Synergetic Tool Environments*

Thomas Ludwig, Jörg Trinitis, Roland Wismüller

Technische Universität München (TUM), Informatik
Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Arcisstr. 21, D-80333 München
email: {ludwig|trinitis|wismuell}@in.tum.de

1 Introduction

In the area of programming parallel and distributed applications we find tools of various types to help us to develop, optimize, and maintain parallel code [6]. Tools can be categorized as on-line and off-line and as interactive and automatic. Off-line tools are trace based and allow a post-mortem program analysis for e.g. performance bottleneck detection or debugging. By their technology they are not able to manipulate the program under investigation. This can only be performed by on-line tools. They provide instant access to the program, thus supporting a wider variety of tools. Interactive tools can be used for e.g. debugging [1], performance analysis [3], program flow visualization, or computational steering, whereas automatic tools offer services for e.g. dynamic load balancing and resource management.

Although we nowadays find many tools from various developers from industry and academia there is a decisive problem when applying them: on-line tools can almost never be used in combination. Due to their complex software infrastructure needed for program observation and manipulation (usually called monitoring system) they require the parallel program to be linked with special libraries and run under strictly specified execution conditions such as especially adapted runtime environments. Unless different tools are implemented by the same producer they are based on incompatible infrastructure concepts which do not allow a concurrent tool usage (see Figure 1).

Thus, synergetic effects of applying more than one tool to the program are impossible. It is the goal of our project to change this situation and to develop concepts and means for the implementation of environments of interoperable tools.

2 The Concepts of Interoperable Tools

Interoperable tools is a concept that is hardly covered in the literature on development and maintenance tools for parallel programs. Let us first have a closer look on the basic ideas behind this concept.

* This work is partly funded by *Deutsche Forschungsgemeinschaft*, Special Research Grant SFB 342, Subproject A1.

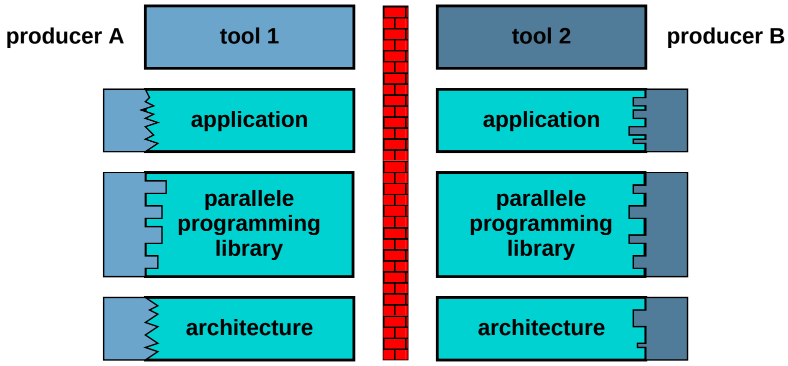

Fig. 1. Incompatible on-line tools

The concept of interoperable tools is to provide a synergetic effect to the developer by allowing to apply more than one tool at a time to a program run. The synergy results from complementary functions in the individual tools which add new features to each other.

Consider the usage of a debugger. You will monitor programs to find errors in the code. With parallel programs that exhibit difficult timing behavior and are used for long running applications like e.g. fluid dynamics simulations this might result in long debugging sessions where the programmer waits for errors to occur. If we add a second tool that can can perform regular and on-demand checkpoints of a set of processes we will end up with a more powerful development environment: If an error occurs we will set the program back to one of its recent checkpoints and start with exploring the causes of the error. By this the debugger will be much more efficient. We will even be able to checkpoint the debugger itself and thus can restart not only the parallel program but a complete debugging session.

Interoperable tools are thus a combination of two or more on-line tools. They may belong to the class of interactive and automatic tools. Their concurrent usage will yield some synergetic effect which is more than just the sum of the functionality of the individual tools.

We have to distinguish interoperable tools from integrated tools joined in a certain tool environment. Integrated tools are usually a collection of interactive on-line tools that can not necessarily be applied concurrently. Furthermore, although it is a set of tools, it is not open to the integration of further tools. In fact, its development concepts are usually identical to those of a single tool with the difference that a larger set of functionality is supported.

What are the inherent problems of designing interoperable tools? In fact there are currently no tools that meet the requirements of our definition. The first problem is the complexity of the software infrastructure required by on-line tools. A powerful monitoring system is necessary to support program observation and manipulation. Different such systems were developed for different tool types but none of them offers enough generality to support also other tools. As monitoring systems are usually tightly joined to the parallel program and its runtime environment, only one such system can be active at a time. Second, there are hardly any powerful standards available, especially for interfaces between tools and monitoring systems. Having such standards could greatly simplify the construction of interoperable tools and would allow to serve several tools with a single monitoring system (see Figure 2).

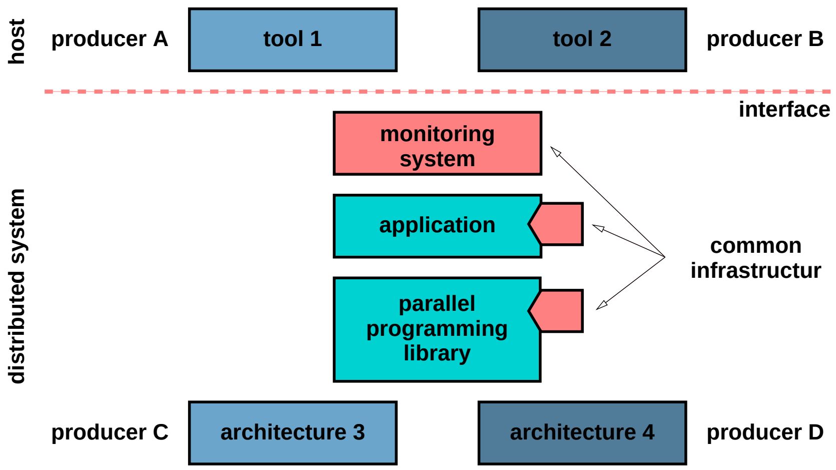

Fig. 2. Interoperable on-line tools

Finally, as there are no concepts for a general purpose tool infrastructure there are consequently also no concepts for tool interoperability. It has not yet been worked out how tools should be designed in order to be interoperable with other, possibly yet unknown tools.

3 Levels of Interoperability

With the concept of interoperability being applied to tools we first have to consider the levels of interoperability that are to be distinguished.

- The first level is characterized as co-existence. Several tools can be applied concurrently to the same program run. However, they have no knowledge of the existence of the other tools. The benefit of having more than one tool is reduced by the danger of a potentially conflicting behavior of the tools. If they manipulate the same objects of the program they might generate an inconsistent program state. Consider a debugger that initiates a single step

mode on a process in combination with a load balancer that migrates exactly this process to another node. A program abortion might be the consequence of these tool activities.
- At the second level we have a concept called consistent co-existence. Tools are obliged to preserve consistency of the program system under investigation. In detail this implies that manipulation of objects are treated as critical regions. It has to be guaranteed that only one tool is in the critical region with respect to specific objects at a time. As with consistency mechanisms in other application fields we can determine several concepts to meet this requirement. One possibility would be that the monitoring system itself coordinates the access to objects by the various tools. Another way can be provided with means for observing tool activities and controlling access to individual objects. In any way there is no direct tool cooperation but a coordinated co-existence.
- The third level is thus characterized by direct cooperation. By means of a communication mechanism tools exchange control information and data. A debugger could inform others of its single-step activity, a load balancer could send its current load evaluation heuristics to some interactive tool to be presented to the user. Direct cooperation is the most advanced concept of interoperability. However, it might already limit the synergetic effects by pre-defining them with the messages to be sent.

The three levels of interoperability require different concepts for their realization. Concepts get more complex with each level. For the first level to be reached it is sufficient to be able to run the tools concurrently. This can easily be achieved by basing them on an identical software infrastructure which is powerful enough to support the sum of the functionality required by the tools. Note that there are hardly any realizations available that provide such a functionality. With the second level we must require concepts for coordination in addition to all concepts for the first level. Coordination could be performed automatically by the infrastructure itself. However, as the monitoring system knows nothing of the tools' semantics it might follow a worst case approach where everything is coordinated by locks etc. even when this would not be necessary. A more flexible approach is to provide the tools with mechanisms to coordinate their activities. This comprises detection of object manipulation by other tools and locking of objects against manipulation from other tools. It is essential to distinguish between object observation and object manipulation. In most cases only the latter has to be coordinated as it bears the chance for inconsistencies in the tools and the programs. Finally, with the third level we also require concepts for tool cooperation. This can be achieved by a message passing mechanism by which tools can control cooperation. Two issues are to be considered. Message passing must follow a protocol that does not depend on the actual receivers to be present. Usually we do not know in advance how many tools will be used concurrently in a session. Messages must correspond to a fixed though expandable format. Different types of message formats must be offered for vice versa tool control and information messages. Problems of this kind are already handled in the ToolTalk approach from Sun. However, there is no methodology how to integrate this into the individual tools.

We conclude that tool interoperability is a goal that is characterized by many different aspects. Various degrees can be achieved with different effort providing the tool user with more or less powerful tool environments and more or less possibilities for synergetic effects.

4 The Infrastructure Concept: OMIS

The last section showed that for interoperable tools to be designed and implemented we can identify one crucial prerequisite: We need a common software infrastructure for all tools that is powerful enough to support their individual functionalities and that allows for the various levels of interoperability. Such an infrastructure concept was conceived in the OMIS project in our research group [2].

The on-line monitoring interface specification (OMIS) was developed in 1995 and first published in January 1996. Its goal is to define an interface between tools and on-line monitoring systems that allows to base various types of tools on top of it. It covers on-line and off-line tools as well as interactive and automatic tools. The interface is currently oriented towards tools for parallel and distributed programming based on the message passing paradigm. It offers a single function by which tools can send requests to the monitoring system to invoke certain activities. Requests are structured as event/action-relations where each such relation tells the monitoring system to watch for the occurrence of the specified event and to invoke the specified actions when it occurs. By sending sets of requests, a tool is able to program the monitoring system to perform a certain functional behavior. Events and actions are composed by the name of a service and a list of parameters. The latter usually identify objects of interest of our parallel program. Event services are e.g. “a process terminates”, “a node is added”, “a message arrives”. Action services fall into two categories, one for observation, one for manipulation. We offer e.g. “get node status information”, “show message contents”, “perform single step on process”, and “modify message contents”. Consequently, the set of object types that can be dealt with comprises nodes, processes, threads, message queues and messages. Additional object types stem from the monitoring system itself: service requests, timers, and counters.

The interface specification offers a set of basic services that covers all typical types and the most common activities performed by current tools. For future adaptations to new tools and new programming paradigms with new objects it employs a mechanism for extensions to be brought in.

Based on this specification we implemented an OMIS compliant monitoring system (OCM) for the PVM programming library on workstations clusters as target architecture [8]. In a first step we put our already existing tools on top of this software infrastructure [7]. This is already level one of interoperability as the tools can execute concurrently, but do not perform coordination or cooperation.

How does the interface specification support the higher levels of interoperability? For a coordinated co-existence the interface offers a comprehensive set of event services. Using them, it is possible for a tool to observe object manipulations invoked by other tools. It may then react appropriately. In order to execute actions without being interrupted, a service for locking is offered. By that a tool is guaranteed to have an exclusive object access.

Cooperation in form of direct tool-to-tool communication is not yet supported by OMIS. Although it would be a minor effort to integrate the messaging mechanism, it is a very complex task to specify the cooperation protocol and the message format. In addition, no profound knowledge is available what features are really useful when having cooperating tools. An investigation of this issue will be preceded by implementing interoperable tools at level two.

Our current research activity concentrates on the definition of co-existence for various tools. Starting with a comprehensive list of on-line tools we investigated, which tool combinations do provide synergetic effects. Among them we identified two combinations that seem to be most promising: the synergy of debugging and checkpointing and the one of performance analysis and load balancing. The resultsof the first will be presented here; the latter will be described in the full paper.

5 Interoperability of Debugging and Checkpointing

The first set of interoperable tools implemented at our chair were a parallel debugger (DETOP) [4] and a checkpointing tool that is based on CoCheck [5]. The initial goal was to achieve consistent co-existence of the two tools.

Our OMIS compliant monitoring system (OCM) is powerful enough to support both kinds of tools. The debugger utilizes services to read and write memory, stack, etc. and to control the program's execution (stop, continue, single-step). The checkpointing tool makes use of OMIS' checkpoint/restore services. These are implemented as a tool extension to the monitor and ensure atomicity of the operations with respect to other services by locking. Thus, OMIS/OCM offers the possibility to achieve *co-existence* of the two tools.

This co-existence, however, is up to this point only a *non-aware* co-existence, where the two tools don't know anything about each other. As both kinds of tools manipulate common objects (e. g. the processes of the parallel application), this can lead to trouble whenever assumptions are made about the state of such objects.

To achieve *consistent co-existence*, additional measures have to be taken. Either the monitoring system has to hide manipulations from the other tools, or the tools have to react appropriately in the event of such manipulations. Because the first approach is far too complicated and expensive in the general case (e. g. saving "before images" in case a process is killed by one tool), we decided to follow the second approach.

As a first step, the tools have to be *aware* of each others critical actions. When such actions take place, the tools have to react on them to achieve consistent co-existence. This of course requires some form of communication.

Our approach was to have the monitoring system support this through a special form of indirect communication, which fits perfectly well into the event/action scheme applied by OMIS. Whenever a checkpoint is to be written, a *will_be_checkpointed* and a *has_been_checkpointed* event are triggered at the appropriate points of time and for each process in question. The debugger configures the monitoring system to have special actions executed before and after checkpoints are written[1]. Through this, modifications done to the processes can be saved and hidden from the checkpointing system. On the other hand, whenever a restore takes place, the debugger is noticed and will update its state (variable views, etc.). Finally, of course, the debugger can initiate checkpoint and restore actions for the processes being debugged, thus potentially dramatically shortening the testing and debugging cycle and reducing software development costs.

6 Project Status and Future Work

The status of the interface specification is currently fixed. Version 2 was published in June 1997. Based on this document we developed an OMIS compliant monitoring system (OCM) for PVM. First results are available with two debugging tools being based on OMIS.

The next step is to combine already available tools to interoperable tools. Within the framework of two research grants we will not only finish the combination of the debugger and the checkpointing facility, but also look into combining a performance analysis tool and a dynamic load balancer. The design phases of these projects are finished (see Figure 3).

OMIS itself will be adapted to the shared memory programming model. This research is also embedded in a national research program.

References

1. R. Hood. The *p2d2* project: Building a portable distributed debugger. In *Proc. SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 127–136, Philadelphia, Pennsylvania, USA, May 1996. ACM Press.
2. 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.
3. B. P. Miller, J. M. Cargille, R. B. Irvin, K. Kunchithap, M. D. Callaghan, J. K. Hollingsworth, K. L. Karavanic, and T. Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 11(28), Nov. 1995.
4. 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.

[1] Of course the same applies to restores

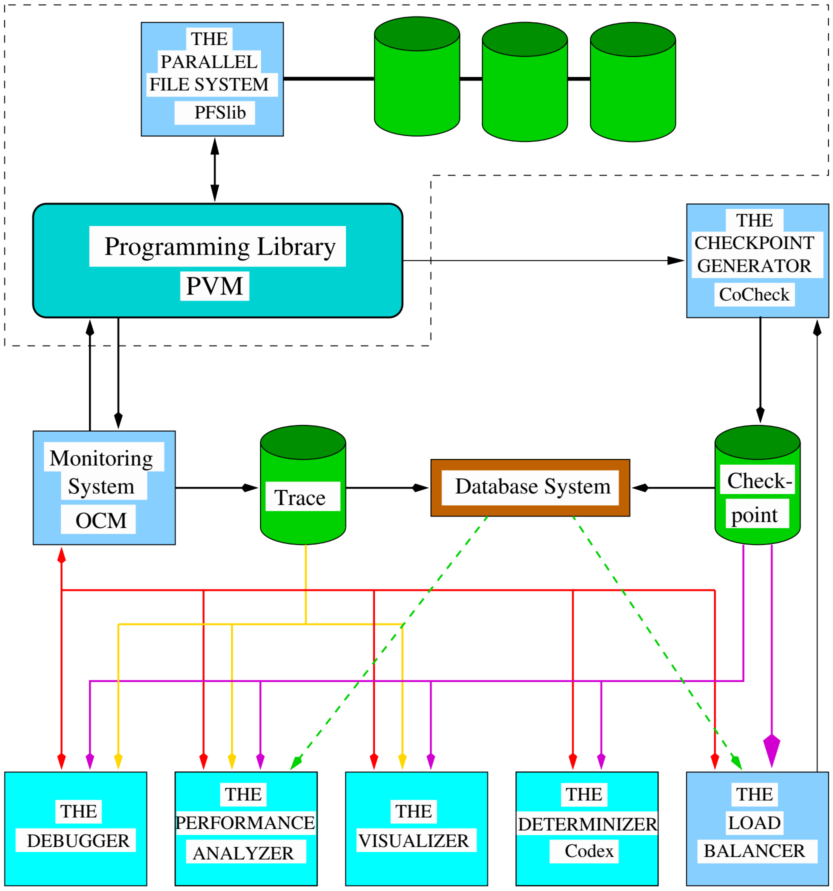

Fig. 3. The Tool-set environment

5. G. Stellner and J. Pruyne. *CoCheck Users' Guide V1.0 – PVM Version*. Technische Universität München, Institut für Informatik, Lehrstuhl für Rechnertechnik und Rechnerorganisation, Nov. 1995.
6. T. Sterling, P. Messina, and J. Pool. Findings of the second pasadena workshop on system software and tools for high performance computing environments. Technical Report 95-162, Center of Excellence in Space Data and Information Sciences, NASA Goddard Space Flight Center, Greenbelt, Maryland, 1995.
7. 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 *Proc. 2nd Sino-German Workshop on Advanced Parallel Processing Technologies, APPT'97*, Koblenz, Germany, Sept. 1997.
8. R. Wismüller, J. Trinitis, and T. Ludwig. OCM — a monitoring system for interoperable tools. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 1–9. ACM Press, August 1998.