Real-Time Execution Monitoring

BERNHARD PLATTNER, MEMBER, IEEE

Abstract—Today's programming methodology emphasizes the study of static aspects of programs. In practice, however, monitoring a program in execution, i.e., monitoring a process, is routinely done by any programmer whose task it is to produce a reliable piece of software. There are two reasons why one might want to examine the dynamic aspects of a program: first, to evaluate the performance of a program, and hence to assess its overall behavior; and second, to demonstrate the presence of programming errors, isolate erroneous program code, and correct it. This latter task is commonly called "debugging a program" and requires a detailed insight into the innards of a program being executed.

Today, many computer systems are being used to measure and control real-world processes. The pace of execution of these systems and their control programs is therefore bound to timing constraints imposed by the real-world process. As a step towards solving the problems associated with execution monitoring of real-time programs, we develop a set of appropriate concepts and define the basic requirements for a real-time monitoring facility.

As a test case for the theoretical treatment of the topic, we design hardware and software for an experimental real-time monitoring system and describe its implementation.

Index Terms—Debugging, monitor, performance evaluation, process interaction, process monitor, real-time monitoring, timing.

I. Introduction

PROGRAM testing, debugging, and performance analysis is a common task for any programmer or engineer who is involved in the development of software products. Perhaps just because it is daily routine to monitor the way a program executes, this topic has not been treated adequately in the research literature of the last twenty years. Reference [1] presents the field systematically and discusses fundamental concepts that have emerged in the past.

In this paper we concentrate on the task of monitoring the execution of real-time programs. Research literature in this area is even harder to find—[2] even complains that real-time software is the "lost world" of software testing and debugging. Recent papers, however, seem to indicate that this topic is increasingly gaining attention [5], [6].

This section defines our notion of real-time execution monitoring, i.e., *what* it is, *why* we do it, and *how* we plan to achieve it.

A process is created when we execute a program, feeding it with a set or a time sequence of input values (the input state) and expecting a set or time sequence of output values (the output state). A process is not repeatable, as it incorporates time as a parameter. In most cases, however, we expect processes that are derived from the same program to exhibit a deterministic behavior with respect to the transformation of the input state to the output state.

Real-time processes are processes whose correctness depends on their behavior with respect to time (i.e., time in the sense of *real*-world *time*). Often, real-time processes are closely coupled to processes existing in the real world (real-world processes), e.g., a chemical plant or a tool machine. Generally, they analyze data generated by the real-world process and/or control the latter. The pace of execution of a real-time process is therefore not determined by internal criteria, e.g., the execution speed of the underlying processor, but by the timing constraints imposed by the real-world process. A proper operation of the whole system—consisting of the real-time target process and the real-world process—depends on the capability of the target process to comply with certain timing conditions. In most cases, these conditions define an upper bound on the time that can be allowed for processing events that occur in the real-world process. As a consequence, a monitoring system must not affect the timing of the target process it monitors in a way that could violate these conditions. If a monitoring system conforms to this requirement, we call it a *real-time monitor.* A real-time monitor enables us to observe the behavior of the production version of a real-time process, in order to collect genuine data, usable for statistics (system and performance analysis), or to detect illegal or unexpected process states (identifying erroneous behavior, debugging). This is very useful in an environment where a simulation of the normal system behavior is not possible, or in cases where software errors only manifest themselves when production is run.

One might argue that with ever increasing processing powers, addressing capabilities, and decreasing hardware costs, capabilities for debugging, testing, and performance monitoring should be built into a product. We agree with this opinion, but experience shows that economic reasons often force the developer to use ad hoc techniques to analyze a product. Therefore, it is important to provide standardized diagnostic tools that fulfill the requirements of real-time monitoring.

We think that the act of monitoring a process should be conducted on a *symbolic level*, that is, the operator communicates with the monitor in terms of the source code the monitored process is written in. We also assume that the programming language used is a modern high level language, like Pascal and its relatives.

The means by which the operator expresses his intentions about what is to be monitored is the *monitoring statement.* It consists of two parts, a *predicate* and an *action.* The predicate is a Boolean expression defined on the state space of the monitored process (to be elaborated in the next section). Whenever a predicate evaluates to a true value, the corresponding action

Manuscript received November 30, 1982; revised January 24, 1984.

The author was with Neu-Technikum Buchs, Buchs, Switzerland. He is now with the University of Zurich, 8057 Zurich, Switzerland.

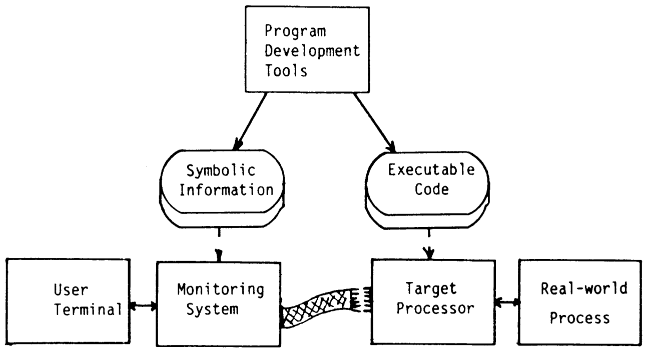


Fig. 1. System configuration for real-time monitoring.

is executed. Actions may consist of operations like incrementing a counter, saving current variable values for later processing, or halting the monitored process. The former operations are normally associated with performance and data analysis, the latter with debugging.

In the following sections we will use the terminology and the hardware and software configuration illustrated in Fig. 1. For the sake of simplicity, we assume that the program development process (program design, coding, compiling, and linking) produces only two sets of data: the executable code, which will eventually be loaded into the *target processor*, and which is bare of any provisions for execution monitoring, and a file containing relevant information (like symbol tables generated by the compiler and the linker), which will be used by the *monitoring system*. The monitored process will also be called the *target process*; it runs on the *target processor* and communicates with the real-world process. The *monitor* or *monitor process* is implemented in the monitoring system and is interactively controlled by the operator. Fig. 1 introduces the monitoring system and the target processor as two physically separate entities. This is necessary as

- it should be possible to add monitoring capabilities by simply plugging the monitoring system into the target processor and
- if we ask that the addition of the monitoring system should not disturb the timing of the target process, it is virtually impossible to share the target processor between target process and monitor process.

II. Process States and Predicates

A. *Modeling the Target Process State*

This section introduces a model for the process state. This is a central issue of this paper, as in order to express predicates on the target process state we need a detailed knowledge of its structure. Moreover, the model will serve us as a guideline for the implementation of the monitoring system.

The structure of the target process state depends heavily on the properties of the programming language the target program is written in, and the target program itself. We will therefore investigate the states that may be produced by programs written in different programming languages.

In our model, the state of a process consists of two parts: the *control substate*, which identifies the current point of execution, and the *data substate*, which contains all data that are currently occupied by the process. As we are only considering the level of abstraction given by the programming language, housekeeping information (like linkage data used by the implementation of the programming language or internal data kept by the runtime system) are not considered to be part of the data substate.

The following paragraphs will investigate languages with different properties and discuss the structure of the resulting process states.

1) A primitive language without a procedure concept and without dynamic allocation of variables exhibits static control and static data substates. In this case the control substate simply and typically is the line or statement number at the point of execution. The data substate is static because variables are allocated statically. In other words, the structure of the process state is constant and can be determined by looking at the program text.

2) If we add a procedure concept (excluding recursive procedures) to our language—an average implementation of Fortran IV—the resulting control substate is dynamic but bounded in size; the point of execution can be stored in a stack with maximum size given by the program text. Here the control substate corresponds to the return stack kept in the implementation at runtime.

3) Some implementations of Fortran IV choose to allocate variables dynamically (this saves memory space, as local variables of subroutines can be overlaid), still denying recursion. This leads to the same control substate as in case 2), but to a data substate which is also dynamic and bounded, corresponding to a stack of activation records for each called subroutine. Note that the user must be well aware of this implementation variant: local variables of subroutines do not survive between calls!

4) If we allow recursion (as do Fortran 77, Pascal, PL/1), the control and data substates are still stacks, but we can no longer tell their maximum sizes by looking at the program text. Instead, the sizes depend on the input data to the process, i.e., the input state. This case is discussed in detail below.

5) Introducing user controlled dynamic variables (based variables in PL/1, pointer referenced data structures in Pascal) changes the picture again: the data substate is extended by a dynamic and unbounded structure which does not exhibit a stack-like behavior, as the user may create or delete data elements at any time and in any order. This structure corresponds to the heap of many Pascal implementations.

6) Languages incorporating concurrent computations lead to process states consisting of several substates of type 4) or 5), one for each active computation. The number of substates may remain constant during the whole life of the process, or may vary with time, depending on the properties of the language used.

For the rest of the paper, we assume that the program text of the target process is written in programming language characterized by 4).

Introducing another concept, the *program state space* will help us in the following discussion. The program state space $S(P)$ of the program P is a set that includes all potential states of processes derived from P. It is a Cartesian product of the definition ranges of all process state substates. This definition

```
    PROGRAM Exl;
      VAR x,y,z;
      PROCEDURE P1;
        VAR a,b;
        BEGIN
          :
        END;
      PROCEDURE P2;
        VAR a,x;
        BEGIN
          .
1         CALL P1;
          .
        END;
      BEGIN (* main program *)
        .
        .
1       CALL P2;
2       CALL P1;
3       CALL P2;
        .
        .
      END.
```

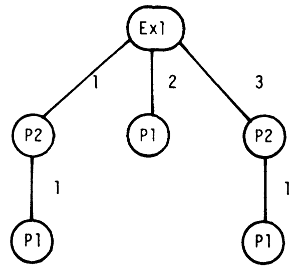


Fig. 2. Nonrecursive program and structure of state space.

```
    PROGRAM Ex2;
      VAR x,y,z;
      PROCEDURE P1;
        VAR a,b;
        BEGIN
          .
          .
        END;
      PROCEDURE P2;
        VAR a,x;
        BEGIN
          :
1         CALL P2;
2         CALL P1;
3         CALL P2;
          :
        END;
      BEGIN (* main program *)
        :
1       CALL P2;
2       CALL P1;
3       CALL P2;
        :
      END.
```

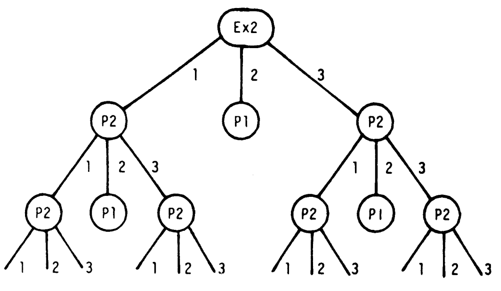


Fig. 3. Recursive program and structure of state space.

deliberately includes states that may be reached by a process, for any input state. An alternative definition of the state space, so as to include exactly the states that can be reached, would be unmanageable; in general, it is undecidable whether there exists a process of P that will ever reach a given state. The chosen definition of $S(P)$ enables us to determine the process state space by looking at P, without considering what input states will be given when executing P. The following examples will clarify this issue.

Fig. 2 presents a simple program written in a very simple Pascal-like language. We assume that local variables of procedures are allocated dynamically.

As program Exl does not use recursive procedures, the resulting process states will be of type 3). A graphical representation of the structure of the state space of Exl takes the form of a multiway tree, which was created using the following recursive algorithm, initially starting at the node labeled Exl.

> Starting at the current node, scan the program text of the associated procedure. If this procedure calls another procedure, create a new successor node, label it with the name of the called procedure, and make this new node the current node. Then execute this algorithm again.
>
> Edges leading to son nodes are numbered sequentially, starting at 1.

Note that this algorithm produces the *structure* of the state space of a program, as opposed to the complete state space. The data substate of individual states may be generated by selecting any node as the current node, following the path up to the root of the tree and assigning legal values (according to the declaration) to any local variable declared in the procedures associated to the nodes along this path. The control substate is contained in the sequence of procedure calls along the path from the current node to the root of the tree.

Fig. 3 differs from the previous example in that $P2$ calls itself recursively. It is obvious that the structure of S (Ex2) now becomes infinite. As a consequence, the algorithm previously used will not terminate, unless we modify it to break the recursion at a certain depth. The resulting tree partially illustrates the structure of S (Ex2), up to a depth of 2. Again we can extract individual states, using the same method as before.

B. Predicates on the Process State Space

The previous discussion enables us to develop a formalism for predicates, which are Boolean expressions on the process state, as we mentioned in the introductory section.

In general, a predicate identifies individual states or a set of states in the state space of the target program. It divides the state space into two regions, a region in which it yields a true value and another in which it evaluates to false.

The notion of program state spaces and predicates has been useful in the domain of static program analysis [7]. In this context, a program is being viewed as a predicate transformer. However, these concepts also find an appealing interpretation when applied to dynamic analysis: a process p derived from a program P corresponds to a trajectory of a point moving in the state space $S(P)$. Whenever this trajectory enters a region specified by a predicate, the value of this predicate changes.

A predicate is a Boolean expression defined on the state space of the monitored process. Operands of a predicate may be

1) the control substate of the process state, e.g., in the predicate (referring to Fig. 2):

> "the current point of execution is at the second statement in $P1$, which was called by $P2$, which in turn was called by the first call in Exl"

2) elements of the data substate of the process state (also called *state variables*):

> "variable x in Exl is written to"
> "variable x in Exl equals variable y in Exl"

At this point a problem arises: How can we unambiguously specify state variables that are instances of local variables? Obviously the variable name "x" is ambiguous, as there are two variables with this name in our example. Using textual inclusion, i.e., qualifying the variable name with the name(s) of the enclosing procedure(s), e.g., "$P2\backslash x$" where "\" denotes textual inclusion, yields ambiguous references too. In Fig. 3, due to recursion, several instances of "$P2\backslash x$" exist concurrently.

The previously developed tree representation of the structure of the state space (see Fig. 3) offers a solution. This tree has one node for each potential activation of a procedure. The path from the root to a node uniquely identifies that node, thus a path expression for a node (using the numbers of the edges in the tree Ex2) may serve as a unique name of a procedure activation. Combining this name with a variable name yields an unambiguous reference to a state variable.

Examples of state variable references (see Ex2 above):

"$1:1\backslash a$" refers to the variable a in $P2$ which was called by the first call in $P2$, which in turn was called by the *first* call in Ex2.

"$3:1\backslash x$" refers to the variable x in $P2$ which was called by the first call in $P2$, which in turn was called by the *third* call in Ex2.

The same notation is useful to formally specify predicates on the control substate of the process state.

"Control substate = $1:1:2\backslash 5$" identifies the statement no. 5 in $P1$, which was called by the second call in $P2$, which was called by the first call in $P2$, which in turn was called by the first call in Ex2.

It is interesting to note that a predicate on a part of the control substate is integrated into each state variable reference, e.g., in "*3:1*$\backslash x$." The interpretation is straightforward: this state variable only exists, and its value is only defined, if the predicate

"control substate = $3:1*$"

is true, where "*" denotes "any legal string."

This discussion prompts us to mention important properties of state predicates.

1) Common parts of the control substate can be "factored out."

"$1:3:1\backslash a = 1:3\backslash x$"
is equivalent to
"$1:3:\{1\backslash a = x\}$"

2) A predicate can only be completely evaluated if all state variables that appear as operands exist. An evaluation may be possible with

"$1:3:1\backslash a + 1:3:1:2\backslash b = 0$"
but definitely not with
"$1:3:1\backslash a + 1:3:2\backslash b = 0$"

as in the second case the procedure activations designated by "1:3:1" and "1:3:2" exclude each other.

3) If a predicate is formulated in a way that does not exclude an evaluation, there exists a condition on the control substate that must be fulfilled before the predicate can be evaluated. If with

"$1:3:1\backslash a + 1:3:1:2\backslash b = 0$"

condition "control substate = $1:3:1:2*$" yields true, all state variables in the predicate exist and the predicate can be evaluated. We call this condition the *evaluation condition.*

III. Principle of Implementation

This section will be devoted to the implementation of an experimental real-time monitoring system. We will first outline the design of the monitoring hardware and software, and finally give details about an actual implementation.

Monitoring the target process means continually observing its state and evaluating the predicates that were submitted to the monitor by the operator. On the other hand, the state of the target process is used and updated by the target processor. Thus, the target process state is a shared resource of two concurrent computations, the monitor process and the target processor. Standard methods for coordinating accesses to the target state (i.e., some kind of mutual exclusion of the monitor and the target processor) would clearly delay the target processor and therefore violate the requirements for real-time monitoring outlined above.

A. Hardware Support for the Monitor Process

As it is not possible to directly use the target process state for real-time monitoring purposes, we decouple the monitor from the target process by inserting a first-in/first-out queue (FIFO) between the target processor and the monitor. The information flowing through the FIFO should allow a complete reconstruction of an image of the target process' state as described below.

In a typical implementation of the target processor (like today's mini- or microcomputers), there are places where all information about the target process is available: at the communication channel between the processor and the memory (typically the system or memory bus) or at the "pins" of the CPU (central processing unit) board or chip. In some implementations, the two variants could be identical. Using standard technology, it is possible to build a device which "listens" to the transactions occurring between the target processor and the other parts of the system. This device is connected to the memory bus and is designed to perform its task without affecting the behavior of the target processor (and consequently also of the target process). This "bus listener" [8] is incorporated into the design of the monitoring hardware that will be discussed below (see Fig. 4).

1) All memory transactions generated by the CPU of the target processor are recognized by the *target processor interface*, which assumes the function of the "bus listener" mentioned above. The data involved in referencing a memory location, i.e., its address, content, and whether the memory location is to be read or written to, are fed into an FIFO.

2) At the output of the FIFO, the temporarily stored data about memory transactions are used to manipulate the *phantom memory*. The phantom memory is a dual port memory, which is accessed by the monitor process and the target processor interface. Physically, the phantom memory has the same address range and the same word size as the memory of the target processor. We say that the phantom memory is *consistent* with the target processor memory if at time t it has exactly the same

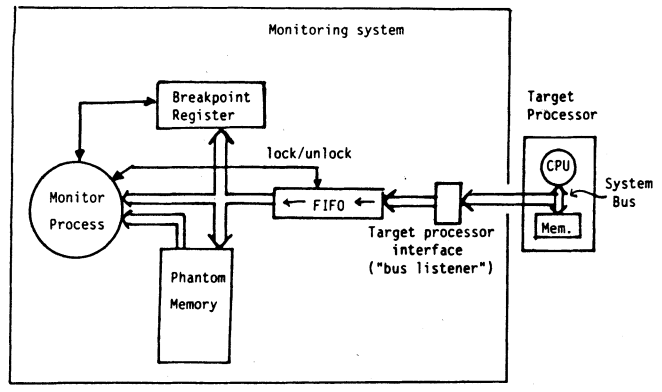


Fig. 4. Real-time monitoring hardware.

content as the target processor memory had at the time $t - td$, where td is the delay introduced by the time it takes the data about memory transactions to pass through the FIFO. Herewith we assume that both memories are properly initialized.

3) The monitor process may read the content of the phantom memory at any time. It may also lock the output of the FIFO, preventing any modification of the phantom memory while the lock is set. During that interval, data about memory transactions that keep on entering into the FIFO are queued. As soon as the monitor process unlocks the FIFO again, they leave the FIFO and have their delayed effect on the phantom memory.

The monitor process may also interpret directly the sequence of memory transactions leaving the FIFO, using the data path from the FIFO.

As long as the FIFO does not overflow, the scheme described above preserves the consistency between the target processor memory and the phantom memory. Obviously, the hardware must assure that the peak data rate between the FIFO and the phantom memory is several times the rate of memory transactions generated by the target processor, so that a full FIFO may be emptied within a reasonable interval.

The chosen arrangement has several important implications.

1) The monitor process now has assumed an additional role. Instead of only executing monitoring statements, using the process state as a resource, it also has to interpret on-line the low level information available in the phantom memory and perhaps at the output of the FIFO, in order to construct a high level image of the process state. Just what high level image it should produce is the topic dealt with in the remainder of the section.

(In this context, "low level" means "close to the machine" and "high level" is "close to target program source level.")

2) The implementation of the monitor process must guarantee an upper bound for the time required to process any state transition occurring in the target process state and to execute the monitoring statements as required. Assuming that the rate at which memory transactions are generated is constant and that the length of the FIFO is finite, the FIFO fills up within a specifiable and fixed time interval. The monitor has to perform its task within this interval, or the requirement for real-time monitoring would be violated.

3) All information about the entire target state must be available in the phantom memory. If parts of the target state are kept elsewhere, e.g., in CPU registers of the target processor, as is often the case with today's processor architectures and compilers, real-time monitoring is not possible.

4) We have to accept a delay between the moment at which a predicate becomes true and the moment this event is recognized by the monitor process. This delay amounts to the sum of the delay introduced by the FIFO and the processing delay of the monitor.

5) This arrangement does not permit the monitor any modification of the target process state.

A last building block is addea to speed up the monitor process. A *multiple breakpoint register*, connected to the output of the FIFO, reports to the monitor any memory transactions referencing a location belonging to a previously defined set of memory locations. An arbitrary number of breakpoints is made possible by implementing the register as a large array of breakpoint bits, providing one bit for each location of the phantom memory. The breakpoints may be switched on or off at the discretion of the monitor process, providing a fast low level monitoring device with a processing time that is independent of the number of enabled breakpoints.

The monitoring hardware sketched above seems to be the minimum required to achieve real-time monitoring. The phantom memory comprises the first in a sequence of steps that will finally deliver the source level information: it converts a sequence of memory transactions to a copy of the target state. It enables the monitor process to access the "phantom" target process state randomly, without interfering with the target process. The breakpoint register makes possible an event-driven processing of the information kept in the phantom memory.

In the following paragraphs we discuss the other functions of the monitoring system. It may appear reasonable to implement some of these functions in hardware, if a benefit in processing speed can be expected. For the purpose of this paper, we assume that these functions are implemented in software.

B. *Software Design of the Monitor Process*

The functions assumed by the monitor process can be separated into two classes.

1) Tracing low level events that change the process state and reconstructing a high level image of the process state.

This class can be subdivided further into functions that are involved with changes in the structure of the process state (i.e., procedure calls and procedure returns) and functions that deal with changes of state variable values (i.e., assignments to state variables).

2) Executing monitoring statements, i.e., evaluating predicates and executing actions if appropriate.

In the design of the monitor process we follow two policies. One policy is to process as little information as possible, in order to achieve a short processing time. As complete low level information about the process state is already available in the phantom memory, we only generate access information, which permits a high level interpretation of these data. More-

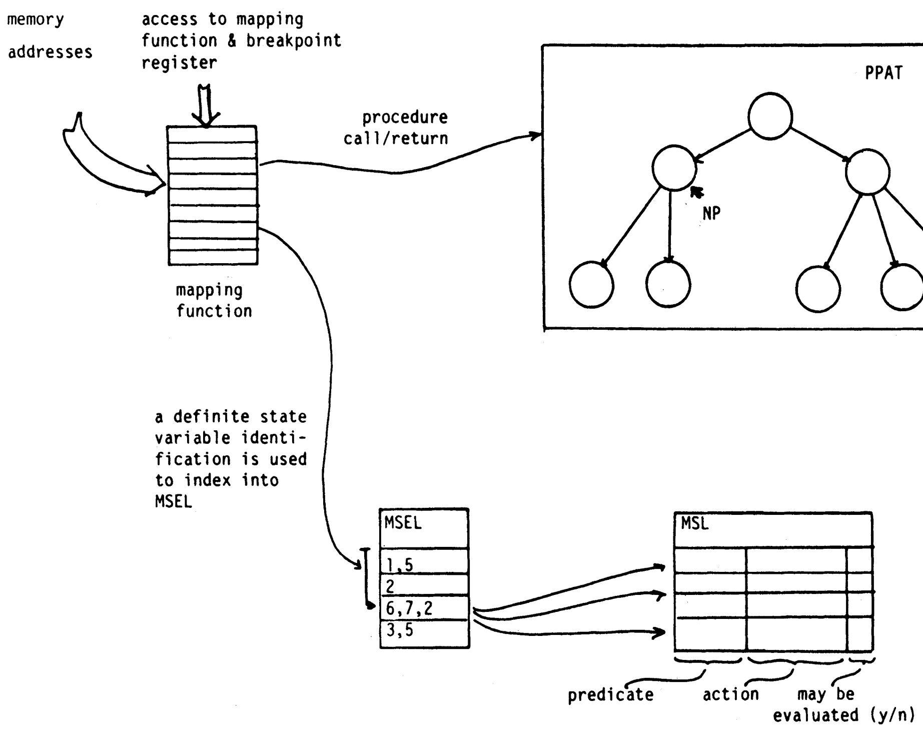

memory addresses
access to mapping function & breakpoint register
procedure call/return
PPAT
NP
mapping function
a definite state variable identification is used to index into MSEL
MSEL
1,5
2
6,7,2
3,5
MSL
predicate
action
may be evaluated (y/n)

Fig. 5. Data structures of the monitor process.

over, we only evaluate predicates when all state variables in a predicate really exist and one of them has been written to. A third step towards a short processing time is to prepare as much information as possible in advance, i.e., use static information structures wherever possible. As implication 2) above requires that processing time be bounded, the other policy is to update all dynamic information structures incrementally, assuring that each increment can be executed within a specifiable amount of processing time.

The discussion in the previous paragraphs leaves us with two open design decisions.

1) How are assignments to state variables dealt with?

2) How do we process procedure calls and procedure returns? We first elaborate on the data structures involved with 1).

The *monitoring statement evaluation list* (MSEL, see Fig. 5) contains, for each state variable reference occurring in a predicate, a list of identifications of monitoring statements that must be executed when the corresponding state variable is written to. The notation for predicates introduced in Section II allows for several state variable references in one predicate. As an example, the predicate

"$1\backslash x - 1:3\backslash a = 1:3:2\backslash b$"

incorporates three state variable references. A single state variable reference may thus appear in the predicates of several monitoring statements. This means that whenever a state variable is written to, a possibly long list of monitoring statements must be scanned and eventually executed. On the other hand, the notation for state variable references is restrictive in that it does not permit accessing multiple state variables in one reference. References of the kind "all variables with the name *xy*" are not supported. In consequence, the total number of state variables relevant for predicate evaluation remains constant once the operator is finished with entering new monitoring statements. This makes it possible to give each state variable a unique identification, which can be used as an index into MSEL. As the set of monitoring statements also remains constant while monitoring, MSEL is static and can be computed before monitoring starts.

Monitoring statements are given unique identifications and are stored in the *monitoring statement list* (MSL). Each monitoring statement consists of a Boolean value, a representation of its predicate, and a representation of its action. The Boolean value indicates whether the predicate can be evaluated, i.e., whether all its state variables exist (see also Section II). This value is frequently modified while monitoring, as it depends on the current point of execution. It is the only dynamic part in MSL.

Now we are ready to discuss the *potential procedure activation tree* (PPAT), the data structure associated with the processing of procedure calls and procedure returns. The execution of a program, with procedure calls and procedure returns frequently happening, may be viewed as a tree walk in a tree of the type introduced in Section II and shown in Fig. 6. If we assume that there is a node pointer NP pointing to the node in the tree corresponding to the current procedure activation, then calling a procedure means moving NP to the appropriate son. Inversely, a return from a procedure consists in moving NP to the father of the current node. The position of NP reflects the structure of the target process state at an instant in time.

However, on procedure calls and returns the monitor process also needs to update the breakpoint register and the MSL.

When a procedure is called, new instances of local variables come into existence. If one of these new variables is a state variable that is referenced in any of the predicates, a breakpoint must be activated at the corresponding memory location, so that a later assignment to this state variable will cause the proper monitoring statements (the identifications of which are stored in MSEL) to be executed.

Additionally, monitoring statements must be activated if the evaluation condition (Section II) of their predicates is fulfilled.

Graphically, the evaluation condition of the predicate identifies one node in this tree; it is fulfilled whenever NP is positioned on or beyond this node.

Summarizing, the data structure used to keep track of the structure of the state is a tree with one node for each potential procedure activation. Each node contains

1) a list of state variable identifications and the absolute addresses of the corresponding memory locations. This list is used to enable or disable breakpoints. Note that the monitor can assign absolute addresses to state variables in advance, if the variable allocation policy used by the compiler is known.

2) a list of monitoring statement identifications, used to activate or disactivate monitoring statements for predicate evaluation.

3) the linkage information required to efficiently find the father and all son nodes.

The reader might wonder how this scheme works when the target program uses recursive procedures, as in such cases the structure of the state space is infinite, making memory requirements for the PPAT rather large. The solution is simple: the number of state variables that can be referenced in predicates is finite and known before the act of monitoring takes place. Thus the PPAT can be built up to a size where it allows for the allocation of only those data–1), 2), and 3) above–that will really be required.

A node that would not host any information simply need not be created. With this solution, the value of NP only reflects the current point of execution as long as it remains within the boundaries of the PPAT. If the point of execution lies outside the PPAT, a simple counter can be used to keep track of procedure calls and returns.

At this point it might be worthwhile to explicitly explain the role of the breakpoint register. As the monitor process is fully event-driven (events are: 1) accesses to state variables referenced in a predicate, 2) procedure calls, and 3) procedure returns), we need a device which is able to report such events with a small delay which is independent of the number of pending events. This can be achieved efficiently using specialized hardware as described above.

C. Timing Analysis

We now show that the design of the monitoring system fulfills the requirements of real-time monitoring.

The requirement for real-time monitoring is that to each individual monitoring activity, a worst case processing time can be assigned by looking at the program text and having a

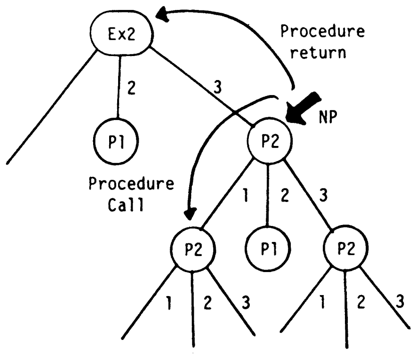


Fig. 6. PPAT and motion of the node pointer NP.

```
LOOP
  "wait for a signal from the breakpoint register"           (1)
  "transform the address of the breakpointed memory
   location to an event type and identification"             (2)
  CASE event type OF

    procedure call:
      "move NP in PPAT to the son node given by the
       event identification"                                 (3)
      "activate breakpoints according to the list of
       state variable identifications in this node"          (4)
      "activate monitoring statements according to the
       list of monitoring statements in this node"           (5)

    procedure return:
      "disactivate breakpoints and monitoring statements     (6)
       and move NP to the father of the current node"

    assignment:
      FOR "each monitoring statement identification in       (7)
       in the entry of MSEL given by the event
       identification" DO:
         IF "the monitoring statement is active" THEN
           "evaluate the predicate"                          (8)
           IF "the predicate yields true" THEN
             "execute the action"                            (9)
      END FOR
  END CASE
END LOOP
```

Fig. 7. Outline of the monitor process.

detailed knowledge about the monitoring statements that were specified by the operator. In particular, the worst case processing time must not depend on the input state given to the target process.

In order to discuss the processing times involved with the various activities of the monitor, we refer to the outline of the control flow in a sample monitor process (Fig. 7) designed according to our proposal.

1) The response time is constant under all circumstances.

2) Using a mapping function, the time required for the transformation can be kept constant. Data compression techniques may be used to minimize storage requirements.

3) The response time is constant under all circumstances.

4) Processing time depends on the content of the node, but a worst case can be specified.

5) Processing time depends on the content of the node, but a worst case can be specified.

6) Activities are symmetric to 3), 4), and 5); thus the processing time is the same as that required for procedure calls.

7) The maximum number of monitoring statement identifications for each entry of MSEL is known, i.e., a worst case value can be computed.

8) Processing time depends of the complexity of the predicate, but can be specified.

9) The processing time depends on the actions that are implemented. The implementation should only offer actions with specifiable processing time.

Consequently, a monitoring system can predict if a given set of monitoring statements can be processed in real-time. However, the processing power of the monitoring system and the target processor, the source program, the code generation details of the compiler used for the target processor, and finally the length of the FIFO used in the monitoring hardware have to be taken into account for this prediction.

D. An Experimental Monitoring System

An experimental real-time monitoring system has been implemented using a DEC LSI-11/2 processor as the target processor and a PDP-11/23 (running the RT-11 operating system) as the monitor. The two processors are linked with a custom-built interface that implements the monitoring hardware as described below.

The programming language used on the target processor has deliberately been kept simple. It is a variant of PL/0 [3], a block structured language similar to a subset of Pascal that includes recursive procedures. Variables are restricted to the data type integer. The compiler was readily available as a Pascal source program and was modified to generate code for the LSI-11 and to produce the information needed for the monitoring system.

An implementation restriction limits the number of monitoring statements that can be specified by the user. Currently the limit is set to 10. This limit could be increased easily at the cost of increased memory requirements.

The user may specify predicates on the data substate in the form

<state variable reference> <relational operator> <value>

where

<state variable reference>	is a reference to a state variable as introduced in Section II
<relational operator>	is chosen among "=," "<>," ">," ">=," "<," "<="
<value>	is an integer constant

The actions implemented comprise

Halt	Stop the monitoring process and return to command level
Set <counter> <value>	Set any of 10 counters to <value>
Increment <counter>	Increment counter no. <counter> by 1
Decrement <counter>	Decrement counter no. <counter> by 1
DispValue <stvref>	Display the value of the state variable identified by <stvref>
DispCounter <counter>	Display the value of the counter no. <counter>
CondHalt <counter> <value>	Execute "Halt" if counter no. <counter> reaches <value>

A sequence of 10 actions may be specified in each monitoring statement. This implementation restriction is due to memory constraints.

The monitoring system was entirely implemented in MODULA-2 [4], a modern high level language that supports data abstraction. This capability was heavily used for encapsulating the data structures described in Section III. MODULA-2 proved to offer a comfortable and reliable programming environment.

The limited virtual address space of 56 kbytes of the PDP-11 proved to be a bottleneck in the implementation of the monitor. Though not much effort was put into saving memory, it is obvious that for a practical system more addressable memory is an absolute requirement.

The experiences with the experimental system indicate that the small difference in processing power between the LSI-11/23 and the LSI-11/2 (the throughput ratio is approximately 2.5:1) is insufficient. It limits the feasibility of real-time monitoring to cases where procedure calls and returns do not happen too frequently. It also limits the complexity and number of monitoring statements that can be submitted to the monitor. Effectively, only one monitoring statement could be monitored in real-time. We estimate that the ratio in processing power should at least be 10-15 if the system were to be used practically.

IV. Conclusions

We have introduced real-time monitoring as the act of observing the sequence of states of a process and assigning a truth value to each of those. How this truth value is to be evaluated is specified by means of a predicate. A positive evaluation of a predicate initiates an action, which can be used to record information about the process. We have discussed the feasibility of real-time monitoring and its implications on monitoring system design. We have proposed a monitoring system which is capable of monitoring in real-time, and on a symbolic level, processes written in a high order, block structured programming language. An experimental implementation shows the practicability of the design.

In our opinion, further research should be directed at the issues listed below.

1) Given a target program, a target processor and compiler, a set of monitoring statements, and a monitoring system implemented according to the proposed design, is it possible to monitor in real-time? The monitoring system should be able to answer this question.

2) Today's programming languages offer a much larger set of facilities than block structure and recursive procedures. Is real-time monitoring possible with user controlled variable allocation and concurrent computations? How do these language properties influence monitoring system design?

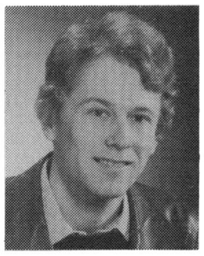
3) Which aspects of processor architecture and compiler design affect real-time monitoring? How can machine architects and compiler writers support real-time monitoring?

4) Which (software) parts of the proposed real-time monitoring system would be more efficient if implemented in hardware?

5) An aspect that was not touched at all is the possibility of including the concept of time (in the sense of *actual time*) into the framework of execution monitoring. Time might appear as a "read-only" variable either in the process state or in the monitor state. Using time in predicates would enable the user to measure the performance of the target process in absolute or relative units of real-time. In this context we should add that modern logic development systems already offer tools of this kind.

References

[1] B. Plattner and J. Nievergelt, "Monitoring program execution, a survey," *IEEE Comput. Mag.*, vol. 14, pp. 76-93, Nov. 1981.
[2] R. L. Glass, "Real-time: The 'lost world' of software debugging and testing," *Commun. Ass. Comput. Mach.*, vol. 23, pp. 264-270, May 1980.
[3] N. Wirth, *Algorithms + Data Structures = Programs.* Englewood Cliffs, NJ: Prentice-Hall, 1976, ch. 5.
[4] ——, "MODULA-2," Berichte des Inst. fuer Informatik ETH, Zurich, Switzerland, Mar. 1980.
[5] L. M. Lemon, "Hardware system for developing and validating software," in *Proc. 13th Asilomar Conf. Circuits, Systems and Computers*, Pacific Grove, CA, November 5-7, 1979, pp. 455-459.
[6] C. A. Witschorik, "The real-time debugging monitor for the Bell System 1A processor," *Software–Practice and Experience*, vol. 13, no. 8, pp. 727-743, Aug. 1983.
[7] E. W. Dijkstra, *A Discipline of Programming.* Englewood Cliffs, NJ: Prentice-Hall, 1976.
[8] R. E. Fryer, "The memory bus monitor," in *Proc. AFIPS Nat. Computer Conf.*, vol. 42, 1973, pp. 75-79.



Bernhard Plattner (S'78–M'80) received the diploma in electrical engineering from the Swiss Federal Institute of Technology (ETH), Zurich, Switzerland, in 1975, and the Ph.D. degree in computer science, also from ETH, in 1983.

He has served as a Consultant for several Swiss companies, and was a Lecturer in Computer Science at the Neu-Technikum Buchs, Buchs, Switzerland, where this work was performed. He is now with the University of Zurich, Zurich, Switzerland. Apart from execution monitoring, his areas of interest are computer networks and operating systems.

Dr. Plattner is a member of the Association for Computing Machinery.

Monitoring for Deadlock and Blocking in Ada Tasking

STEVEN M. GERMAN

***Abstract*—We present a deadlock monitoring algorithm for Ada tasking programs which is based on transforming the source program. The transformations introduce a new task called the monitor, which receives information from all other tasks about their tasking activities. The monitor detects deadlocks consisting of circular entry calls as well as some noncircular blocking situations. The correctness of the program transformations is formulated and proved using an operational state graph model of tasking. The main issue in the correctness proof is to show that the deadlock monitor algorithm works correctly without having simultaneous information about the state of the program.**

In the course of this work, we have developed some useful techniques for programming tasking applications, such as a method for uniformly introducing task identifiers.

We argue that the ease of finding and justifying program transformations is a good test of the generality and uniformity of a programming language. The complexity of the full Ada language makes it difficult to safely apply transformational methods to arbitrary programs. We discuss several problems with the current semantics of Ada's tasks.

***Index Terms*—Concurrent algorithms, concurrent programming languages, correctness proofs of concurrent programs, deadlock detection, exceptions, program transformations, semantics of Ada tasking, state graph models, task identifiers.**

Manuscript received October 1, 1982; revised January 10, 1984. This work was supported in part by the Rome Air Development Center under contract F30602-80-C-022 while the author was at Stanford University.

The author is with the Computer Science Laboratory, GTE Laboratories, Waltham, MA 02254.

I. Introduction

We consider the problem of detecting deadlock in the tasking mechanism of a high level programming language, Ada. Our approach is to systematically transform an original program P into a new program P', such that P' has a potential deadlock iff P does, and P' signals whenever a deadlock occurs. In principle, the transformations can be applied mechanically, giving a practical tool for debugging deadlocks. Since we modify only the source program, our method can be used with any Ada implementation, without special knowledge of the implementation of tasking.

The main contributions of this paper are deadlock detection algorithms for the entry call model of multiprocessing, and discussion of certain problems in the current semantics of Ada's tasks. Specifically, we introduce state graphs appropriate for modeling deadlock in entry call processing, define a class of