Interoperability Support in the Distributed Monitoring System OCM*

Roland Wismüller [a]

[a]Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Institut für Informatik, Technische Universität München
D-80290 München, Germany
wismuell@in.tum.de

Abstract

In order to develop and maintain parallel and distributed software systems, tools that observe and manipulate the run-time behavior of these systems are indispensable. Often it would even be preferable use multiple tools at the same time in order to have enhanced functionality. Today, tools allowing such a simultaneous use only exist in the form of monolithic integrated environments produced by a single vendor. As soon as we want to combine independent tools we are faced with severe problems.

The paper presents the features included in the OMIS compliant monitoring system OCM that help to solve these problems. We derive the necessary features by looking at an example scenario, where several tools are used together to provide an efficient debugging environment, allowing the user to reset the debugged program to any previously executed state. This scenario also shows that tool interoperability is not only a feature by itself, but also leads to an added value for the user.

1 Introduction

The work presented in this paper is centered around two concepts: run time tools and interoperability. So before going into more detail, we will try to define our view of these concepts and give a motivation for our work.

Run Time Tools. In general, a run time tool observes and/or manipulates the execution of a software system. We distinguish between two classes: tools that operate off-line, and tools operating on-line. *Off-line tools* are applied only after a program execution has finished. They use raw data acquired and stored during the execution as their information base. Typically, this data has the form of a trace file, containing a sequence of time-stamped events. An event entry can contain e.g. information about a function call and its parameters. This data can be used to visualize the program behavior as well as to detect causes for programming errors.

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

In *on-line tools*, data analysis and display is done while the program under investigation is still executing. On-line tools offer the advantage that they provide the user with an immediate feedback on the program behavior and the ability to change the type of data being monitored at any time, based on this feedback. Even more, on-line operation allows the tool to manipulate the program's execution if there is a need. Manipulation can be initiated by the user, e.g. temporarily stopping a process during debugging, or can be performed by the tool automatically, based on the observed program behavior, e.g. the migration of a process for load balancing purposes.

Both kinds of tools are usually based on some module that is responsible for acquiring the necessary data during run-time (and may perform manipulations). We will call this module a *monitoring system*. Note that the monitoring system itself is always operating on-line (even in the case of an off-line tool). Thus, our following statements on interoperability problems of on-line tools apply to the monitoring systems of off-line tools as well. Still, monitoring systems for off-line tools are much easier to build as they only observe the program and store the data.

Interoperability. In the context of run-time tools, interoperability simply means that we can apply two or more independent tools to the same software system at the same time and in some meaningful way.[2] Today, the application of multiple tools to the same target system is only possible, if these tools form a single *integrated* environment. This means that each tool "knows" about the other tools, and also "knows" their implementation. The drawback is that the tools' implementations strongly depend on each other, resulting in a monolithic structure that is difficult to change or enhance.

The simultaneous use of multiple, independent tools offers advantages both for tool developers and users. A tool set with interoperable tools can be built in a much more modular way than integrated tool sets. Even the functionality of existing tools could be split into several parts, allowing to "factor out" dependencies from the target platforms. For example, a parallel debugger that allows to inspect the messaging system not only depends on the operating system, but also on the message passing library. Splitting it into a traditional multi-process debugger and a message queue manager like MQM[1] results in tools with a much broader market.

For the user, interoperable tools offer the advantage that he is no longer bound to a specific set of tools as in an integrated tool set, but may choose the individual tools that best fit his needs. The concurrent use of tools can also result result in synergetic effects. Consider a performance analysis tool and a process migration based load balancer. When the load balancer notifies the performance analyzer of migration decisions, the latter could visualize migrations in its graphical displays. On the other hand, the user could trigger additional migrations manually via the performance analysis tool when the displayed performance values show an unsatisfactory behavior.

Interoperability is talked about a lot. However, with parallel on-line tools there are virtually no realizations up to now, as the essential questions are only started to be tackled. The first problem is to run the tools concurrently. This is only possible if

[2] For off-line tools, this means that we apply the tools' (different) monitoring systems at the same time, *not* that we analyze an already generating trace with multiple tools.

supported by an appropriate infrastructure. The second question is how to make the tools interoperate. Not only do we require that they do not negatively influence each other. Even more they should yield synergetic effects, maybe not only by their pure co-existence but also by additional functionality that refers to other tools.

In the remainder of the paper, we present a representative scenario for the concurrent use of tools. By looking at some critical interactions between these tools, we derive requirements that should be met by infrastructures supporting interoperable tools. Finally, we present the concepts that are integrated into the distributed on-line monitoring system OCM in order to meet these requirements.

2 A Scenario for Interoperable Tools

In order to analyze the needs of interoperable tools, we will now examine a concrete set of tools, where interoperability yields synergetic effects: the combination of a debugger and a checkpointing system. Here we have the most exciting case where multiple tools manipulate the target system.

2.1 Debugging Programs with a Long Execution Time

In general, debugging is a cyclic process: When the effect of a bug has been observed during testing, the programmer runs the program with a debugger, trying to isolate the location of the bug's cause by repeatedly stopping the program and inspecting its state. When a deviation from the expected state is observed, the cause of the bug lies between the current stopping location and the previous one. This means, however, that in order to get nearer to the bug's cause, the program needs to be re-run. The approach therefore reaches its practical limits when a bug occurs near the end of a program's execution that runs for a very long time. In such cases, productivity could be greatly increased by a debugging tool that allows to quickly return to past states of the program without a need for a complete re-run.

It is possible to realize this functionality by combining a debugger and a checkpointing tool. There already exist some debugging tools with integrated checkpointing for sequential programs, e.g. Cray's CDBX debugger [2]. However, we are focusing on distributed systems and on interoperability of independent tools rather than tight integration. The idea of the operation of the combined tools is depicted in Fig. 1. The checkpointer periodically saves the state of the debugged program. When the programmer wants the execution to return to a selected state, the program is actually

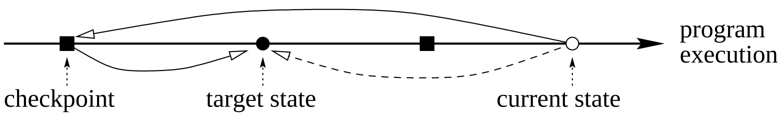

Figure 1: Resetting a program to a target state. Re-executing from a checkpoint until the target state is reached (solid arrows) is equivalent to a direct reset (dashed arrow)

reset to the immediately preceding checkpoint and is then executed until it reaches the target state. For this idea to work in practice, two additional issues must be addressed:

1. How shall the programmer specify the desired state?
2. Distributed programs may be nondeterministic due to the unpredictable order in which messages from different senders arrive at a process. The program may therefore exhibit a different behavior when it is restarted after the checkpoint, so it may not reach the target state.

The first problem can be addressed by adding a program flow visualizer to the tool environment. The visualizer records a sequence of program states at a coarse grained level. It allows the programmer to browse through this sequence, selecting one state at a time that is displayed in more detail. This state can be used as the target state for the program restart. To solve the second problem, we need an additional tool that enforces deterministic execution.

In the next subsections, we will briefly discuss the four tools making up our environment. The presentation is limited to those features that are essential for the following examination of the tools' interactions. This examination will lead to a number of requirements for infrastructures supporting interoperable run-time tools. We will not address other implementation issues of the tool environment in this paper, e.g. how to detect that the program has reached the target state.

2.2 The Individual Tools

To implement the outlined environment, we will use tools developed at LRR-TUM. The advantage of this approach is that we have a detailed knowledge on their implementations, enabling us to accurately derive the mutual interactions. This may seem to contradict our goal of examining the interoperability of independent tools developed by different vendors. However, the tools have been programmed by different groups in the framework of different projects, so they really can be viewed as a representative multi-vendor multi-tool environment.

2.2.1 DETOP: Parallel Debugging

DETOP supports the debugging of parallel and distributed programs using the PVM programming library. It offers the usual set of debugging commands to control, inspect and modify the executions of multiple processes at the source code level (see [3]). However, as it is the case with virtually all other parallel debuggers, DETOP does not interact with the programming library, e.g. to show the state of message queues or barriers. The only, but important exception is the fact that the debugger displays the processes' PVM task IDs in order to help the user in identifying his processes.

The implementation of DETOP uses a process structure that is typical for most tools for distributed systems. Besides a central component, which controls the user interface, there is a distributed monitoring system with a local monitor process (often called a debugging server) on each node. These monitor processes interact with the operating system to observe and manipulate the local target processes.

2.2.2 CoCheck: Consistent Checkpointing

There are several well established checkpointing mechanisms for non-communicating processes, e.g. the Condor checkpointing library [4]. In order to checkpoint a distributed application it is not sufficient to save the state of all processes, as the communication network contains state, too, i.e. the messages in transit. As this state is typically not directly accessible, CoCheck implements a protocol that flushes all messages in transit and stores them in the processes' address space [5]. Once the protocol has finished, all processes can be checkpointed using the Condor library.

In order to ensure that subsequent calls to PVM receive routines still find the right messages, CoCheck intercepts all calls to these routines and first checks whether there is a matching message buffered during the checkpointing protocol. In addition, CoCheck also intercepts most other PVM calls in order to translate the PVM task IDs passed to these calls. This is necessary, as the PVM system will assign new task IDs to the processes after a restart, but the processes still use the original ones.

2.2.3 VISTOP: State Based Program Flow Visualization

VISTOP visualizes a program's communication and synchronization behavior. It provides a sequence of snapshots of the program's global states, which the user can browse forward and backward to see the state changes.[3] The currently displayed state can be inspected in more detail, e.g. to see the contents of message queues.

As Fig. 2 shows, VISTOP works by monitoring calls to PVM routines in all target processes. A description of each event is entered to the event buffer local to the process's node. From time to time, or when one of the buffers is nearly filled, the local buffers are read and merged into a single event stream that is consistently ordered according to causality constraints. The stream is then forwarded to a component that computes and stores a sequence of states by modeling the behavior of PVM [6].

2.2.4 Codex: Controlled Deterministic Execution

In general, nondeterminism in parallel and distributed programs is created by unordered accesses to common objects. Codex removes this nondeterminism by enforcing the order of accesses to match user-specified access patterns. An advantage of this approach relative to the record-and-replay approach is that it not only enables deterministic re-execution, but also supports systematic testing [7].

For PVM, the relevant common objects are message queues that have a very special semantics. In this case, it is not necessary to control all the accesses (i.e. send and receive operations). Instead, the actual implementation of Codex for PVM applications just controls the receive operations. For this purpose, Codex intercepts all calls to PVM receive routines. When a target process wants to receive a message, Codex checks which message should be received according to the access pattern specified for the accessed message queue and modifies the call's parameters in such a way that this message will be received (see Fig. 3).

[3]Of course, a distributed program usually doesn't have a unique sequence of global states. VISTOP only displays one *possible* sequence that is consistent with the program's execution.

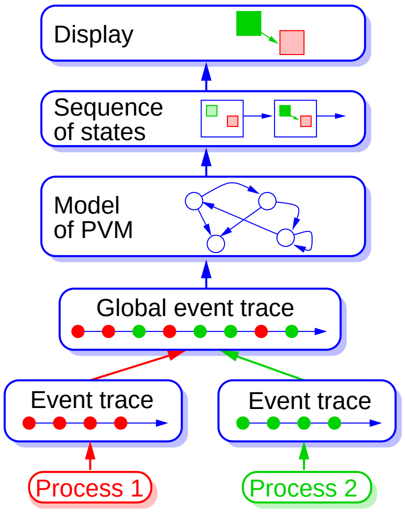

Figure 2: Data flow in VISTOP

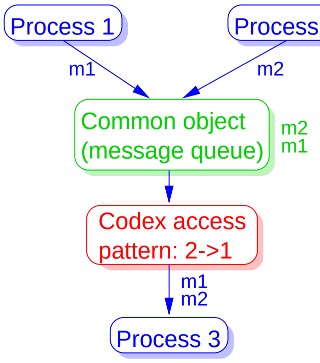

Figure 3: Codex: messages are received in the order m2, m1 although the order in the queue is m1, m2

3 Requirements for an Infrastructure Supporting Interoperable Tools

We will now use our tool scenario to derive a couple of essential requirements which should be met by a software infrastructure in order to support interoperable tools. We will do this by examining the mutual interactions of the tools presented above. Of course, we can only look at very few examples in this paper. However, the examples have been selected in a way that each of them is typical for a distinct class of interactions. Although there are many more interactions, they all fall info one of these classes.

Example 1: Monitoring of PVM Calls

The first example shows the most obvious, but also most important interaction: all the tools observe and maybe even manipulate the same target system. In particular, CoCheck, VISTOP, and Codex monitor calls to PVM library functions in each target process, whereby both CoCheck and Codex change the calls' parameters and/or results. From an abstract point of view, we have several software components (the tools) that access the same resources (the target system).

This very closely reflects the analogous situation in the context of operating systems, where several processes may access the hard disk, or in the context of data bases, where multiple clients access the stored data. In these contexts, it is general practice to have a common software component (i.e. the operating system or data base server) that coordinates these accesses based on some structure imposed on the resources (e.g. a file system for the hard disk). Unfortunately, in the context of run-time tools, such a component usually does not yet exist. This means that current tools access the target system in a completely uncoordinated manner, which easily leads to incorrect behavior or even crashes[4]

The first (and strongest) requirement for supporting interoperable tools therefore should be: Provide a common, open interface for on-line monitoring of distributed systems that is able to support multiple tools at the same time and offers mechanisms for coordinated access to the target system. As we will see in Section 4, this requirement is starting to be recognized, although there are only very few and limited implementations. For the following discussions, we assume that all the tools are based on such an interface, which is implemented by a distributed monitoring system.

Example 2: Consistent Presentation of State Information

Additional interactions between tools may become necessary due to user interface considerations. When a programmer uses multiple tools to observe the same program, he will easily be confused if these tools present different, seemingly contradicting information. Unfortunately, such a situation can occur with DETOP and VISTOP: By its nature, the debugger always presents information on the program's current state, while the visualizer may at the same time show information on a past state. A well engineered user interface should indicate that some information is currently invalid, e.g. by "graying out" parts of the display. If the user is working with VISTOP and scrolls back to a past state of the program, the information displayed by DETOP should be grayed out. When the user continues working with the debugger, the visualizer should automatically display the last (i.e. current) state again to make the displays consistent. Of course, this consistency should remain while the user is working with the debugger. Since the debugger only displays information after a process has stopped, it is necessary and sufficient to update the visualizer's display whenever a process stopped during debugging. This update requires to read the local event buffers and to process the merged event stream in order to compute the program's current global state.

Let us assume that the initiative for keeping the displays consistent should be with VISTOP[5]. When the user scrolls back to a previous state, VISTOP must request DETOP to gray out its displays. In addition, VISTOP must monitor the activation of the DETOP window and display the target program's last state once it receives a notification of this event. This leads to the requirement that the tool infrastructure should allow a tool to get notified about events in another tool, but also to request the execution of actions in another tool. We will denote this as *direct interactions*.

In addition, VISTOP must be notified when DETOP has stopped a target process in order to update its display. More abstract, a tool should be able to monitor modifications of a target object performed by some other tool. Since this kind of interaction is mediated by a target object, we will call it an *indirect interaction*.

[4] Of course, there are operating system services for process monitoring, e.g. `ptrace` or the `/proc` file system, but they do not offer any support for coordinating accesses and for distributed systems.

[5] It can also be with DETOP or with a third program that wraps around both tools. However, this does not lead to different requirements.

Example 3: Translation of Task Identifiers

While the previous example considered the issue of consistency between tools, this one treats a transparency issue. In Section 2.2.2, we have mentioned that when CoCheck restarts a PVM program from a checkpoint, all processes get new PVM task IDs. In order to make this change invisible to the program, CoCheck intercepts PVM calls and modifies their parameters and/or results. But of course, the change of task IDs should also be transparent to the other tools like DETOP. It is important to note that this is not automatically accomplished by CoCheck's instrumentation of PVM calls. First, CoCheck only controls the target processes, not the monitoring system or the tools. Second, the monitoring system need not be a PVM application, i.e. it may use other means than a PVM call to determine the PVM task ID of a target process.

However, if the tool environment fulfills our first requirement, every information about a monitored object is acquired via the common monitoring interface. So in principle, CoCheck could somehow catch the accesses to the PVM task ID performed by other tools and modify their results. Of course, this assumes that the monitoring system provides suitable mechanisms to do so. Thus, our fourth requirement for a monitoring infrastructure is that tools should be able to intercept accesses to specified state components of monitored objects requested by other tools.

Example 4: Modeling of Message Queues

Although the solution of a large number of interoperability problems can be supported by appropriate concepts in a monitoring middleware, there still remain situations that can not easily be dealt with: As we have seen in Section 2.2.3, VISTOP models the state of the PVM system based on the stream of events monitored in the target application. One component of this state is the contents of each process's queue of receivable messages. The contents of this queue can be computed by looking at the events that represent the arrival of a message (i.e. enqueue) or the receipt of a message (i.e. dequeue). This computation relies on the fact that the receive queues obey a FIFO semantics as defined by the PVM system. But remember that Codex can force messages to be received in an order different from the one in which they arrive, since the receive order is solely determined by the specified access patterns. Thus, VISTOP's modeling of message queues is no longer correct in the presence of Codex. In fact, VISTOP exits with a failed assertion.

The reason for this problem is that one tool violates a basic assumption of another tool. As a consequence, these tools cannot interoperate unless the latter one is adapted to no longer rely on the assumption. These situations usually are the most serious obstacles to interoperability. They can only be avoided if tool developers carefully think about whether assumptions about the target system still hold in the presence of other tools.

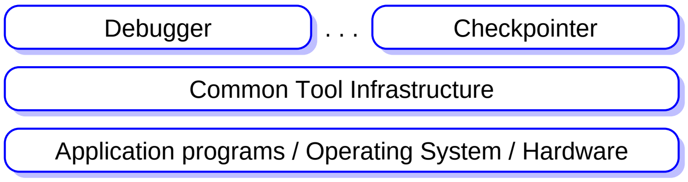

Figure 4: Infrastructure of a multi-tool environment

4 Supporting Infrastructures

The preceding chapter showed that the concept of interoperable tools is accompanied by a list of requirements that have to be met. These requirements refer to the common infrastructure the individual tools are based on. Fig. 4 depicts the software layers that can be distinguished in such an environment. The layer to be observed and manipulated is the complex of running application programs, the operating system and the hardware. Tools are connected to this layer by means of a special tool infrastructure.

In the process of tool construction the question arises how the middle layer should be designed. From the viewpoint of a tool developer that just uses this layer, it would obviously be advantageous to have it as thick as possible. This implies that it covers a large functionality, making it relatively easy to put new tools on top of it. If the infrastructure layer is thin, more functionality has to be provided in the tools themselves, leading to a possible replication of identical functionality. We propose to have an infrastructure layer that is extensible and is just thick enough to avoid such replication. It should comprise essential common tool functionality as well as concepts for tool interoperability. A couple of approaches are already available to be integrated into this layer.

4.1 Monitoring Systems

The monitoring system is the part of the tool infrastructure responsible for observing and manipulating the target system. Today, there are almost no stand-alone monitoring systems available. Usually, monitoring facilities are integrated into the respective tools following a monolithic construction process. In the case of multi-tool environments it is crucial to have a versatile monitoring system that offers a comprehensive set of services.

One such approach is the DAMS environment (Distributed Applications Monitoring System) [8]. DAMS is already 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 indirect tool interactions, it does not integrate interoperability concepts that satisfy our additional requirements.

The DPCL approach (Dynamic Probe Class Library) [9] 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, as the merging 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 not explicitly support tool interoperability. DPCL concentrates on performance analysis but its functionality also allows any type of manipulation services.

A third approach is the monitoring system OCM, which is discussed in Section 5.

4.2 Infrastructures for Tool Interoperability

We also find concepts and approaches to support interoperability which, however, are not dedicated to parallel run time tool environments. An interesting question is whether these concepts can be joined with monitoring approaches to meet the requirements presented in Section 3.

An established concept for interoperability is ToolTalk [10]. It was conceived by SunSoft and aims at providing a messaging mechanism for multiple clients in a distributed environment. ToolTalk is applied in the Common Desktop Environment to take care of inter-window cooperation. This role would also be appropriate for a multi-tool environment. ToolTalk's working paradigm is to send requests via messages to others that might provide services to handle these requests. In case of no appropriate client listening, the message yields no effect. The ToolTalk semantics is well adapted to environments with a varying number of partners potentially being unknown to each other at startup.

Another approach exists in the realm of software engineering. PCTE (Portable Common Tool Environment) [11] aims at integrating tools of multiple vendors. The concepts in PCTE are based on an object management system, where the individual objects are specifications, software modules etc. Thus, it does not fit into our field of application, which is event oriented. Nevertheless, PCTE exhibits clever concepts for interoperability in general which could be transferred to other areas of interest.

There are also other concepts for multi-client interoperability, e.g. SNMP and Corba. Their level of abstraction is very high and a link with low-level monitoring concepts will necessarily lead to a loss of efficiency in the implementation. Furthermore, it is not clear how they can meet the stated requirements.

5 Interoperability Support in the OCM

The OCM [12] is an implementation of the monitoring interface OMIS (Online Monitoring Interface Specification) [13, 14] that has been developed at LRR-TUM. Its primary goal is to provide a universal monitoring infrastructure for run-time tools. Since it is not possible to know in advance the monitoring capabilities needed by future tools, a basic design concept of the OCM is extensibility, i.e. it is possible for a tool developer to add new modules to the monitoring system that provide tool specific monitoring services. However, in this paper we will not plunge into this topic, but concentrate on the interoperability support.

For a better understanding, we will first provide some details on the concepts of OMIS. For a full description, see [14]. OMIS is based on an object model of the target system: the target system is viewed as a hierarchical collection of objects, where the object types defined in the base interface specification are nodes, processes, threads, message queues, and messages. For these objects, OMIS defines services that fall into one of three classes:

1. *information services* that just return information on a set of objects,
2. *manipulation services* that change the state of objects,
3. *event services* that notify certain state transitions in objects.

The first two types of services are also called *actions*. An event service can be combined with a list of actions to form a conditional service request, where the actions are executed whenever the event is detected. A parameter passing mechanism allows to propagate information on the event occurrence to the actions. In this way, the monitoring system can be programmed by a tool and can autonomously execute the action list without any further tool interaction. The following examples show two typical requests used by DETOP and VISTOP:

```
thread_creates_proc([n_1,n_2]):
       thread_stop([$new_proc]) proc_get_info([$new_proc], 12)

thread_creates_proc([n_1,n_2]):
       trace_buffer_append(trace_b_1, $time, [5, $proc, $new_proc])
```

The first request instructs the monitoring system to stop any process created by a monitored thread on nodes 1 or 2 and to return some information on that process to the debugger. The second request uses the same event, but specifies a different action: append event information to a trace buffer.

We are now ready to explore the interoperability support in the OCM in detail.

5.1 Multi-Client Interface

The OCM already supports multiple concurrent clients, since this has been a major goal when designing the monitoring interface. Therefore, the OCM is implemented using a distributed server architecture, where a central component distributes requests to the local monitors on each node. The OCM maintains a private environment for each tool that contains e.g. the set of objects to be monitored. Thus, tools are not forced to monitor exactly the same target system. However, since the tools usually monitor some common objects, the OCM includes basic coordination features:

- Services are mutually exclusive on single objects, i.e. no other service can access the same object concurrently.
- Requests relating to more than one node are distributed to the local monitors using an atomic multicast protocol to ensure that requests are executed in the same order on each node.

- Finally, it is possible to lock a request, such that no other service is executing on any node while the locked request is processed.

Note that there are no explicit locks for individual objects in order to avoid the danger of deadlocks.

These coordination features turned out to be sufficient in practice, with one exception: As action lists in conditional requests may modify the state of the target system, the order in which action lists bound to the same event are executed can be important. The problem is how a tool issuing a conditional request can specify this order, since the tool usually does not know whether other tools monitor the same event, too. A promising approach is to automatically determine the order in the OCM, based on given properties of the action lists. Such a scheme already works well for determining the order in which event filters are called inside the OCM, i.e. functions determining whether an event service should trigger its actions. However, whether this scheme can also be applied to the more complex case of action lists is still an open research issue.

5.2 Events and Actions in Other Tools

As outlined in [12], event detection and the execution of actions is not solely carried out by the OCM's local monitor processes, but can also be performed in the context of the monitored target processes themselves. This feature has been included primarily for efficiency reasons. It enables to execute a class of requests like "increment a counter whenever process x calls `pvm_send`" without the need for context switches on each event occurrence.

We are currently using two alternative implementation schemes: The monitored program can be relinked with special versions of programming libraries (e.g. the PVM library) where the interface functions include code for event detection, and additional monitor libraries that contain parts of the monitoring system, especially the available actions. The alternative approach uses the `dyninstAPI` [15] to dynamically insert the event detection code into a running process and to dynamically link it with the monitor libraries on demand. To ensure that these mechanisms are transparent at the monitoring interface, the OCM also handles cases where the detection of an event and the execution of its associated actions must happen in different processes. This is done by inserting special actions that transfer control to the other process, which may even be located on a different node.

Given these mechanisms, it is obvious that they also can be used to implement the detection of events and the execution of actions in another tool, provided that this tool becomes a part of the monitored system. Thus, this step basically removes the distinction between observing tools and the observed system. An advantage of this implementation of direct tool interactions, as compared to using systems like e.g. ToolTalk, is that the adaptions required in the tools' source code are minimal. Events corresponding to the entry to and the exit from functions in a tool can be detected without a need for source code changes. The same holds for actions if there is already a function in the tool that performs the action. We expect that these conditions will usually already be satisfied in well structured tools. The topic will be further examined in a collaboration with Univ. of Madison, Wisconsin[6].

5.3 Monitoring of Object Accesses

In Section 3, we identified the requirements that a monitoring infrastructure should be able to notify changes in object state and should offer means to intercept accesses to object components. Given a system like the OCM that already includes mechanisms to notify events and to execute actions upon event occurrences, the problem that remains is how to detect accesses to object components. Since the monitoring system must be extensible and its code should not be cluttered with access checks, we are looking for a *generic* mechanism that allows to specify and monitor the events of interest. The issues of such a mechanism are:

1. How to name and identify object components?
2. How to make sure that we do not miss any accesses?

The solution we propose is to refine the object model of OMIS in the following way:

1. Objects consist of a set of components. We implicitly used this in our previous discussions, but now components become an explicit part of the model.
2. Any piece of information that can be accessed by the monitoring system is modeled as a component.
3. Each component of an object is identified by a unique selector.
4. Components do not overlap, i.e. changing a component does not influence the other ones. This requires that the model must be rather fine grained.
5. Access to components is only possible via two generic methods `read` and `write`.

Under these prerequisites, it becomes possible to specify accesses of interest as a pattern of the form `(access_mode, object, selector, value)`, where `access_mode` is `read`, `write`, or `any`. The `read` and `write` methods then can match the actual access against all defined patterns and notify or intercept the access in case of a match.

For practical reasons there are a few more restrictions on this model. First, the selectors should be static, i.e. they should not change during run-time. Furthermore, in order to simplify the use of the model within the monitoring system, the selectors should give an indication of the component's semantics, so we can not just use an index. Ideally, they should enable a hierarchical naming such that e.g. memory cell 1234 of a process gets a selector like `memory.1234`. In order not to miss any accesses, each access to the target system's state must happen via the `read` and `write` methods. This means that e.g. to stop a process `p_1`, the monitoring system has to set the process's `stopped` component by calling `write(p_1, stopped, True)`, which in turn calls some operating system service to actually stop the process.

A major implementation problem comes from the extensibility requirement of the OCM: it is possible to extend the OCM by just linking new extension libraries to it, without having to recompile. Together with dynamic linking techniques, this

[6] Funded by the German Science Foundation (DFG), Grant WI1346/1-1.

even allows a dynamic extension on demand. But since an extension may enable the monitoring of additional aspects of an object's behavior, it must be able to introduce new components for existing objects. This means that our object model must be dynamic and cannot simply be mapped to C++ or Java objects, where the structure is fixed at compile time. Investigating efficient implementation techniques for this dynamic object model is part of our ongoing research.

6 Conclusions and Future Work

Interoperability of run-time tools is a promising concept that could help both tool developers and tool users in producing and using sophisticated and extensible tool environments. In this paper, we investigated the support that must be offered by an infrastructure for run-time tools in order to allow the construction of such interoperable tool environments. We derived the requirements by examining some interactions that occur in a realistic environment that aims at efficient debugging of long running programs. To put these requirements into practice is not a trivial task, a fact that explains that currently no such infrastructures exist. Based on the concepts of the on-line monitoring interface specification OMIS and its implementation OCM, we outlined techniques to meet these requirements in a monitoring system. Our future work will concentrate on the construction of a more formal model to describe interoperability concerns and on an integration of the underlying concepts into OMIS/OCM. Finally, we will implement a working prototype of the sketched debugging environment.

References

[1] The Parallel Tools Consortium. PTools MQM working group home page. `http://www.ptools.org/projects/mqm/`.

[2] Cray Research, Inc. *UNICOS CDBX Debugger User's Guide*, 1996. Cray Publication SG-2094 8.0.

[3] 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`.

[4] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. Checkpoint and migration of UNIX processes in the Condor distributed environment. Techn. Report 1346, Univ. of Wisconsin-Madison, Comp. Sci. Dept., Apr. 1997. `http://www.cs.wisc.edu/condor/doc/ckpt97.ps`.

[5] G. Stellner and J. Pruyne. Resource management and checkpointing for PVM. In J. Dongarra et al., editor, *EuroPVM'95*, volume 5 of *Parallèlisme, réseaux et répartition*, pages 131–136, Lyon, France, Sep. 1995. Hermès.

[6] 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`.

[7] 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. IEEE Computer Society. `http://wwwbode.in.tum.de/~oberhube/PUBS/PS/pdse97.ps.gz`.

[8] J.C. 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, Sep. 1998. Springer Verlag.

[9] 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`.

[10] A.M. Julienne and B. Holtz. *ToolTalk & Open Protocols — Inter-Application Communication*. SunSoft Press, Englewood Cliffs, NJ, 1994.

[11] F. Long and E. Morris. An overview of PCTE: a basis for a portable common tool environment. Technical Report CMU/SEI-93-TR-1, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Mar. 1993.

[12] R. Wismüller, J. Trinitis, and T. Ludwig. OCM — a monitoring system for interoperable tools. In *Proc. 2nd SIGMETRICS Symp. on Parallel and Distributed Tools SPDT'98*, pages 1–9, Welches, OR, USA, Aug. 1998. ACM Press. `http://wwwbode.in.tum.de/~wismuell/pub/spdt98.ps.gz`.

[13] T. Ludwig and R. Wismüller. OMIS 2.0 – a universal interface for monitoring systems. In M. Bubak, J. Dongarra, and J. Wasniewski, editors, *Recent Advances in Parallel Virtual Machine and Messag Passing Interface, Proc. 4th European PVM/MPI Users' Group Meeting*, volume 1332 of *Lecture Notes in Computer Science*, pages 267–276, Crakow, Poland, Nov. 1997. Springer Verlag. `http://wwwbode.in.tum.de/~wismuell/pub/europvm97b.ps.gz`.

[14] T. Ludwig, R. Wismüller, V. Sunderam, and A. Bode. *OMIS — On-line Monitoring Interface Specification (Version 2.0)*, volume 9 of *LRR-TUM Research Report Series*. Shaker Verlag, Aachen, Germany, 1997. ISBN 3-8265-3035-7. `http://wwwbode.in.tum.de/~omis/OMIS/Version-2.0/version-2.0.ps.gz`.

[15] J. K. Hollingsworth and B. Buck. *DynInstAPI Programmer's Guide Release 1.2*. Comp. Sci. Dept., Univ. of Maryland, College Park, MD 20742, Sep. 1998. `http://www.cs.umd.edu/projects/dyninstAPI/rel1.2/dyninstProgGuide.pdf`.