On-line Distributed Debugging on Scalable Multiprocessor Architectures *

Thomas Bemmerl[1] and Roland Wismüller[2]

[1] RWTH Aachen, Lehrstuhl für Betriebssysteme (LfBS), D-52056 Aachen, FRG
[2] TU München, Lehrstuhl für Rechnertechnik und Rechnerorganisation, D-80290 München, FRG

Abstract. Debugging parallel programs is one of the most tedious jobs in programming scalable multiprocessor architectures. Due to the distributed resources of these machines, programming is often architecture dependent. Most development tools still reflect this dependency even during the analysis phase of parallel programs. This paper presents the distributed debugger DETOP, which offers a global name space and hides architectural features like the mapping of processes. DETOP is part of the integrated tool environment TOPSYS implemented on iPSC hypercubes, networks of SPARCstations and partly on Transputer and PowerPC based systems.

1 Introduction and Problems

Many compute intensive applications require computing performance far beyond the performance of todays fastest supercomputers. These fields of applications include, but are not limited to fluid dynamics in automobile industry and weather forecast, image processing, all types of simulations, databases in banking and artificial intelligence. Many computer scientists claim that the performance increases necessary for these "grand challenges" can only be reached with parallel computers based on the distributed memory architecture.

The distributed memory multiprocessors (DMMP's) offer a higher degree of potential parallelism because of their scalability. Therefore these architectures are favored for performance increases by numbers of magnitudes to solve the grand challenges of the future [15, 12]. Computer science research in the last ten years has lead to scalable distributed memory computers, which are already partly commercially available. Research projects all over the world are on the way to build new parallel machines with significant increases in performance and decreases in price performance ratio.

Although it has been shown, that a performance increase by numbers of magnitudes is feasible with parallel machines, they are currently not widely used in the production process. This is due to the necessity of explicit parallelization and the architecture dependencies of efficient parallel programs, which makes programming DMMP's more complicated compared to sequential systems. Although some of the DMMP's already offer hardware concepts to hide the specific

* Partly funded by the German Science Foundation, Contract: SFB 0342

architectural features (e.g. topology is hidden by cut-through routing and virtual channels), programming these machines is still more time consuming than writing programs for workstations [8]. A lack in adequate programming tools for parallel architectures boosts this effect. In particular the design and implementation of adequate on-line debugging tools is a very challenging task because of the following problems:

- The distribution of resources (e.g. processors, memory, I/O) introduces many problems with respect to time relationships between resources and consistency of states.
- The complexity of the target system with respect to the numbers of elements (e.g. processes, processors, channels) the programmer has to deal with.
- Another source of complexity is the dynamic behaviour of parallel systems, in particular on target systems with highly dynamic process models and applications.
- A well known problem of parallel architectures is the natural non-determinism of parallel programs, which prevents programmers from a simple way of replaying execution runs.
- When monitoring the execution of a parallel program, it is well known, that parallel debuggers interfere with the application program and may change its dynamic behaviour. This probe effect is a critical problem to overcome in the implementation of a debugger for DMMP's.
- Finally, advanced node processor architectures, usually superscalar, superpipelined RISC processors or VLIW processors in combination with sophisticated highly optimizing compilers are contradictory to a simple implementation of debugging tools.

2 State of the Art and Related Work

To deal with the difficulties that arise when debugging parallel programs, different techniques have been developed during the last few years. These research activities can be divided into four main categories:

1. Off-line debugging
2. On-line debugging
3. Deterministic reexecution
4. Static analysis

2.1 Off-line Debugging

An off-line debugging session consists of two different phases. During the execution phase, information is gathered about the behaviour of the program, usually in the form of a trace-file for each node or process. The events that are traced are often fixed at compile time. In the second phase, the traces can be analyzed with different tools to detect unnormal behaviour. These tools offer either some kind of graphical representation of the trace or database like information retrieval languages.

This debugging strategy offers the advantage that the program doesn't need to be deterministic, because all relevant information is acquired within a single execution. The main disadvantage is the enormous amount of memory (and time) needed for storing detailed traces. Therefore some work has been done to reduce the amount of traced data by recording only certain infrequent events, like communication events or procedure calls, and regenerating the full trace by interpretation or reexecution of small parts of the program ("incremental tracing" [13], "abstract execution" [10]).

2.2 On-line Debugging

On-line (interactive) debugging allows the user to interact with the debugger during runtime of the debugged program. The key feature of an on-line debugger is the ability to stop the program at any points of the user's interest (breakpoint). Once the program is halted, the program state can be examined or modified. Finally the program is continued until another breakpoint occurs. When an unnormal behaviour has been detected, one usually needs information about previous states of the program to detect the reason for this behaviour. This information can be taken from a trace - in a way similar to off-line debugging - or by rerunning the program. The latter technique is known as "cyclic debugging" and usually requires techniques to ensure deterministic reexecution.
To specify the point where to stop the program, two different approaches have been developed.

- In the behavioural approach the user specifies the desired behaviour in terms of relations between program events (e.g. sending a message). Any deviation of this specification is regarded as an error condition and results in stopping the program. This means that the description of program behaviour must be complete in terms of the selected events. Therefore the behavioural approach is more suited for testing than for debugging. On the other hand it has the advantage of partially avoiding reexecution of the program [1].
- The event based approach also allows the user to define events and relationships between events. However, these relationships do not describe correct program behaviour, but a situation when the program has to be stopped, i.e. the program is immediately halted, when the sequence of program events satisfy the given relationships [14, 9].

Both approaches require techniques to halt all processes of a distributed system in a consistent state, i.e. a state that could have occurred during normal execution. The easiest method is to stop only the process that has triggered the halting algorithm. All other processes continue execution until they block due to synchronisation. Although the processes are always in a consistent state, they may have executed many statements since the halting algorithm has been initiated. So the inspection of this state may not give enough useful information to detect the cause of a bug.

A simple improvement is to stop all processes as soon as possible. This can be achieved, if the process that triggers the halting algorithm sends a broadcast message to all other processes. These processes stop when they receive the message. Since the processes are halted earlier, their states will be more meaningful than in case of the previous solution.

If the user wants to get information about the cause of a breakpoint, the best method is probably to stop the distributed system in the earliest possible consistent state ("causal distributed breakpoints" [7]). However, this can only be achieved by resetting processes to some previous state, which requires expensive logging and replay techniques. In fact, causal distributed breakpoints can be viewed as a combination of the previous halting method and reverse execution.

2.3 Deterministic Reexecution

Parallel programs are usually nondeterministic because of the different relative speed of processes and non predictable message delays. Therefore, the order, in which messages arrive at a process, may not be the same during reexecution of a program, resulting in different program behaviour.

A couple of techniques address the problem of reproducible reexecution of parallel programs. They are especially useful for debuggers based on the cyclic debugging paradigm.

"Data driven replay" allows reexecution of an isolated process in an artificial environment provided by the debugging system. During reexecution, all messages sent to this process are taken from a log (where they must have been stored during the original execution) and are supplied to the process in the correct order. Replay is useful, if only a single process is debugged. However, logging all messages sent to that process may be prohibitively expensive.

"Instant replay" [11] only requires logging the relative order of messages rather than the messages' contents. Due to this restriction it is necessary to rerun all processes of the program. During replay, receive operations are delayed, so that each process receives its messages exactly in the same order as in the original execution. Instant replay can be implemented either as part of the user program [6] or in the operating system.

Both replay and instant replay allow only reexecution from the start of the process or the program, respectively. "Reverse execution", in contrast, allows the state of a process to be reset to any previous state. In its simplest form, this is achieved by introducing logging points in each process, where each component of the program state (e.g. program counter, variables, message queues ...) that have changed since the last logging point, is stored to disk. Then any previous program state can be reached by a combination of restoring the state from disk and reexecution. Naturally, the cost for generating the log is even higher as with replay.

2.4 Static Analysis

In contrast to the previous techniques, static analysis techniques allow the detection of some classes of program errors without ever executing the program.

These classes include deadlocks, synchronisation errors and access anomalies in shared memory systems [16]. Since static analysis examines all possible program executions, it can detect these errors, even if they don't occur in the program's execution by chance.

Many of these techniques supplement each other, so that a powerful debugger may implement a large subset of them. In the following chapter, we will focus on the event based approach and introduce a transformation-based method for efficient event recognition.

3 Event Based Debugging with the DETOP Debugger

3.1 The Basic Programming Model

We assume a programming model for a distributed memory multiprocessor based on light weight processes and message passing. Such a programming model has been implemented within the Multiprocessor Multitasking Kernel (MMK) developed by our research group [5]. The MMK offers three kinds of programming objects, that can be dynamically created and deleted:

1. **tasks**, i.e. light weight processes for computing
2. **mailboxes** for communication between tasks
3. **semaphores** for synchronization purposes

The combination of different mailbox buffer sizes and timeout parameters in the communication calls allows for a number of different communication schemes (e.g. buffered/unbuffered, blocking/nonblocking ...). All objects can be accessed independently of their current location in the distributed system by means of globally unique identifiers (location transparency). This not only simplifies programming, but also allows for object migration and dynamic load balancing.

3.2 The DETOP Debugging Model

A debugger for a parallel system is largely influenced by the underlying programming model. The concept of location transparency leads to an abstraction from the distributed machine architecture and thereby defines a virtual machine (i.e. one processor executing the whole program). This concept simplifies both program coding and debugging, because the user need not have in mind the physical locations of the programming objects. On the other hand it forces the debugger to provide the user with the same virtual machine. This means, the debugger must allow a global view of the distributed program and must implement some kind of (virtual) global time. The ability of dynamic creation and deletion of objects also influences the debugger. The main problems to solve are how to name these objects and how to keep the debugger informed about all existing objects and their locations without introducing a large communication overhead.

In the next paragraphs we will describe a method for the implementation of distributed breakpoints that uses transformations to achieve efficient monitoring. The underlying basic concept is the event action model. In this model the user can specify some events he is interested in and relationships between these events. Possible events include sending and receiving of messages or execution of a specified statement. In general, relationships between events can be described by temporal logic or more specialized description languages [9]. In the proposed scheme we use two different descriptions: On the user level, a simple event description language is used. The event recognition algorithm uses deterministic finite automata to represent event relationships.

Once events, relationships and actions have been defined, the debugger permanently monitors the specified events. If they satisfy the given conditions, then the actions are performed, e.g the program is halted. Other actions are also possible, for instance start or stop a timer, trace an event etc.

This scheme is well suited for parallel debugging since it is a natural extension of the methods provided by usual sequential debuggers. This helps users to move from sequential to parallel programming environments. In addition it allows efficient monitoring, because no tracing or logging of large data sets is necessary. In contrast to the behavioural approach predicates are usually small and especially independent of program size, because the user only specifies the behaviour he is interested in. As will be shown distributed monitoring of the predicate is possible in a scalable way, so large and massively parallel programs can be efficiently supported. Finally, since actions are not limited to program halting, the event action paradigm is not only useful for debugging, but also for performance monitoring or visualization.

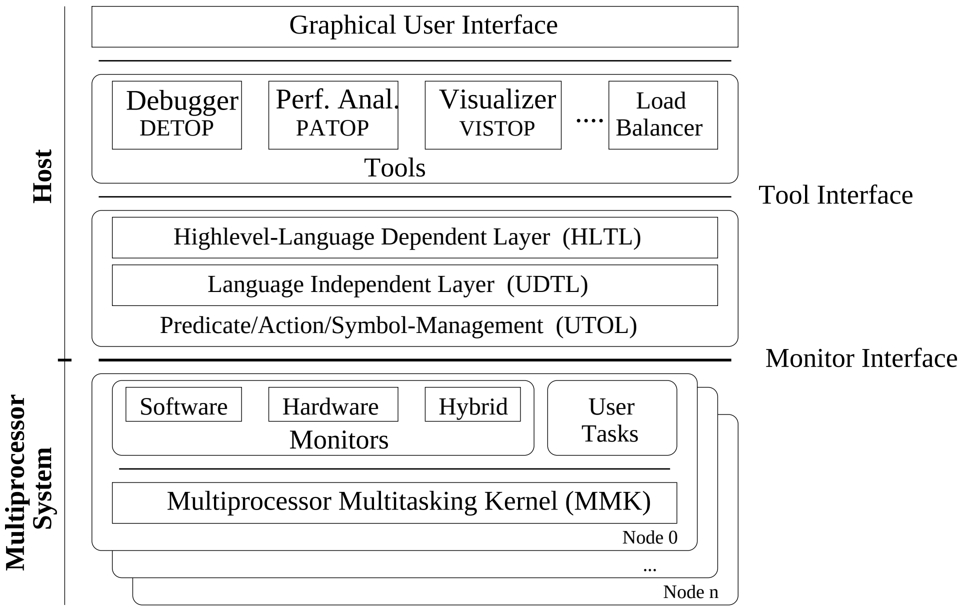

Fig. 1. Hierarchical model of TOPSYS

3.3 Architecture of the DETOP Debugging System

DETOP is part of the TOPSYS programming environment [3], that is built in a hierarchical manner, as shown in fig. 1. We have defined two important interfaces:

The **monitor interface** allows to observe a target program at the machine level. TOPSYS uses a distributed monitoring system with a monitor attached to each node of the multicomputer. Each monitor recognizes local events (occurring on the monitor's node), tests relationships between these events and performs local actions. To implement global predicates and actions, the monitors can exchange messages via a virtually fully connected communication network. Beside a pure software implementation, there is also a hybrid version that uses special hardware and a dedicated communication bus to detect and combine events [4].

The **tool interface** offers the same functionality at the level of source language and programming model and provides a global view of the distributed system. So while the tool interface allows to specify global events related to the source code, e.g. 'execution of line 23 in file foo.c' or 'sending a message to mailbox A', the monitoring system is based on machine code and hooks in the operating system, e.g. 'execution of address $1e34' or 'insertion into send queue'.

A central layer named UTOL is responsible for the transformations between these abstraction levels and the management of predicates, actions and symbol tables. To increase portability there are two sublayers: the high-level language dependent part (HLTL) contains parsers for commands and expressions and also handles complex data types. The language independent layer (UDTL) translates symbolic names into addresses and vice-versa. It also implements the predicate transformation described in the next chapter.

Execution Events	Data Events	Communication Events
execute(task,procedure)	write(task,variable)	send(task,mailbox,msg_ptr)
execute(task,line)	read(task,variable)	receive(task,mailbox)
	assign(task,variable,value)	request(task,semaphore,units)
		release(task,semaphore,units)

Table 1. User level events (wildcards for objects are allowed)

3.4 Monitoring of Global Predicates

As stated before, we use a simple event description language to define global predicates on the user level. Such a predicate consists of an arbitrary number of events (see table 1) that may be combined with any of the following operators:

$\vee$: $A \vee B$ becomes TRUE when A has become TRUE or B has become TRUE
$\wedge$: $A \wedge B$ becomes TRUE when both A and B have become TRUE
$\rightarrow$: $A \rightarrow B$ becomes TRUE when B becomes TRUE after A has become TRUE

Note that for $A \wedge B$ to be fulfilled, A and B don't need to occur simultaneously. In fact, $A \wedge B$ is equivalent to $(A \rightarrow B) \vee (B \rightarrow A)$ which means both events occur in arbitrary order. Of course, also other operators would be possible, e.g. 'between' or 'repeat', but until now we had no need for these operators. However, our implementation could be easily extended to support more complex event relations.

In contrast to most other approaches our monitoring system does not directly interpret the global user predicate. Instead, each monitor only gets those parts of the predicate that can be supervised locally on the monitor's node. These parts are determined by inspecting the objects that participate in the events. If only one object is involved, the event can only be detected on the object's node. However, if two objects participate in the event, e.g. 'task A sends to mailbox B', it is possible to detect the event on the task's and the mailbox's node, giving us the chance to take an optimal choice. For some events, e.g. 'some task communicates with some mailbox' no participating object is known. In this case the event must be monitored on all processing nodes.

The mapping from object identifiers to the corresponding node numbers is established by a localizer module within the UTOL. Since objects may be created and deleted dynamically, the localizer in general can not determine an object's node without communication with the target system. In order to keep this communication infrequent, we take a cache-based approach. The localizer holds an internal table that contains object identifiers and the corresponding node numbers. If an identifier is found in this table, no communication is needed. In the other case, the localizer requests each node to send the identifiers of all existing objects on this node and rebuilds the whole table. This method is possible since identifiers are globally unique, not reused and objects are currently not migrated. If an object, that has been deleted, is still in the localizer table, a monitor may receive an event description with a non-existing object in it. In this case the monitor can either issue an error message or simply ignores the event, which can be done since the event can never occur.

To achieve distributed global event detection, there must be some communication between the monitors. In our debugging system we use a concept that we call "crosstriggers". A crosstrigger can be viewed as a Boolean variable whose initial value is FALSE and that can only be changed to TRUE. They are used to realize both communication and the '$\vee$' operator. For example, assume that $P = (a_0 \rightarrow b_1) + (c_2 \rightarrow d_2)$, where the indices indicate the nodes that can detect the subpredicates. Distributing this predicate using crosstriggers results in $P = CT0$; $CT1 \ \models a_0$; $CT0 \ \models CT1_1 \rightarrow b_1$; $CT0 \ \models c_2 \rightarrow d_2$. The '$\models$' operator means that the crosstrigger on the left hand side is set to TRUE when the predicate on the right hand side evaluates to TRUE.
In the software monitor the crosstriggers are implemented by messages, where

- $CTx \models P$ means: associate action 'send message of type x' with predicate P
- CTx_i on a right hand side means the event 'message of type x arrives at node i'

In our hardware monitor a crosstrigger is a single wired-or signal in the 8-bit wide crosstrigger bus.

A monitor could be based directly on this representation of a global predicate with the following algorithm: Whenever an event occurs on the monitor's node, evaluate all the local predicates and perform the associated actions, i.e. either send a crosstrigger or state that the user predicate is fulfilled.

However, to keep the run time overhead of event detection as low as possible and to allow for an efficient implementation of the detection algorithm as a hardware module, we decided to compile the set of predicates for each node into a deterministic Mealy type automaton, defined as $\mathcal{A} = (S, E, A, g, d, s_0)$, where S is the set of states, s_0 is the initial state, E is the set of possible events, $g : S \times E \rightarrow S$ is the state transition function and $d : S \times E \rightarrow A^*$ is the output function, where the output consists of sequences of local actions. This automaton is built by simulating the parallel evaluation of all predicates on one node. Fig. 2 shows an example of this transformation.

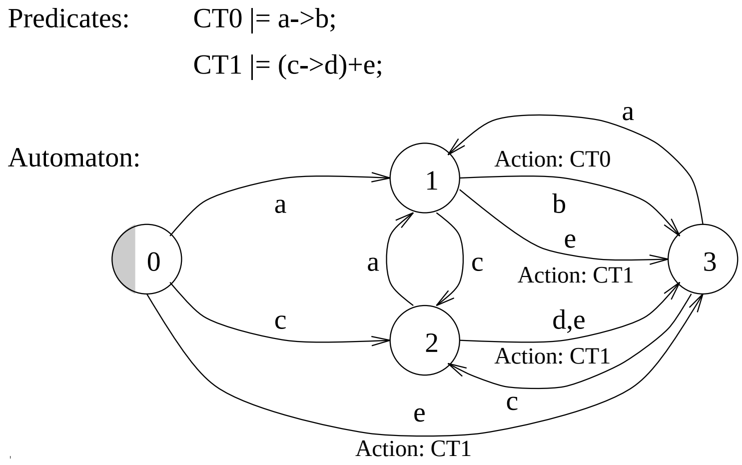

Fig. 2. Example of predicate translation

Although this step requires additional computing effort when a predicate is defined, it greatly reduces the overhead during the monitoring of the predicate, since the event detection algorithm is now a simple automaton interpreter. This procedure is comparable to the compilation of a high level language to a machine language. As in a real compiler the predicate transformation can include some optimizations, for instance to reduce communication between monitors. In this way we can further increase the scalability of the monitoring scheme.

Once the automata for each node have been built, transitions triggered by complex predicates can be replaced by automata that describe the event in terms of more simple events. Therefore the monitor can be used more versatile, because it needs only deal with a small class of primitive events, while user level events may have an arbitrary complexity. This transformation is also important for use with the hybrid monitor, since it cannot directly detect the events of interest.

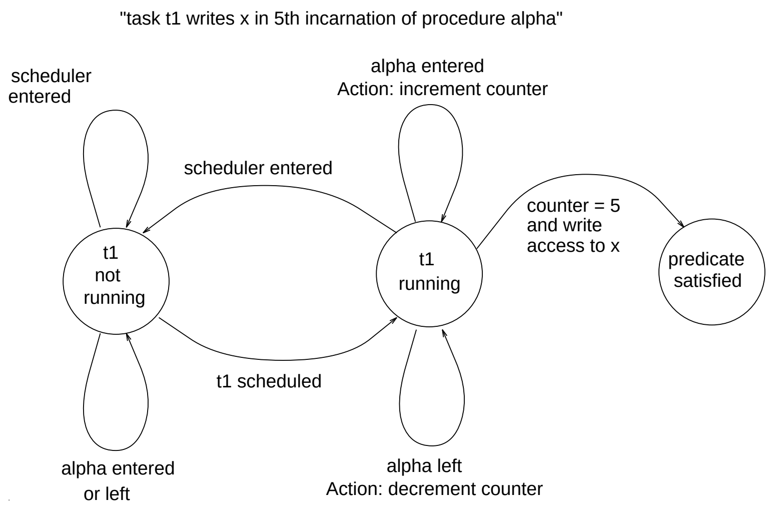

Fig. 3. Automaton for a simple predicate

So, for example, the event 'task t1 writes x in the 5th incarnation of procedure alpha' is replaced by the automaton in fig. 3.

The software monitor represents the automaton as a set of so-called "OR-blocks", one for each state. An OR-block is an array of pairs (event, action), where the transition to another state is regarded as a special action. This data structure makes monitoring of predicates simple: When an event occurs, the software monitor looks at the current OR-block and searches for a matching entry. If such an entry is found, the associated actions are performed, otherwise the event is ignored.

The hardware monitor uses a RAM based state machine that is programmed by the monitor software when a predicate is defined. Once programmed this state machine is an exact hardware representation of the automaton. Control-flow, data and crosstrigger events are detected directly by the hardware via programmable comparators based on tag-RAMs. Communication events must be converted to data events by store instructions that are inserted into the communication routines.

As shown, the strategy of transforming and distributing predicates has a couple of advantages:

- Predicate supervision is completely distributed.
- Large parts of the predicate can be monitored locally without any communication overhead.
- Communication is minimized, because there is no need to broadcast every single event. Furthermore, because in most cases it is statically known, which monitor generates a crosstrigger and which ones may receive it, broadcasts can be replaced by cheaper point-to-point communications.

- Event processing during monitoring is cheap, since much of the work has been done during predicate transformation.
- The monitoring algorithm can be easily implemented in hardware and is independent of the events and operators offered at the user level.

4 The Distributed Debugger DETOP

Based on the monitoring scheme introduced above, we have built the distributed debugger DETOP. The debugger uses a graphical interface offering commands that fall into five major groups:

1. **Execution control:** This group contains commands to start, stop and single step tasks.
2. **Inspection of program state:** These commands allow to retrieve information about the program's state, e.g. a list of all currently existing objects, their mapping to processing nodes, scheduling state of tasks, message and task queues of mailboxes, value and type of variables.
3. **Modification of program state:** Currently only modification of variables is supported.
4. **Breakpoints and traces:** As stated in chapter 3.4 distributed breakpoints can be defined based on the events in table 1. The user can also specify events that will be logged to a trace. Starting and stopping the trace can be controlled by a predicate with an associated trace trigger action.
5. **Miscellaneous:** Beside configuration menus and on-line help, multiple listing windows can be used to view source code, to define execution events and to display the current line of execution of a task.

Figure. 4 shows how breakpoint definitions are supported by the user interface. In the small window in the upper left, the user defines the overall structure of the predicate, i.e. chooses between different event types and operators. The detailed definition of each event and the associated actions is done in separate windows in order to keep the display clear. All objects are accessed with symbolic names whenever possible. In the example, all tasks are stopped, when task *upper* receives a message from mailbox *mb4* that has been sent by *task_3*.

In order to evaluate the efficiency of the proposed event monitoring strategy and as a basis for further optimizations we have thoroughly investigated the probe effect that is introduced by DETOP's software monitor. For this purpose, parts of the monitor have been modelled with timed petri nets, where each transition represents a program part of the monitor. In order to measure the time needed by these transitions we have used the hardware monitor to observe the software monitor in real-time.

One of the results that were achieved on an iPSC/2 is shown in fig. 5 (from [2]). The figure shows that due to the preprocessing done by the UDTL, analyzing the event and switching to the next state in the predicate automaton is very fast compared with the other transitions. It can also be seen that the task switch to the monitor's analyzer task is very expensive, so in the current implementation of

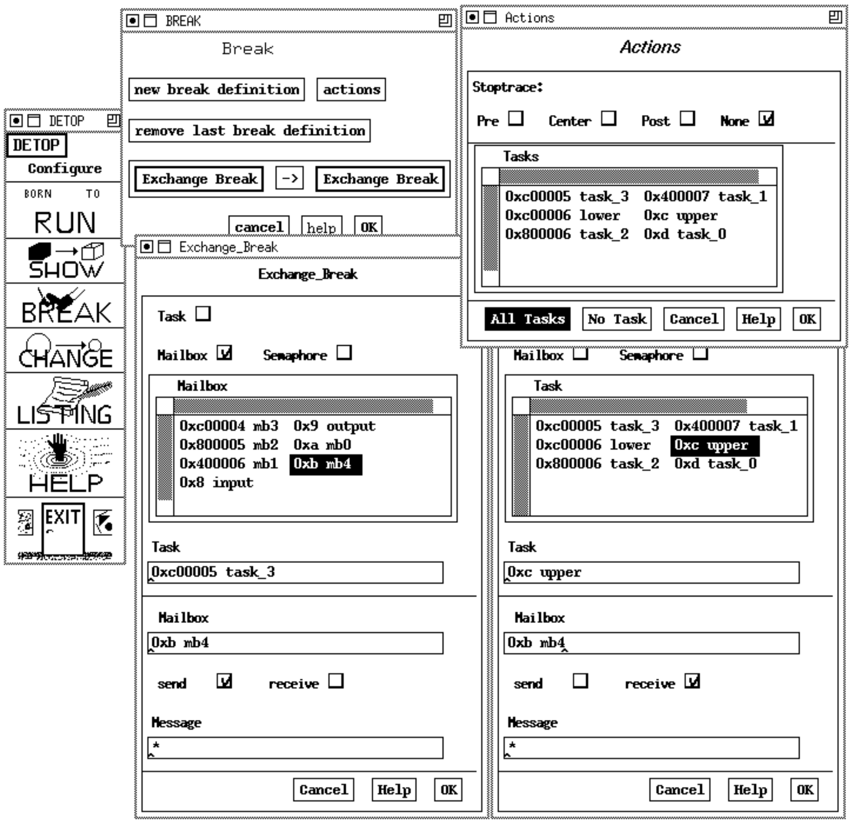

Fig. 4. Definition of a breakpoint with DETOP

the monitor we decided to analyze events in the context of the interrupt handler or the user task whenever possible to avoid the task switch.

References

1. P. Bates and J. C. Wileden, "An Approach to High-Level Debugging of Distributed Systems", In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, vol. 18(8) of *ACM SIGPLAN Notices*, pp. 107–111, Aug. 1983.
2. T. Bemmerl, J. Bauer, and T. Treml, "Leistungsanalyse von verteilten Beobachtungs- und Bewertungswerkzeugen", SFB-Bericht 342/14/90A, TU München, München, July 1990.
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.
4. T. Bemmerl, R. Lindhof, and T. Treml, "The Distributed Monitoring System of TOPSYS", In *Proceedings of the CONPAR '90 - VAPP IV Conference*, Zürich, Schweiz, Sept. 1990.

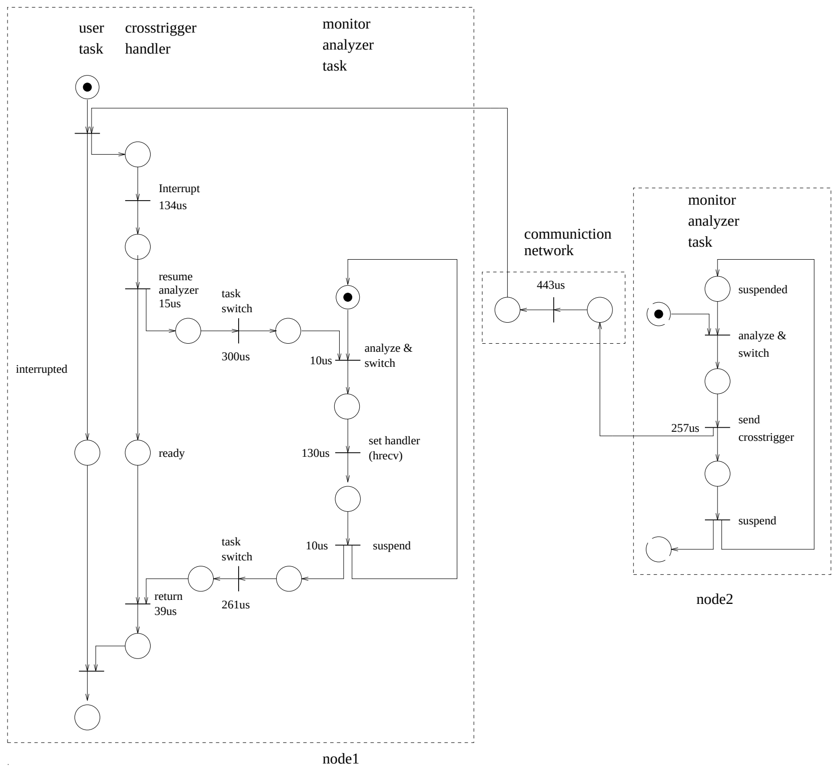

Fig. 5. Timed petri net of a crosstrigger event

5. T. Bemmerl and T. Ludwig, "MMK - A Distributed Operating System Kernel with Integrated Dynamic Loadbalancing", In *Proceedings of the CONPAR '90 - Vapp IV Conference*, Zürich, Schweiz, Sept. 1990.
6. R. H. Carver and K.-C. Tai, "Replay and Testing for Concurrent Programs", *IEEE Software*, vol. 8, no. 2, pp. 66–74, Mar. 1991.
7. J. Fowler and W. Zwaenepoel, "Causal Distributed Breakpoints", In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pp. 134–141, Paris, May 1990.
8. G. C. Fox, M. Johnson, G. Lyzenga, O. S. W., J. Salmon, and D. Walker, *Solving problems on concurrent processors*, Prentice Hall, Englewood Cliffs, 1988.
9. D. Haban and W. Weigel, "Global Events and Global Breakpoints in Distributed Systems", In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, vol. II, Sortware Track, pp. 166–175, 1988.
10. J. R. Larus, "Abstract Execution: A Technique for Efficiently Tracing Programs", *Software - Practice and Experience*, vol. 20, no. 12, pp. 1241–1258, Dec. 1990.
11. T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay", *IEEE Trans. Comput.*, vol. C-35, no. 4, pp. 471–481, Apr. 1987.
12. D. May, "The Next Generation Transputers and Beyond", In A. Bode, editor, *Distributed Memory Computing, Proceedings of the 2nd European Distributed Memory Computer Conference*, vol. 487 of *Lecture Notes in Computer Science*, pp. 7–22,

München, Apr. 1991, Springer-Verlag, Berlin, Heidelberg, New York.
13. B. P. Miller and J. D. Choi, "A Mechanism for Efficient Debugging of Parallel Programs", In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, vol. 23(7) of *ACM SIGPLAN Notices*, pp. 135–144, Atlanta, Georgia, June 1988.
14. B. P. Miller and J. D. Choi, "Breakpoints and Halting in Distributed Programs", In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pp. 316–323, Los Alamitos, California, 1988, CS Press.
15. J. Rattner, "The New Age of Supercomputing", In A. Bode, editor, *Distributed Memory Computing, Proceedings of the 2nd European Distributed Memory Computer Conference*, vol. 487 of *Lecture Notes in Computer Science*, pp. 1–6, München, Apr. 1991, Springer-Verlag, Berlin, Heidelberg, New York.
16. R. N. Taylor and L. J. Osterweil, "Anomaly Detection in Concurrent Software by Static Data Flow Analysis", *IEEE Trans. Softw. Eng.*, vol. SE-6, no. 3, pp. 265–277, May 1980.