting*, volume 1497 of *Lecture Notes in Computer Science*, pages 273–280, Liverpool, UK, Sept. 1998. Springer Verlag.

[7] European Computer Manufacturers Association (ECMA). Portable common tool environment: Abstract specification (standard ECMA-149), June 1993. `ftp://omg.org/pub/PCTE/ECMA_PCTE/`.

[8] G. E. Fisher. A CSL View of Applications, Portability, Scalability, and Interoperability. *StandardView*, 4(2):107, June 1996.

[9] M. Frey and M. Oberhuber. Testing and Debugging Parallel and Distributed Programs with Temporal Logic Specifications. In *Proc. of 2nd Workshop on Parallel and Distributed Software Engineering 1997*, pages 62–72, Boston, May 1997. `http://wwwbode.in.tum.de/~oberhube/PUBS/PS/pdse97.ps.gz`.

[10] R. Hockauf, W. Karl, M. Leberecht, M. Oberhuber, and M. Wagner. Exploiting Spatial and Temporal Locality of Accesses: A New Hardware-Based Monitoring Approach for DSM Systems. In D. Pritchard and J. Reeve, editors, *Euro-Par'98 Parallel Processing, 4th International Euro-Par Conference, Southampton, UK, September 1-4, 1998 Proceedings*, volume 1470 of *Lecture Notes in Computer Science*, Berlin, Sept. 1998. Springer Verlag.

[11] J. K. Hollingsworth and B. Buck. *DyninstAPI Programmers' Guide, Release 1.2.* Univ. of Maryland, College Park, MD 20742, Sept. 1998. http://www.cs.umd.edu/projects/dyninstAPI/dyninstProgGuide.pdf.

[12] IEEE Standard for Scalable Coherent Interface (SCI). IEEE Std 1596-1992, 1993. The Institute of Electrical and Electronics Engineers Inc. (IEEE), 345 East 47th Street, New York, NY 10017-2394, USA.

[13] A. M. Julienne and B. Holtz. *ToolTalk & Open Protocols: Inter-Application Communication.* SunSoft Press, 1994. ISBN 0-13-031055-7.

[14] W. Karl, M. Leberecht, and M. Oberhuber. *SCI: Scalable Coherent Interface Architecture and Software for High-Performance Compute Clusters*, volume 1734 of *Lecture Notes in Computer Science*, chapter SCI Monitoring Hardware and Software: Supporting Performance Evaluation and Debugging., pages 89–101. Springer, Berlin, 1999. http://www.springer.de/cgi-bin/search_book.pl?isbn=3-540-66696-6.

[15] W. Karl, M. Leberecht, and M. Schulz. Optimizing Data Locality for SCI-Based PC-Clusters with the SMiLE Monitoring Approach. In *Proceedings 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT'99) October 12-16,1999 Newport Beach, CA*, pages 169–176. IEEE Computer Society, 1999.

[16] W. Karl, M. Leberecht, and M. Schulz. Supporting Shared Memory and Message Passing on Clusters of PCs with a SMiLE. In A. Sivasubramaniam and M. Lauria, editors, *Network-Based Parallel Computing: Communication, Architecture, and Applications / Third International Workshop, CANPC'99, Orlando, Florida, USA, January 9th, 1999*, volume 1602 of *LNCS*, Heidelberg, Jan. 1999. Springer Verlag.

[17] T. Ludwig, J. Trinitis, and R. Wismüller. Synergetic Tool Environments. In V. Malyshkin, editor, *Parallel Computing Technologies, Proc. PaCT'99*, volume 1662 of *Lecture Notes in Computer Science*, pages 248–257, St. Petersburg, Russia, Sept. 1999. Springer Verlag.

[18] T. Ludwig, R. Wismüller, M. Oberhuber, and A. Bode. An Open Interface for the On-Line Monitoring of Parallel and Distributed Programs. *Intl. Journal of Supercomputer Applications and High Performance Computing*, 11(2):160–174, Summer 1997.

[19] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. OMIS – On-line Monitoring Interface Specification. Technical Report TUM-I9609,

SFB-Bericht Nr. 342/05/96 A, Technische Universität München, Feb. 1996.
http://wwwbode.in.tum.de/~omis/OMIS/Version-1.0/version-1.0.ps.gz.

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

[21] M. Oberhuber and R. Wismüller. DETOP - An Interactive Debugger for PowerPC Based Multicomputers. In P. Fritzson and L. Finmo, editors, *Parallel Programming and Applications*, pages 170–183. IOS Press, Amsterdam, May 1995.
http://wwwbode.in.tum.de/~wismuell/pub/zeus95.ps.gz.

[22] OMG (Object Management Group). The Common Object Request Broker: Architecture and Specification — Revision 2.2. Technical report, February 1998.

[23] C. M. Pancake and C. Cook. What users need in parallel tool support: Survey results and analysis. In IEEE, editor, *Proceedings of the Scalable High-Performance Computing Conference, May 23–25, 1994, Knoxville, Tennessee*, pages 40–47, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994. IEEE Computer Society Press.

[24] D. Pase. Dynamic Probe Class Library (DPCL): Tutorial and Reference Guide, Version 0.1. Technical report, IBM Corporation, Poughkeepsie, NY, 1998.
http://www.ptools.org/projects/dpcl/tutref.ps.

[25] M. Schulz. SCI-VM: A flexible base for transparent shared memory programming models on clusters of PCs Martin Schulz. In J. Rolim, F. Müller, and et. al., editors, *Parallel and Distributed Computing / Proceedings of HIPS '99*, volume 1586 of *Lecture Notes in Computer Science*, pages 19–33, Berlin, 1999. Springer Verlag.

[26] M. Schulz, M. Völk, W. Karl, F. Munz, and S. Ziegler. Running a spectral analysis code on top of SCI shared memory using the TreadMarks API. In G. Horn and W. Karl, editors, *Proceedings of SCI-Europe '99, The 2nd international conference on SCI–based technology and research*,

pages 35–43. SINTEF Electronics and Cybernetics, Sept. 1999. ISBN: 82-14-00014-9.

[27] K. J. Sullivan. *Mediators: Easing the Design and Evolution of Integrated Systems.* PhD thesis, Dept. of Computer Sciences and Engineering, Univ. of Washington, 1994. Technical Report 94-08-01. `ftp://ftp.cs.washington.edu/tr/1994/08/UW-CSE-94-08-01.PS.Z`.

[28] J. Trinitis. *Interoperability of Distributed Checkpointing and Debugging Tools*, volume 19 of *LRR-TUM Research Report Series.* Shaker Verlag, Aachen, 2000. ISBN 3-8265-1858-6. Ph.D. Thesis, Fakultät für Informatik, Technische Universität München, 1999.

[29] J. Trinitis, V. Sunderam, T. Ludwig, and R. Wismüller. Interoperability support in distributed on-line monitoring systems. In M. Bubak, H. Afsarmanesh, R. Williams, and B. Hertzberger, editors, *Proceedings of High Performance Computing and Networking — 8th International Conference, HPCN Europe 2000, Amsterdam, The Netherlands*, volume 1823 of *Lecture Notes in Computer Science*, pages 261–269, Berlin, April 2000. Springer.

[30] P. Wegner. Tutorial Notes: Models and Paradigms of Interaction. Technical Report CS-95-21, Department of Computer Science, Brown University, Providence, Rhode Island 02912, Sept. 1995. `http://www.cs.brown.edu/publications/techreports/reports/CS-95-21.html`.

[31] R. Wismüller. State Based Visualization of PVM Applications. In A. Bode, J. Dongarra, T. Ludwig, and V. Sunderam, editors, *Parallel Virtual Machine – EuroPVM'96*, volume 1156 of *Lecture Notes in Computer Science*, pages 91–99, Munich, Germany, Oct. 1996. Springer Verlag. `http://wwwbode.in.tum.de/~wismuell/pub/europvm96.ps.gz`.

[32] R. Wismüller. Interoperability Support in the Distributed Monitoring System OCM. In *Proc. 3rd International Conference on Parallel Processing and Applied Mathematics - PPAM'99*, pages 77–91, Kazimierz Dolny, Poland, Sept. 1999. Technical University of Czestochowa, Poland. `http://wwwbode.in.tum.de/~wismuell/pub/ppam99b.ps.gz`.

[33] R. Wismüller, T. Ludwig, A. Bode, R. Borgeest, S. Lamberts, M. Oberhuber, C. Röder, and G. Stellner. The Tool-set Project: Towards

an Integrated Tool Environment for Parallel Programming. In X. De, K.-E. Großpietsch, and C. Steigner, editors, *Proceedings of Second Sino-German Workshop on Advanced Parallel Processing Technologies, APPT'97*, pages 9–16, Koblenz, Germany, Sept. 1997. Fölbach. `http://www.in.tum.de/~ludwig/LN/papers/tl9703/PS/article.ps.gz`.

[34] R. Wismüller, M. Oberhuber, J. Krammer, and O. Hansen. Interactive debugging and performance analysis of massively parallel applications. *Parallel Computing*, 22(3):415–442, Mar. 1996. `http://wwwbode.in.tum.de/~wismuell/pub/pc95.ps.gz`.

[35] I. Zoraja, G. Rackl, and T. Ludwig. Towards monitoring in parallel and distributed environments. In D. B. N. Rozic, P. I, and B. Burmaz, editors, *Proceedings of the International Conference on Software in Telecommunications and Computer Networks (SoftCom'99)*, pages 133–141. Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture University of Split, Split, Croatia, 1999.