数理解析研究所講究録
第655巻 1988年 20-39

EDEN – An Event-Driven Execution Monitor for Ada® Tasking Programs : Implementation and Application

程 京徳　　白木原 敏雄　　荒木 啓二郎　　牛島 和夫
Jingde CHENG, Toshio SHIRAKIHARA, Keijiro ARAKI and Kazuo USHIJIMA

Department of Computer Science and Communication Engineering
Kyushu University
Hakozaki, Higashi-ku, Fukuoka 812, Japan

Abstract

This paper describes the design, implementation and applications of an event-driven execution monitor for Ada tasking programs named EDEN. We discuss two major problems which confronted us in developing the execution monitor, i.e., what to observe in monitoring an Ada tasking program, and how to reduce the interference from the monitoring actions. These are intrinsic problems in monitoring concurrent programs. In this paper we describe our approaches which we adopted to those problems in designing and implementing EDEN. This paper also presents practical applications of EDEN including communication deadlock detection. Finally we assess our monitoring mechanism and our monitor EDEN.

Keywords Concurrent programming, Ada tasking, Execution monitoring, Monitoring domain, Monitor transparency, Communication deadlock detection, Ada programming support environment

1. Introduction

Debugging concurrent programs is more difficult than debugging sequential programs because the former presents a higher level of complexity than the latter. Effective tools and methodologies which are developed specifically for concurrent programming are hardly available so far. Traditional debugging tools and methodologies for sequential programs do not provide enough information and means to deal with the problems in debugging concurrent programs. Therefore, we must develop new tools and methodologies in order to deal with the new level of complexity.

The events that occur in a concurrent program during its execution may be categorized into two groups : sequential (or local) and concurrent (or interactive) events. A sequential event concerns a local action inside the process such as sequential

® Ada is a registered trademark of the U. S. Government (Ada Joint Program Office).

control transfer or access to the local data. A concurrent event relates to an interaction between processes such as inter-process synchronization or communication. The behavior of a concurrent program may be regarded as streams of such sequential and concurrent events. In order to debug a concurrent program, it is indispensable to provide information about the events occurred during execution of the program. Such information can collected by monitoring executions of the program.

We particularly treat Ada [DoD-83] programs with concurrent tasks. In this paper we describe the design, implementation and applications of an event-driven execution monitor for Ada tasking programs. This monitor named EDEN collects, analyses, saves, and reports information about the tasking behavior of the target program. It also detects tasking communication deadlocks in the program at run-time. The description of the tasking behavior reported by EDEN is presented at the Ada source code level. In order to deal with the most inherent problems in monitoring Ada tasking programs, we devoted our attention to the concurrent events in Ada programs and neglected the sequential events when we developed EDEN.

The central problems we treat in this paper are : what to observe in monitoring an Ada tasking program, and how to reduce the interference from the monitoring actions. These two problems are intrinsic for monitoring concurrent programs. Therefore, we discuss them from the viewpoint of both Ada programming specifically and concurrent programming in general.

For the first problem, we have defined the monitoring domain of Ada programs with which we describe the monitored tasking behaviors of the programs. The monitoring domain of a target Ada program consists of tasking events which occur and are observed at the run-time of the program. For the second problem, we have presented a new concept, named partial order transparency, as a criterion used to measure the transparency of an execution monitor with respect to the monitored behavior of a target program.

Recently, several execution monitors and/or debuggers with an execution monitoring approach for concurrent programs have been proposed and/or developed with various aims [Baiardi-86, DGC-84b, Gait-85, German-84, Helmbold-85a,b, Holdsworth-83, LeDoux-85, Maio-85, Mauger-85, Smith-85], but none of these monitors and/or debuggers systematically deals with the above two problems. Accordingly, it could be difficult to present significance, sufficiency, and accuracy of the information provided by the monitors and/or debuggers about the monitored behavior of a target program.

In section 2 we give a brief overview of EDEN. In section 3 we discuss the problems in monitoring concurrent programs, and present our approaches to them. We introduce the monitoring domain for Ada tasking programs and describe our monitoring mechanism adopted in EDEN. Section 4 describes the implementation of EDEN. Section 5 presents some practical applications of EDEN. Concluding remarks are given in section 6.

2. Overview of EDEN

EDEN consists of a preprocessor (3,000 lines of text) and a run-time monitor (7,000 lines of text). We implemented it as an Ada package on a Data General ECLIPSE MV/10000 [DGC-84a]. We aimed that it can be easily introduced in any APSE [DoD-80]. Currently, EDEN is a prototype system and serves for experimental use.

2.1 Monitoring and Debugging Process

The basic idea of monitoring mechanism of EDEN is as follows : the preprocessor transforms a target Ada program P into another Ada program P' such that P' preserves tasking semantics of P. The transformed program is compiled, linked with the run-time monitor, and then executed. During its execution, P' will communicate with the run-time monitor when each tasking event occurs in P' and passes information about the tasking event of P' to the run-time monitor. The run-time monitor can analyze, save and report the collected information as the tasking behavior of the target program P. Fig. 1 shows the monitoring process with EDEN.

2.2 Debugging Facilities

EDEN provides a number of basic debugging facilities. It currently supports :

a) Reporting snapshots of task states

The user can designate task(s) in the target program whose snapshots at the current or designated time are to be reported, e.g., a single task, all dependent tasks of a designated task, or all tasks. Each snapshot includes, for designated or each elaborated task, the name in source code, the internal identification number, the state, and if any, the entries ready for communication, or the communicating entry and the communicating task.

b) Reporting snapshots of entry queue states

The user can designate entry or entries in the target program whose snapshots at the current or designated time are to be reported, e.g., a single entry, all entries of a designated task, or all entries. Each snapshot includes, for designated or each entry, the members and their order in the entry queues.

c) Reporting snapshots of task and entry queue states conditionally

EDEN reports a snapshot in the same way as above a) and/or b) according to the condition which the user specifies on the value of a task state or entry queue state.

d) Reporting dependence relationship between tasks and their masters

The user can designate task(s) and/or time as in a).

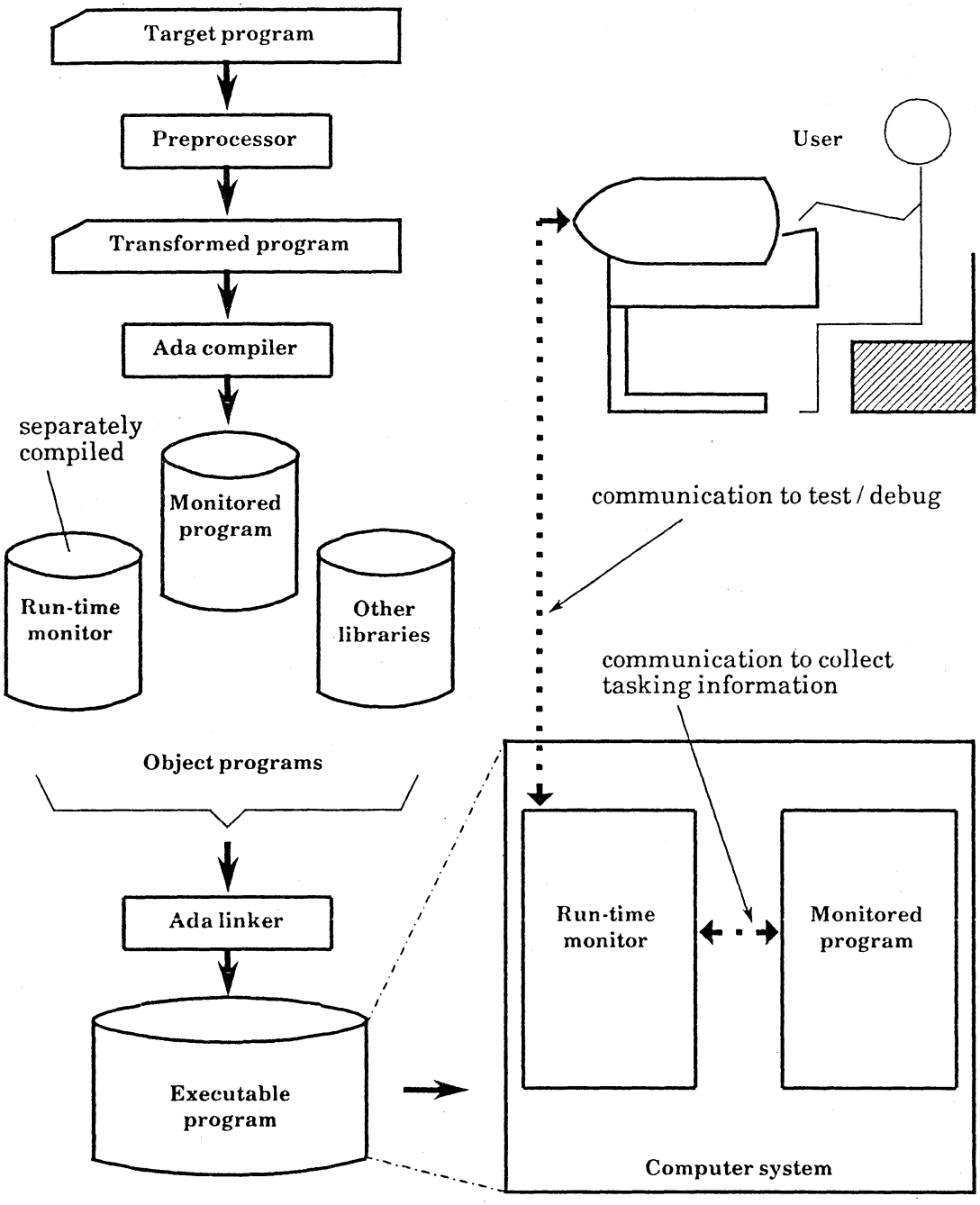


Fig. 1 Monitoring process with EDEN

e) Reporting histories of task states

EDEN reports the execution histories as sequences of task states of the target program for the designated duration. When a task state is a communication state, the communicating entry is also reported. The user can designate task(s) and/or time as in a).

f) Reporting histories of entry queue states

EDEN reports the histories of the operations on entry queues of the target program for the designated duration. The user can designate entry or entries as in b).

g) Breaking

EDEN breaks the execution of the target program at run-time in response to the user's command or the condition which the user specifies on the value of a task state or entry queue state.

h) Saving execution histories

EDEN saves the execution histories of the target program to the designated external files.

i) Detecting tasking communication deadlocks

EDEN detects some kinds of tasking communication deadlocks just before they occur during execution of the target program, and then reports the tasks and communicating entries involved in the deadlock (we shall describe tasking communication deadlocks in section 5.2).

By using these facilities, the users of EDEN can easily understand what the program is doing and how it is being executed, and test and/or debug an Ada tasking program at the Ada source code level. EDEN provides two modes of usage : the interactive and batch modes. At the interactive mode, as the interface is menu-driven, and the user may easily select a command.

3. Monitoring Mechanism

In order to monitor the execution of a concurrent program at run-time, we must solve the problems pointed out in section 1. We rewrite those two problems below (1 and 4), and also present two additional problems deeply related (2 and 3) :

1) What is the concurrent behavior of the target program? What behavior of the program should be monitored and is monitorable?

2) How to identify uniquely each process for monitoring its behavior?

3) How to collect information about the concurrent behavior of the target program?

4) How to reduce the interference from the monitoring actions of the execution monitor as little as possible and guarantee the accuracy of information reported by the execution monitor?

The complete solutions of these problems depend on the concurrent programming language to be used. Here, we present our approaches in designing the monitoring

mechanism of EDEN. We believe that the principles of our strategies used in EDEN are also applicable to execution monitors for other concurrent programming languages.

3.1 Tasking Event Space

When developing an execution monitor for concurrent programs, the first problem is deciding what is the concurrent behavior of the target program, and what behavior of the program should be monitored and is monitorable? We call the set of such monitorable entities the monitoring domain. This problem is very important for the monitor because of three reasons. First, it determines the domain where the monitor works. Second, it is a basis for formally defining and discussing various properties (e.g., completeness [Plattner-81], pertinence [Plattner-81], and transparency) of the monitor. Third, it is a criterion for practically evaluating the monitor.

Now, we define the monitoring domain of EDEN. In constructing a formal model of a physical system, it is a good strategy to define the basic concepts in terms of attributes that can be directly or indirectly observed or measured. To describe the tasking behaviors of Ada programs, we define and use tasking events as primitive entities which are detectable as atomic tasking actions at Ada source code level and regard the tasking behavior of a task as a tasking event stream. Then a state of a task which describe what that task is doing may be defined by two contiguous tasking events. Most of the works in debugging Ada programs (including our earlier studies) use states of task as primitive entities to describe the tasking behavior of Ada programs. But, we have found out that tasking events are more suitable as primitive entities because a tasking event stream provides a higher abstraction of the tasking behavior than a state stream does. It become possible that we devote our attention only to the task interaction without concerning details inside tasks.

The *lifetime* of a task is the duration from the moment at which the task to be elaborated to the moment at which the task to terminate.

First of all, we decide what kinds of tasking events are detectable as atomic tasking actions of a task at Ada source code level; and assign a different name to each kind. According to the syntax and semantics of Ada tasking [DoD-83], the following kinds of tasking events may occur in the lifetime of a task, we specify each kind by a *tasking event name* and a informal description of its semantics below :

1) *Elaboration start* : The elaboration of the object declaration of the task is to be started.

2) *Activation start* : The elaboration of the declarative part of the task is to be started.

3) *Activation completion* : The elaboration of the declarative part of the task is completed.

4) *Execution start* : The first statement of the task body is about to be executed.

5) *Creation* : An allocator which creates some task(s) is evaluated during the execution of the task.

6) *Creation completion* : The evaluation of an allocator which creates some task(s) during the execution of the task is completed.

7) *Entry call* : The task issues an entry call (simple, conditional or timed) to another task.

8) *Acceptance* : The task is ready at an accept statement for accepting any corresponding entry call.

9) *Selection* : The task is ready at a selective wait statement for selecting any of its opened select alternatives.

10) *Entry call cancellation* : An entry call (conditional or timed) issued by the task is canceled because the corresponding rendezvous can not be started immediately or within the specified duration.

11) *Selection cancellation* : A selective wait is canceled because the corresponding rendezvous can not be started immediately or within the specified duration.

12) *Rendezvous* : The task has issued an entry call to another task which has accepted this entry call; or the task has accepted an entry call issued by another task.

13) *Continuation* : The task which has issued an entry call resumes its execution as the result of the completion of the corresponding rendezvous; or the task has completed the execution of an accept statement and continues its execution.

14) *Abort* : The task executes an abort statement.

15) *Aborted* : The task is aborted as the result of the execution of an abort statement.

16) *Block activation start* : The elaboration of the declarative part of a block statement in the body of the task is to be started; or the elaboration of the declarative part of a subprogram called by the task is to be started.

17) *Block activation completion* : The elaboration of the declarative part of a block statement in the body of the task is completed; or the elaboration of the declarative part of a subprogram called by the task is completed.

18) *Block execution start* : The first statement of a block statement in the body of the task is about to be executed; or the first statement of a subprogram called by the task is about to be executed.

19) *Block execution completion* : The execution of a block statement in the body of the task is completed or aborted; or the execution of a subprogram called by the task is completed or aborted.

20) *Block termination* : The execution of a block statement in the body of the task terminates; or the execution of a subprogram called by the task terminates.

21) *Activation exception* : An exception is raised during the elaboration of the declarative part of the task.

22) *Execution exception* : An exception is raised (including a propagated exception) during the execution of statements (excluding the accept statement) of the body of the task.

23) *Communication exception* : An exception is raised during the rendezvous of the called task; or such a communication exception is propagated to the task which is calling the corresponding entry.

24) *Completion* : The execution of the body of the task is completed or aborted.

25) *Termination* : The task terminates.

A *tasking event* is a 5-tuple (T, N, E, M, t). Where T is the identifier of a task in which the tasking event occurs; N is a tasking event name; E is a set of entries to be used for communication of T with other tasks, and is defined if and only if N is in any one of the following tasking event names : "Entry call", "Acceptance", "Selection", "Entry call cancellation", "Selection cancellation", "Rendezvous", and "Continuation", otherwise, E is not defined and denoted by $\perp$; M is a set of messages which are passed between the task T and the other task during their rendezvous, and is defined if and only if N is "Rendezvous" or "Continuation", otherwise, M is not defined and denoted by $\perp$; t is time† when the tasking event occurs.

A state of a task which describes what the task is doing may be defined by two contiguous tasking events. Here, we briefly specify each kind of task states by a *task state name* and the detailed discussion can be found in [Cheng-88] : *Elaborated*, *Activating*, *Execution waiting*, *Creating*, *Working for internal affairs*, *Entry calling*, *Accepting*, *Selective waiting*, *Suspended by rendezvous*, *Abnormal*, *Block Activating*, *Block execution waiting*, *Block completed*, *Completed*, *Terminated*.

We may define a partial order on the set of tasking events occurred during execution of an Ada tasking program according to Ada tasking semantics. For example, a task can only be activated after the start of execution of its master and must terminate before the termination of its master, but different tasks of the master may start their execution in any order. This partial order can be regarded as a minimum constraint on the causal order of the tasking events occurred during the execution of the program and is determined only by the tasking semantics of the program. We have formally defined the partial order [Cheng-87b].

Based on the concept of tasking event, the tasking behavior of a task can be regarded as a tasking event stream. An execution history of an Ada tasking program can be

† This time may be physical time in an interleaved implementation of Ada, or may be virtual time [Lamport-78, Jefferson-85] in a distributed implementation of Ada.

thought of as a partially ordered set of tasking events. We call this partially ordered set *tasking event space*. The monitoring domain of EDEN is this tasking event space.

When defining monitoring domain of an execution monitor for concurrent programs, if we can distinctly define concurrent events and sequential events of the programs respectively, then we should do so. Distinguishing concurrent events from sequential events has two advantages. First, it may often expose inherent problems that have to be dealt with in monitoring and may reduce complexity of various properties of the execution monitor. Secondly, since the amount of full information about the behavior of a target concurrent application program generally becomes very huge, providing the full information for the programmers will be infeasible. An acceptable strategy would be providing only information which reflects the most inherent properties of problem. In this case, it is suitable to use concurrent events well defined as primitive entities of monitoring domain.

3.2 Naming Tasks Uniquely

In order for an execution monitor to follow what is happening in the execution of a concurrent program and report both the behavior of each process and the interactions among the processes, the execution monitor must be able to uniquely identify each process instance at run-time. This might not be a problem for such concurrent programming languages in which there is exactly one instance for each declared process as DP [Brinch Hansen-78] and SR [Andrews-82]. But, this is an important problem for such concurrent programming languages as Ada in which there may be multiple instances for each declared process. Moreover, processes can be created at any time during execution and so the number of processes is variable.

In EDEN, for instance, when a statement concerning tasking in a task body is executed, the run-time monitor must be able to know who executes that statement. When a task is called by another task, the run-time monitor must be able to know who is called.

The identifier of a task object in the Ada source text cannot generally be used to uniquely identify that task object inside the task body itself because all task objects of the same task type have the same task body. It is obvious that any Ada run-time support environment must uniquely identify each task object. But, it is difficult for us to use the internal information maintained by the Ada processor. Besides, we cannot use such an internal information since we intend to implement EDEN at the Ada source code level.

According to [DoD-83], "if an application needs to store and exchange task identifiers, it can do so by defining an access type designating the corresponding task objects and by using access values for identification purposes." In this approach, however, we would have to modify and recompile EDEN for each target program because of the strong type checking of the Ada language. This is an impracticable approach for EDEN. Therefore, EDEN has to name internally each task object in a uniform way at run-time in order to uniquely identify each task object.

Some naming strategies for Ada tasks have been proposed such as [Booch-82, German-84]. The basic idea of the naming strategies is adding a new entry to each task such that it receives its internal name from its parent program unit. But, there are two problems in such naming strategies. First, the internal name of a task can only be referred after the activation of that task because a task can not accept an entry call in the declarative part of its body. This means that these strategies cannot be used for monitoring inter-task communication during activations of tasks. Second, these strategies may modify the behavior of the target program because all task objects depending a parent program unit have to start their execution in only order according to calls of the parent to such new entries but not in concurrently according to the original semantics of the target program.

In order to assign a unique name to each task object, we introduced a *task-name-server* as a package into the program library of the existing APSE. Any target program is transformed such that some local objects used to keep the internal names of task are introduced at the head of the declarative parts of task bodies. Such local objects are initialized by invoking a function in the task-name-server. Uniqueness of the task internal names is guaranteed since all the above initializations call the single entry of a task in the task-name-server. This naming strategy can easily be used for monitoring inter-task communication during activations of tasks in the target program and do not modify the execution start order of such tasks [Cheng-86].

This approach can be applicable to naming uniquely the masters of a task in monitoring the tasking behavior of Ada programs. It can also applicable to other applications where it is necessary for each task object to know its own unique identifier.

3.3 Collecting Information about Tasking Behavior

When designing the monitoring mechanism of an execution monitor for concurrent programs, there are alternative methods to collect information about concurrent behavior of the target program. One is the *central collection method* and the other is the *distributed collection method*.

The central collection method is suitable for a multiprogramming and/or multiprocessing environment. In this approach, a central information collector receives and saves information about the behavior of the target program sent by each process. Generally, the information collector should be a process executed concurrently with processes of the target program. A potential drawback in this approach is that the central information collector may become a bottleneck.

The distributed collection method is suitable for a distributed processing environment. In this approach, no central information collector exists, but a central manager may exist. Each process must record its own execution history and send the recorded information as output according to instructions of the central manager. A potential problem in this approach is how to record a full execution history of the target program if a process is allowed dynamically created and eliminated.

We adopt the central collection method in the monitoring mechanism of EDEN. We call a moment in the lifetime of a task a *tasking point* when a tasking event occurs during the execution of the target program. We call a point in the source text of the target program an *information collection point* which corresponds to a tasking point at run-time.

In order to collect information about monitored tasking behavior of the target Ada program, we introduced a *tasking-information-collector* as a task which has a separate entry for each kind of information collection points. We need to find all information collection points in the source text of the program, and insert an entry call to the tasking-information-collector at every information collection point. The information can be passed to the tasking-information-collector through parameters of the entry call. Thus, during the execution of the transformed program, the tasking-information-collector can collect the information about monitored tasking behavior of the target program at each tasking point.

Fig. 2 shows a program transformation rule for collecting information about accept statement. The transformation rule consists of a pattern and a replacement. A term in angle brackets, for instance <id>, is a pattern variable; the program text covered by a pattern variable is copied into the replacement. The pattern variables are restricted; for instance, <id> can only match a identifier. Square brackets enclose optional items.

Pattern:

```
begin
  <statements-1>
  accept <entry simple name>[ <formal part> ][do
    <statements-2>
    [ <text piece-1> return <expression>; <text piece-2> ]
    <statements-3>
  end[ <entry simple name> ];
```

Replacement:

```
begin
  <statements-1>
  Tasking_Information_Collector.ACCEPTANCE(
    TASK_ID, CLOCK, GET_simple_name("<entry simple name>"));
  accept <entry simple name> [ <formal part> ] do
    Tasking_Information_Collector.RENDEZVOUS(
      TASK_ID, CLOCK, GET_simple_name("<entry simple name>"));
    [ <statements-2> ]
    [ <text piece-1>
    Tasking_Information_Collector.CONTINUATION(
      TASK_ID, CLOCK, GET_simple_name("<entry simple name>"));
    return <expression>; <text piece-2> ]
    [ <statements-3> ]
    Tasking_Information_Collector.CONTINUATION(
      TASK_ID, CLOCK,GET_simple_name("<entry simple name>"));
  end[ <entry simple name> ];
```

Fig. 2 Program transformation rule for collecting information about accept statement

3.4 Partial Order Transparency

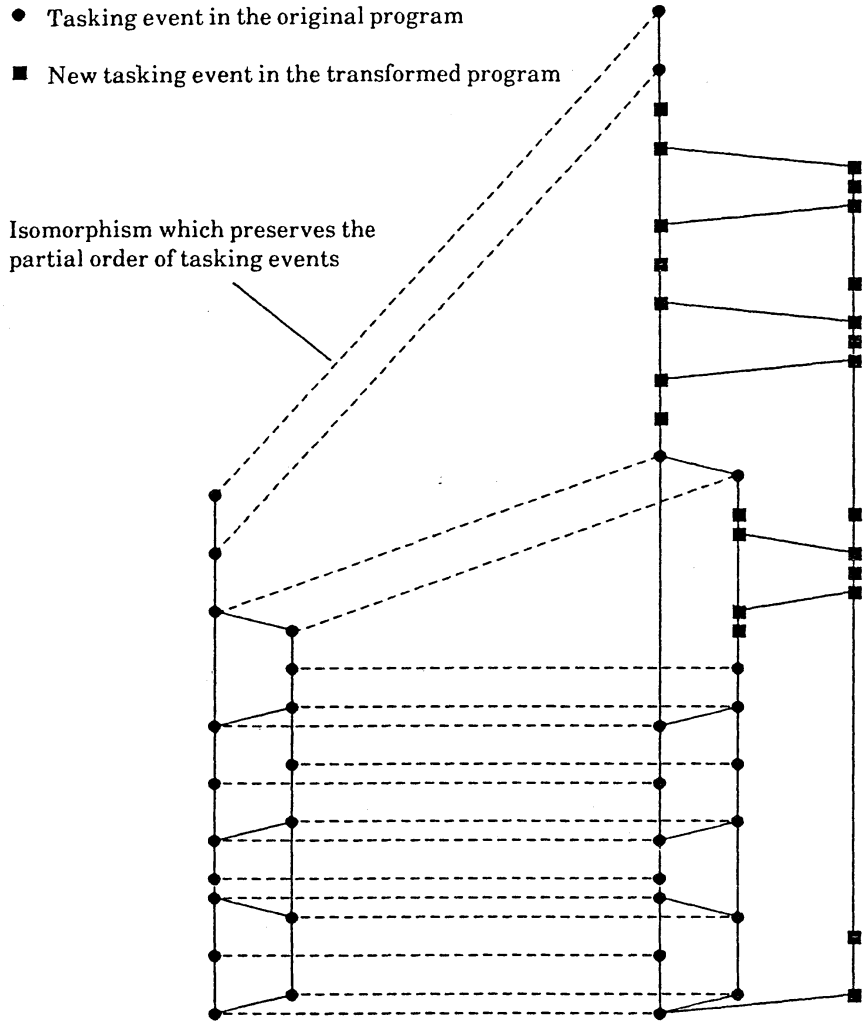
In monitoring the behavior of a concurrent program, the monitoring actions may interfere and even modify the behavior of the program. This is particularly a serious problem for monitoring the concurrent behavior of the programs. It is impossible to completely eliminate this interference because the behavior of a concurrent program generally depends on the rate of each process in the program [Brinch Hansen-73]. Therefore, when we develop an execution monitor for concurrent programs, it is indispensable to make certain whether or not the monitor always report accurate information about the monitored behavior of the target program. In other words, we should establish an acceptable criterion with which we can measure the accuracy of the information about the monitored behavior of the target program reported by the monitor.

When we monitors an execution of a target concurrent program with an execution monitor as EDEN, the observed behavior, of course, is the behavior of the program transformed by the preprocessor of the monitor and not the original behavior of target program. In order to accurately observe and report the monitored behavior of target program, the monitor should and must be transparent to the monitored behavior. We have presented a new concept, called "*partial order transparency*", as a criterion used to measure the transparency of an execution monitor with respect to the monitored concurrent behavior of target program [Cheng-87b]. The basic idea of the concept is making execution monitor transparent to the concurrent events in the program with respect to the order of their occurrence. We have proposed a theoretical foundation for our "partial order transparency" concept. Here, we describe our concept briefly. The reader is referred to the detailed discussion [Cheng-87b].

If we regard an execution history of an Ada tasking program as a partially ordered set of tasking events, then any correctness of the execution can be determined only by the tasking event space of the program. Therefore, when transforming an Ada tasking program, if the tasking events space of the transformed program preserves full properties of the tasking event space of the original program, then we may say that the behavior of the transformed program completely includes the behavior of the original program.

Our approach is based on the viewpoint described above. We first abstract the tasking behavior of Ada programs with respect to the task interaction by using lattice theory. We showed that an execution history of an Ada tasking program forms a lattice consisting of the tasking events occurred in the execution. Then we discuss the correspondence between execution histories of the target program to be monitored and the transformed program, which is really executed during monitoring, in terms of abstract algebraic structures based on the above lattice model. If a program transformation guarantees that there is an isomorphism from the history lattice for the original program to a sublattice of the history lattice for the transformed program, then the partial order of tasking events is completely preserved by the transformation. We claim if the program transformation used in an execution monitor for concurrent Ada programs has the above property, then the monitor is transparent to the

monitored behavior of the target programs in terms of partial order transparency. Fig. 3 shows an isomorphism which preserves the partial order of tasking events in the target program.



(a) Lattice of the original program (b) Lattice of the transformed program

Fig. 3 Preserving the partial order of tasking events

We do not claim it is possible for an execution monitor to completely eliminate the interference from the monitoring actions, but we can develop an interference-free execution monitor in the sense of the partial order transparency. The principles of our approach are also applicable to execution monitors and/or debuggers for other concurrent programming languages.

4. Implementation

Fig. 4 shows the overall structure of EDEN.

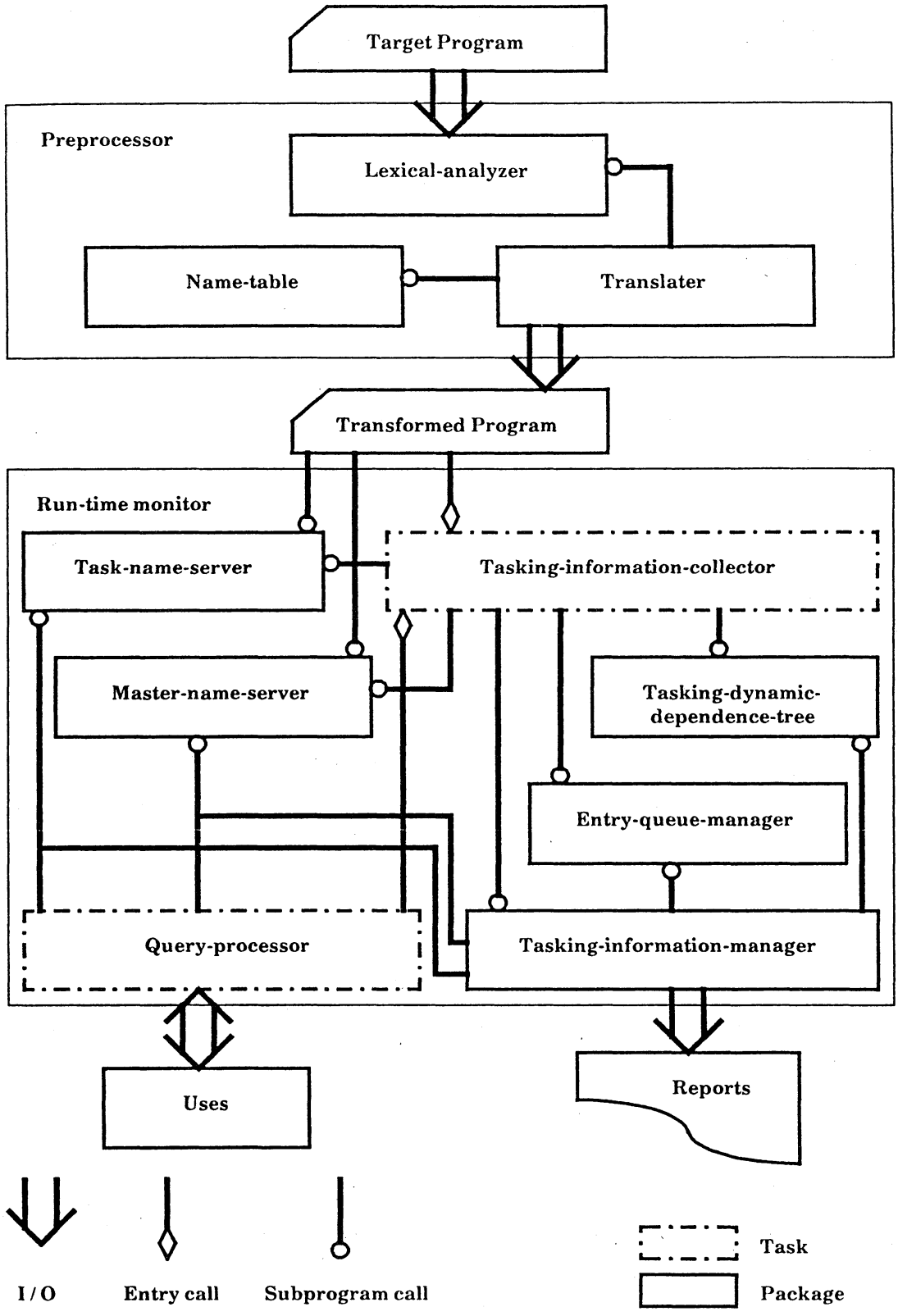


Fig. 4 Overall structure of EDEN

The preprocessor of EDEN transforms a target Ada program into another Ada program which communicates with the run-time monitor. In order to identify declaration of task types and task objects during transformation, a symbol table is necessary in the preprocessor. The run-time monitor of EDEN consists of seven major parts: task-name-server, master-name-server, tasking-dynamic-dependence-tree, entry-queue-manager, tasking-information-collector, tasking-information-manager, and query-processor.

The task-name-server and master-name-server create and manage a task name table and a master name table respectively. Each name-server is implemented as a package.

The tasking-dynamic-dependence-tree is a data structure used for managing dynamic dependence relation between tasks and their masters in the target program. Its nodes correspond to elaborated task objects, currently executing blocks, currently executing subprograms, or a library package. The relation between any two nodes corresponds to the dependence relation between the tasks and their masters. During the execution of the program, when a task object is elaborated, or a block or subprogram is executed, the run-time monitor inserts a new node for it into the tasking-dynamic-dependence-tree. When the task object, block or subprogram terminates, the run-time monitor deletes the corresponding node. The tasking-dynamic-dependence-tree is implemented as a package.

The entry-queue-manager creates and manages a queue for each entries of the target program. It creates a queue for each entry of a task object when the task is elaborated, and deletes these queues when the execution of the task is completed. When an entry of a task is called, the entry-queue-manager inserts an item which contains the internal name of its caller and the calling time into the corresponding entry queue. When the call succeeds in rendezvous or is canceled, the entry-queue-manager deletes the corresponding item from the entry queue. The entry-queue-manager is implemented as a package.

The tasking-information-collector collects information about the tasking behavior of the target program. It is also regarded as the interface between the target program and the run-time monitor. During the execution of the program, entries of the tasking-information-collector are called at every tasking point of the target program. Then the information about tasking behavior of the program is passed through parameters of entries and saved at the tasking-information-manager by the tasking-information-collector.

The tasking-information-manager maintains all information collected by the tasking-information-collector, and reports the information saved in itself. In order to ensure mutual exclusion of read/write operations on the data saved in the tasking-information-manager, all operations for the tasking-information-manager are done only by the tasking-information-collector. The tasking-information-manager is implemented as a package.

The query-processor is a command interpreter. It also serves as a menu-driven user interface of EDEN. The query-processor is implemented as a task.

5. Practical Applications

An execution monitor for concurrent programs can be used as a testing and/or debugging tool to detect and locate the presence of errors. It can be used as a performance evaluation tool, too. But, it is more important that the execution monitor may become a common basis for development of testing and/or debugging tools with various aims in a general concurrent programming environment.

5.1 TwoLevel Debugging

We may distinguish three kinds of errors in a concurrent program with respect to its specification as follows :

1) *Synchronization error* There is an invalid sequence of synchronizations in the program [Tai-85];

2) *Communication error* A process sends/receives an invalid message in the program, or writes/reads an invalid value on a shared variable in the program;

3) *Computation error* There is an invalid sequential control transfer or an invalid read/write operation on a local variable inside a process in the program.

Based on this classification of errors in a concurrent program, we may consider a strategy, called *two level debugging*, for debugging concurrent programs in order to reduce the complexity of the debugging procedure. This strategy is to debug a concurrent program at two levels; i.e., at the higher level (or inter-process level), intending to debug synchronization errors and communication errors for interaction between processes; at the lower level (or intra-process level), intending to debug computation errors for each process.

An event-driven execution monitor for concurrent programs can be used in the higher level debugging as a general tool. We have used EDEN to test and/or debug several small Ada tasking programs. Although EDEN mainly takes notice of the tasking activities in Ada programs, it reports sufficient information about run-time interaction between tasks.

5.2 Tasking Communication Deadlock Detection

A *tasking communication deadlock* in an Ada tasking program is a situation where all members of a group of tasks will wait forever at communication points, and hence can never proceed with their computation. We say that a tasking communication

deadlock is *global* (*local*) if the group of tasks includes all (a part of) tasks in the program. Tasking communication deadlock is one kind of the most typical synchronization errors in Ada tasking programs. Here, we describe briefly the tasking communication deadlock detection. The detailed discussion can be found in [Cheng-87c].

There may occur a variety of tasking communication deadlocks during the execution of an Ada tasking program, e.g.,

1) *Self-blocking* : A task has issued an entry call to itself.

2) *Circular-entry-calling* : There is a closed loop of tasks such that each task has issued an entry call to the next task in this loop.

3) *Dependence-blocking* : The task T_1 is (directly or indirectly) dependent on a block statement in the body of the task T_2 or dependent on a subprogram called by T_2, and T_1 has issued an entry call to T_2. When T_2 proceeds into the execution of the block statement or the procedure call, the block statement or the procedure body will never terminate unless the dependent task T_1 terminates. In this case, however, T_1 will be blocked at the entry call to T_2 which will never be accepted since T_2 is at the block statement or the procedure call and will not execute the accept statement. Thus, T_1 and T_2 will fall into a deadlock.

4) *Acceptance-blocking* : A task is waiting at an accept statement or a selective wait statement for acceptable entry calls, but no entry call is issued to such entries.

We modeled the following two relations by using arc-labeled digraph, i.e., the dependence relation between tasks and their masters, and the entry calling relation among tasks . Based on this model, we showed that the self-blocking, circular-entry-calling, or dependence-blocking occurs iff a cycle exists in the digraph [Cheng-87c].

By analyzing the information about tasking behavior of the target program, EDEN can easily create and operate such an arc-labeled digraph at run-time. EDEN directly detects the self-blocking, circular-entry-calling, or dependence-blocking by detecting a cycle in the digraph.

A global tasking communication deadlock occurs when every active task is frozen at a state and never resumes its execution. EDEN detects a deadlock of this type by counting the numbers of the active tasks and the blocked tasks. An acceptance-blocking can be indirectly detected by detecting a global tasking communication deadlock.

5.3 Other Applications

The information provided by EDEN will be helpful in evaluating the performance of the target program and in discovering bottleneck of the target program. A

performance evaluation tool for Ada tasking programs may work based on the execution history data collected by EDEN.

When we intend to develop a dynamic analyzer for execution properties of the executable statements in Ada tasking programs, a task naming strategy is indispensable in order to uniquely identify each task object as mentioned in section 3.2. In this case, EDEN provides a ready-made task name server. The dynamic analyzer can easily refer to the task name server predefined in the program library in the APSE by using a with clause.

In a database-based debugging approach, debugging a program can be regarded as queries and updates on a database that contains both the program source and the execution state information [Powell-83]. When we intend to develop such a database for debugging in an APSE, EDEN can serve as a ready-made information collector for the database to collect information of the tasking behavior of the target program.

Some knowledge-based debugging systems need an execution trace of the target program as an input [Seviora-87]. When we intend to develop such a knowledge-based debugging system in an APSE, the execution history data of the target program collected by EDEN may be used as the input of the knowledge-based debugging system.

Therefore, EDEN may become a common basis for development of testing and/or debugging tools with various aims in an APSE.

6. Discussions and Conclusions

In comparison with other proposed Ada debuggers such as [DGC-84b, German-84, Helmbold-85a,b, Holdsworth-83, LeDoux-85, Maio-85, Mauger-85], EDEN has the following features :

1) When we develop an execution monitor and/or debugger for concurrent programs, it is indispensable to make certain that the execution monitor and/or debugger should always report accurate information about the monitored behavior of the target program. To solve this problem, it is necessary to define its monitoring domain and to make it transparent to the monitored concurrent behavior of the target program. We dealt with these two requirements systematically when we developed EDEN.

2) EDEN supports most tasking facilities which are not supported by some other Ada debuggers, such as entry families, task types, inter-task communication during activations of tasks, selective wait, conditional entry call, timed entry call, task abortion, and exception handling. Therefore, EDEN is a more useful and powerful tool than such debuggers are.

3) EDEN provides more information about the tasking behavior of the target program than other Ada debuggers do, such as inter-task communication during activations of tasks, task abortion, states of entry queues, and relationship between tasks and their masters. As a consequence, EDEN can detect most tasking

communication deadlocks which can not be detected by other debuggers proposed so far, such as deadlocks that occur in inter-task communication during activations of tasks, dependence-blocking and so on [Cheng-87c]. The users of EDEN can more easily discover abnormal situations by analyzing information provided by EDEN.

4) EDEN provides no facility for breakpoints while some other Ada debuggers do [Maio-85]. We consider that traditional breakpoint facilities are not suitable for debugging concurrent programs because the behavior of a concurrent program generally depends on the rate of each process in the program. Any debugging facility should affect such rates as little as possible.

We are now improving EDEN, accumulating experience with it, and evaluating its usability and effectiveness from the practical point of view. EDEN is aimed to become an effective tool which supports the programming in the large with Ada.

Our experience with the development of EDEN has led us to the following conclusions:

1) An execution monitor for concurrent programs is a very useful testing and/or debugging tool in a general concurrent programming environment. It may become a common basis for development of testing and/or debugging tools with various aims in the environment.

2) When we develop an execution monitor and/or debugger for concurrent programs, it is necessary to define its monitoring domain and to make it transparent to the monitored concurrent behavior of the target program.

3) When we intend to specify and/or describe the behavior of a concurrent program at a high abstraction level, the concurrent events are more suitable as primitive entities than process states are.

4) The lattice theory is a powerful tool to abstract and formalize the properties of concurrent programs because of the simplicity of its concept and the consistency with some inherent properties of the programs.

References

[Andrews-82] Andrews, G. R. : The Distributed Programming Language SR - Mechanisms, Design, and Implementation, Software - Practice and Experience, Vol. 12, No. 8, pp. 719-754, 1982.

[Baiardi-86] Baiardi, F. , De Francesco, N. , Vaglini, G. : Development of a Debugger for a Concurrent Language, IEEE Transactions on Software Engineering, Vol. SE-12, No. 4, pp. 547-553, 1986.

[Booch-82] Booch, G. : Dear Ada, ACM Ada Letters, Vol. 2, No. 3, pp. 10-13, 1982.

[Brinch Hansen-73] Brinch Hansen, P. : Concurrent Programming Concepts, ACM Computing Surveys, Vol. 5, No. 4, pp. 223-245, 1973.

[Brinch Hansen-78] Brinch Hansen, P. : Distributed Processes: A Concurrent Programming Concept, Communications of the ACM, Vol. 21, No. 11, pp. 934-941, 1978.

[Cheng-86] Cheng, J. and Ushijima, K. : Assigning Unique Identifiers to Ada Tasks, Technology Report of IEICE, Vol. 86, No. 258, pp. 39-46, COMP86-55, 1986 (in Japanese).

[Cheng-87a] Cheng, J., Araki, K., and Ushijima, K. : Event-Driven Execution Monitor for Ada Tasking Programs, Proc. COMPSAC 87, pp. 381-388, 1987.

[Cheng-87b] Cheng, J., Araki, K., and Ushijima, K. : Lattice Model for Execution Histories of Concurrent Ada Programs and Partial Order Transparency in Execution Monitoring, Technical Report CSCE-87-C33, Department of Computer Science and Communication Engineering, Kyushu University, October 1987.

[Cheng-87c] Cheng, J., Araki, K., and Ushijima, K. : Detecting Communication Deadlocks in Ada Tasking Programs, Submitted to Transactions of IPS Japan, 1987 (in Japanese).

[Cheng-88] Cheng, J. and Ushijima, K. : Modeling the Ada Tasking Using Extended Petri Nets, to appear in Memoirs of the Faculty of Engineering, Kyushu University, Vol. 48, No. 1, 1988.

[DGC-84a] Data General Corp. : Ada Development Environment (ADE) (AOS/VS) User's Manual, 1984.

[DGC-84b] Data General Corp. : Ada Source Code Debugger (ADE) (AOS/VS) Reference Manual, 1984.

[DoD-80] United States Department of Defense : "STONEMAN", Requirements for Ada Programming Support Environment, Feb. 1980.

[DoD-83] United States Department of Defense : Reference Manual for the Ada Programming Language (ANSI/MIL-STD-1815A), Jan. 1983.

[Gait-85] Gait, J. : A Debugger for Concurrent Programs, Software – Practice and Experience, Vol. 15, No. 6, pp. 539-554, 1985.

[German-84] German,S.M. : Monitoring for Deadlock and Blocking in Ada Tasking, IEEE Transactions on Software Engineering, Vol. SE-10, No. 6, pp. 764-777, 1984.

[Helmbold-85a] Helmbold,D. and Luckham,D. : Debugging Ada Tasking Programs, IEEE Software, Vol. 2, No. 2, pp. 47-57, 1985.

[Helmbold-85b] Helmbold,D. and Luckham,D.C. : Runtime Detection and Description of Deadness Errors in Ada Tasking, ACM Ada Letters, Vol. 4, No. 6, pp. 60-72, 1985.

[Holdsworth-83] Holdsworth, D. : A System for Analysing Ada Programs at Run-time, Software - Practice and Experience, Vol. 13, pp. 407-421, 1983.

[Jefferson-85] Jefferson, D. R. : Virtual Time, ACM Transactions on Programming Languages and Systems , Vol. 7, No. 3, pp. 404-425, 1985.

[Lamport-78] Lamport, L. : Time, Clocks, and Ordering of Events in a Distributed System, Comm. ACM, Vol. 21, No. 7, pp. 558-565, 1978.

[LeDoux-85] LeDoux, C. H. and Parker, Jr. D. S. : Saving Traces for Ada Debugging, Ada in Use, pp. 97-108, Proc. of the Ada International Conference, Paris May 1985.

[Maio-85] Maio, A. D. , Ceri, S. and Reghizzi, S. C. : Execution Monitoring and Debugging Tool for Ada Using Relational Algebra, Ada in Use, pp. 109-123, Proc. of the Ada International Conference, Paris May 1985.

[Mauger-85] Mauger, C. and Pammett, K. : An Event-driven Debuger for Ada, Ada in Use, pp. 109-123, Proc. of the Ada International Conference, Paris May 1985.

[Plattner-81] Plattner, B. and Nievergelt, J. : Monitoring Program Execution: A Survey, IEEE Computer, Vol. 14, No. 11, pp. 76-93, 1981.

[Powell-83] Powell, M. L. and Linton, M. A. : A Database Model of Debugging, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, ACM Software Engineering Notes, Vol. 8, No. 4, SIGPLAN Notices, Vol. 18, No. 8, pp. 67-70, 1983.

[Seviora-87] Seviora, R. E. : Knowledge-Based Program Debugging Systems, IEEE Software, Vol. 4, No. 3, pp. 20-32, 1987.

[Smith-85] Smith, E. J. : A Debugger for Message-based Processes, Software – Practice and Experience, Vol. 15, No. 11, pp. 1073-1086, 1985.

[Tai-85] Tai, K-C. : On Testing Concurrent Programs, Proc. COMPSAC 85, pp. 310-317, 1985.